Linear algebra, signal processing, and wavelets. A unified approach. Python version

Øyvind Ryan

Feb 19, 2016

Preface

The starting point for this book was a new course developed at the University of Oslo, called "Applications of Linear Algebra". This was given for the first time in 2012. At the university we had recognized that students who just had their first course in linear algebra already have the right background to learn about several important and interesting topics in signal processing and wavelet theory. Unfortunately, most textbooks on these subjects are written in a language which does not favor a basic background in linear algebra. This makes much literature unavailable to a large class of students, and only available to engineering- and signal processing students. Moreover, it is not a common textbook strategy to introduce signal processing and wavelets together from scratch, even though the two can very much motivate each other. Why not write such a self-contained textbook, where linear algebra is the main fundament? This question is the motivation behind this book.

Some examples on where many signal processing textbooks fail in refering to linear algebra are:

- Matrix notation is often absent. Instead, linear operations are often expressed by component formulas, and matrix multiplication is instead refered to as convolution (when filters are used).
- Many operations, which really represent change of coordinates, such as the DFT, the DCT, and the DWT, are not represented as such. These operations are thus considered outside a linear algebra framework, so that one does not use tools, notation, and results from linear algebra for them.
- Eigenvalues and eigenvectors are not mentioned, even if these often are at play behind the scene: It is often not mentioned that the Fourier basis vectors are eigenvectors for filters, with the frequency response being the corresponding eigenvalues. Also, the property for filters that convolution in time corresponds to multiplication in frequency can in linear algebra terms be summarized by that the frequency representation is obtained from diagonalization, so that multiplication in frequency corresponds to multiplying diagonal matrices where the frequency responses are on the diagonal.
- Function spaces are often not put into a vector space/inner product context, even if Fourier series can be seen as a least squares approximation on the

Fourier spaces, and the Fourier integrals for the Fourier coefficients can be seen as the inner product coefficients of the orthogonal decomposition formula.

Several other books have also seen the need for writing new textbooks which exploit linear algebra. One example is [33], which goes further in using matrix notation than many signal processing textbooks. Still, the author feels that this book and others should do even more (such as addressing the issues above) to integrate a linear algebra framework, so that students feel more at home when they have a basic linear algebra background. As an example, it seems that many textbooks refer to matrices with polynomial entries, something which stems from signal processing and the Z-transform. We will see that this is unnecessary, as one can identify the polynomial entries with Toeplitz matrices, and such "non-standard matrices" confuse students.

This book is an introduction to Fourier analysis and signal processing (first part of the book) and wavelets (second part of the book), assuming only a beginning course in linear algebra. Without such a course, the value in this book is limited. An appendix has been included so that students can repeat the linear algebra background they need, but a full course on these topics is preferred in order to follow the contents of the book. This book is fitting for use in one or two university level undergraduate courses, and is perhaps best directly after a beginning linear algebra course. Also, some of the theory from a beginning course in linear algebra is further developed: Complex vector spaces and inner products are considered (many introductory linear algebra textbooks concentrate only on real vector spaces and inner product spaces). Also, while many introductory linear algebra textbooks consider inner product spaces which are function spaces, they often do not go very far in training the student on these spaces. This book goes longer in this respect, in that both Fourier and wavelet function spaces are heavily used, both in theory and exercises. The book also builds more intuition on changes of coordinates, such as the DFT, the DCT, and the DWT, and the basic properties of these operations. The book itself can thus be seen as an extension to a linear algebra textbook. In the future, the author hopes that this material can become additional chapters in a new linear algebra textbook, or as part of a general learning package comprising of different theory integrated together.

Since a linear algebra background is assumed, and this is the common denominator between the presented topics, some with signal processing background may feel excluded. In particular, signal processing nomenclature is not used. To also make this book accessible for these students, we have included several comments in the various chapters, which may help to unify the understanding from a signal processing and a linear algebra perspective. We have also included another appendix which can serve as a general translation guide between linear algebra and signal processing.

This book has been written with a clear computational perspective. The theory motivates algorithms and code, for which many programming issues need to be addressed. A central idea is thus to elaborate on the interplay between

theory, numerical methods, and applications: We not only want to explain the theoretical foundations of the selected topics, but also to go from there to numerical methods, and finally motivate these by their applications in diverse fields. The book goes a long way in integrating important applications, such as modern standards for compression of sound (MPEG) and images (JPEG, JPEG2000). In some respects we go longer than other books with the name "applications" in their title: many books on linear algebra sneak in words like "applied" or "applications" in their title. The main contents in most of these books may still be theory, and particular applications where the presented theory is used are perhaps only mentioned superficially, without digging deep enough to explain how these applications can be implemented using the presented theory.

Implementations and important algorithms related to the presented theory are presented throughout the book, and not only as isolated exercises without corresponding examples. There is a focus on implementation and good coding practice, algorithms and the number of arithmetic operations they need, memory usage, and opportunities for parallel computing. We unveil how we can turn the theory into practical implementations of the applications, in order to make the student operational. By "Practical implementation" we do not mean a "full implementation", which typically involves many other components, unrelated or only weakly related to the theory we present. This focus on the computational perspective has been inspired by the project "Computing in science education" at the University of Oslo, which is an initiative to integrate computations into the basic science curriculum from the very first semester at the university. Wavelet theory in particular has a large amount of detail incorporated into it. Much literature skip some of these details, in order to make a simpler presentation. In this book we have attempted not to skip details for the sake of completeness, but attempted to isolate tricky details from the ideas. We attempt to point out which details can be skipped for the introductory reader, but the interested reader still has the opportunity to go through all details by following all aspects of the book. There are many more topics which could have been included in this book, but much of these would require more detail. We have therefore chosen a stopping point which seems to be a good compromise with the level of detail one needs to go into.

Programming. It is assumed that the student already has been introduced to some programming language or computational tool. It is to prefer that the student has taken a full course in programming first, since the book does not give an introduction to primitives such as `for`-loops, conditional statements, lists, function definitions, file handling, and plotting. At the University of Oslo, most students take such a Python-based course the first semester where such primitives are gone through.

This book comes in two versions: One where Matlab programming is used throughout, and one where Python programming is used throughout. The version of the book you are reading uses Python. If you search the internet for recommendations about what programming language to use in a basic course in

a linear algebra, you may find comments such as "python is too advanced for such a beginning course", or "Matlab is much quicker to get started with". The author believes that such comments would not have been posted if all students received the same training in Python programming the first semester as they do at the university of Oslo: Once educational institutions agree on a common framework for programming for the students, we believe that the programming you see in this book will not feel too advanced, irrespective of which version of this book you have.

A code repository accompanies the book, where all Matlab and Python code in the book can be found. The code repository can be found on the webpage for the book, and contains the following:

- IPython notebooks and Matlab publish files which list the examples in the book in such a way that they can be run sequentially. The example code within the book may not run on its own, as it may rely on importing certain packages, or defining certain variables. These imports and definitions will in any way be part of the notebook. Each chapter lists the notebooks the examples can be found in, and there is typically one notebook per chapter. The notebooks also contains a lot of solution code to the exercises. This code is also found in the solution manual.

- Function libraries which are developed throughout the book. The most notable of these are the FFT and DWT libraries. The book may list simplified versions of these which the students are asked to extend to more general implementations in the exercises, so that for instance the code is valid for sound with any number of channels, or images with any number of color components. The solution manual may then list the full versions of these functions, as they appear in the code repository. In the book, we always list what modules the referred functionality can be found in.

- Documentation for all functions. As the student often is asked to implement much of the functionality himself, this documentation is a good source to ensure that he interprets input and output parameters and return values correctly.

- Test code for the functions we develop.

- IPython notebooks and Matlab publish files which generate all the figures in the book. Some exercises in the book also aim at generating these figures. The solution manual refers to these notebooks in this case. The figures in the printed version of the book have been generated with Python and Matplotlib.

If you compare the code in the Matlab and Python versions of the book, you will see that the programming style is such that much of the code in the two languages is very similar: Function signatures are almost the same. Code indentation follows the Python standard, where indentation is an important part of the syntax. Classes could have been used several places in the Python code, but

this has been avoided, since they are not supported in Matlab. Much of the programming syntax is also similar.

There are also some differences in the Matlab and Python versions, however. The python code is structured into modules, a very important structuring concept in Python. In Matlab the concept of a module does not exist. Instead functions are placed in different files, rather than in modules, and this leads to a high number of files in the Matlab part of the code repository. For the Python version, each chapter states what modules the accompanying functionality are part of. In Python, it is customary to place test code in the same module as the functions being tested. This can't be done with Matlab, so the compromise made is to create a separate file with the test code, and where there is a main function which calls different test functions in the same file, with these test functions following the Python naming conventions for test functions.

Another difference has to do with that Matlab copies input and return parameters whenever they are accessed. This is not the case in Python, where input and return parameters are passed by reference. This means that we can perform in-place computation in Python code, i.e. we can write the result directly into the input buffer, rather than copying it. As this can lead to much more efficent code, the Python code attempts to perform in-place operations wherever possible, contrary to the Matlab code where this approach is not possible. This affects the signatures of many functions: Several Python functions have no return values since the result is written directly into an input buffer, contrary to the Matlab counterparts which use return parameters for the result.

Matlab has built-in functionality for reading and writing sound and images, as well as built-in functionality for playing sound and displaying images. To make the Python code similar to the Matlab code, the code repository includes the modules `sound` and `images`, with functions with similar signatures to the Matlab counterparts. These functions simply call Python counterparts, in such a way that the interface is the same.

Although the code repository contains everything developed in the book, it is recommended that the student follows the code development procedure of the book, and establishes as much code as possible on his own. In this way he is guided through the development of a full library, which is general in purpose. The student is encouraged to create his own files where the functions have the same signatures as the fully developed functions in the code repository. To ensure that the student's functions are run, rather than the functions in the code repository, it is important that the student's files are listed first in the path. With Python the student can also place his code in separate modules, and override code in the modules in the code repository.

This book has been typeset in a sophisticated language called `doconce`. Due to this tool we have been able to use a single source text to contain both Matlab and Python versions, with a minimum of duplication. Another consequence of the doconce tool is that we also can generate this book also comes in HTML and sphinx versions. The doconce tool has also been used to generate the ipython notebooks and matlab publish source.

Structure of the book. Part 1 of the book (chapter 1-4) starts with a general discussion on what sound is. Chapter 1 also introduces Fourier series as a finite-dimensional model for sound, and establishes the mathematics for computing and analyzing Fourier series. While the first chapter models sound as continuous functions, the chapter 2 moves on to digital sound. Now sound is modeled as finite-dimensional vectors, and this establishes a parallel theory, where the computation of Fourier series is replaced with a linear transformation called the Discrete Fourier Transform. Two important topics are gone through in connection with this: Fast computation of the Discrete Fourier Transform, and the sampling theorem, which establishes a connection between Fourier series and the Discrete Fourier Transform. In chapter 3 we look at a general type of operations on sound called filters. When applied to digital sound, it turns out that filters are exactly those operations which are diagonalized by the Discrete Fourier Transform. Finally, the chapter 4 ends the first part of the book by looking at the Discrete Cosine Transform, which can be thought of as a useful variant of the Discrete Fourier Transform.

Part 2 of the book of the book starts with a motivation for introducing what is called wavelets. While the first part of the book works on representations of sound in terms of frequency only, wavelets take into account that such a representation may change with time. After motivating the first wavelets and setting up a general framework in chapter 5, chapter 6 establishes the connection between wavelets and filters, so that the theory from the first part of the book can be applied. In chapter 7 we establish theory which is used to construct useful wavelets which are used in practice, while chapter 8 goes through implementation aspects of wavelets, and establishes their implementations. Chapter 9 takes a small step to the side to look at how we can experiment with images, before we end the book in chapter 10 with setting up the theory for wavelets in a two-dimensional framework, so that we can can use them to experiment with images.

Assumptions. This book makes some assumptions, which are not common in the literature, in order to adapt the exposition to linear algebra. The most important one is that most spaces are considered finite-dimensional, and filters are considered as finite-dimensional operations. In signal processing literature, filters are usually considered as infinite-dimensional operations. In generality this is true, but in practice one rarely encounters an infinite set of numbers, which justifies our assumptions. Since Fourier analysis implicitly assumes some kind of periodicity, we are left with the challenge of extending a finite signal to a periodic signal. This can be done in several ways, and we go through the two most important ways, something which is often left out in signal processing literature.

New contributions. It would be wrong to say that this book provides new results. But it certainly provides some new proofs for existing results, in order

to make the results more accessible for a linear algebra point of view. Let us mention some examples.

- The sampling theorem, which is proved in more generality with more advanced Fourier methods than is presented here (i.e. with Continuous-time and Discrete-time Fourier transforms), has been restricted to periodic functions, for which a much simpler proof can be found, fitting our context.
- The DCT and its orthogonality is found in a constructive way.
- The quadrature formula for perfect reconstruction is reproved in simple linear algebra terms.

What has been omitted. In the book, analytical function-theoretical proofs have been avoided. We do not define the Continuous-time and Discrete-time Fourier transforms. We do give a short introduction into analog filters, however, as they are useful in explaining properties for their digital counterparts. We do not define the Z-transform, and we do not make filter design based on the placement of poles and zeros, as is common in signal processing literature.

Notation. We will follow linear algebra notation as you know it from classical linear algebra textbooks. In particular, vectors will be in boldface ($\boldsymbol{x}$, $\boldsymbol{y}$, etc.), while matrices will be in uppercase (A, B, etc.). The zero vector, or the zero matrix, is denoted by $\mathbf{0}$. All vectors stated will be assumed to be column vectors. A row vector will always be written as $\boldsymbol{x}^T$, where $\boldsymbol{x}$ is a (column) vector. We will also write column vectors as $\boldsymbol{x} = (x_0, x_1, \ldots, x_n)$, i.e. as a comma-separated list of values.

How to use this book. Note that this is a curricular book, not an encyclopaedia for researchers. Besides the theory, the focus is on drilling the theory with exercises. Each chapter also has a list of minimum requirements, which may be helpful in preparation for exams.

There are many important topics in the exercises in this book, which are not gone through in the text of the book. A detailed solution manual for many of these exercises can be found on the web page of the book. These solutions represents important material which is theoretical material in many other books. It is recommended that these solutions be used with care. Much of the learning outcome depends on that the students try and fail for some time in solving exercises. They should therefore not take the shortcut directly to the solutions: Although they may understand the solution to an exercise in this way, they may not learn the thinking process on how to arrive at that solution, and how to solve it logically and understandably.

The entire book is too much for a one-semester course. Two semesters should suffice to go through everything. There are several different ways material can be chosen so that the amount fits a one-semester course. Most material in chapters 2

and 3 can be gone through independently of Chapter 1, in that one sacrifices the motivation of this material in ternms of analog filters. Chapter 4 can be skipped. Chapter 5 can be read independently from the first part of the book. The same applies for Chapter 9. Chapters 9, and 10 can be omitted if time is scarce, since they are the only chapters which concentrate on images and the two-dimensional perspective.

Acknowledgment. Thanks to Professor Knut Mørken for input to early versions of this book, and suggesting for me to use my special background to write literature which binds linear algebra together with these interesting topics. A special thanks also to thank Andreas Vågvang Solbrå for his valuable contributions to the notes, both in reading early and later versions of them, and for teaching and following the course. Thanks also to students who have taken the course, who have provided valuable feedback on how to present the topics so that they understand them. I would also like to thank all participants in the CSE project at the University of Oslo, for their continuous inspiration.

Øyvind Ryan

Oslo, January 2014.

Contents

List of Exercises

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.

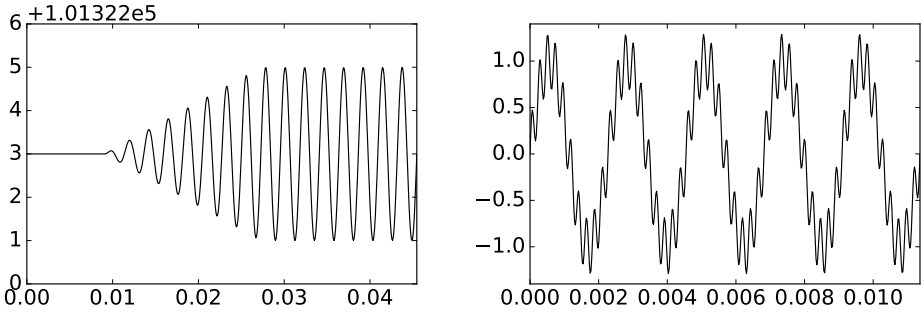

Figure 1.1: Two examples of audio signals. In terms of air pressure (left), and in terms of the difference from the ambient air pressure (right).

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.

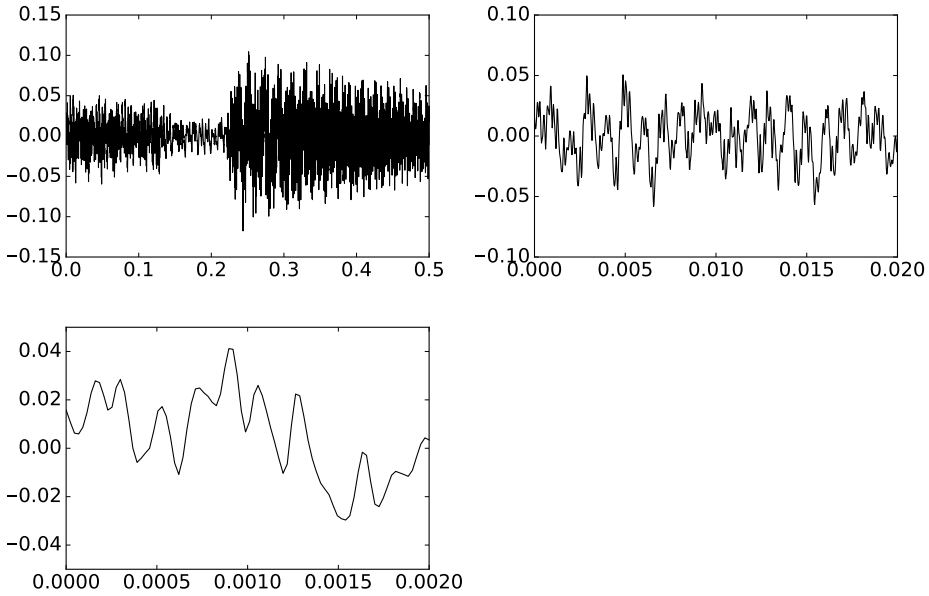

Figure 1.2: Variations in air pressure during parts of a song. The first 0.5 seconds, the first 0.02 seconds, and the first 0.002 seconds.

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.

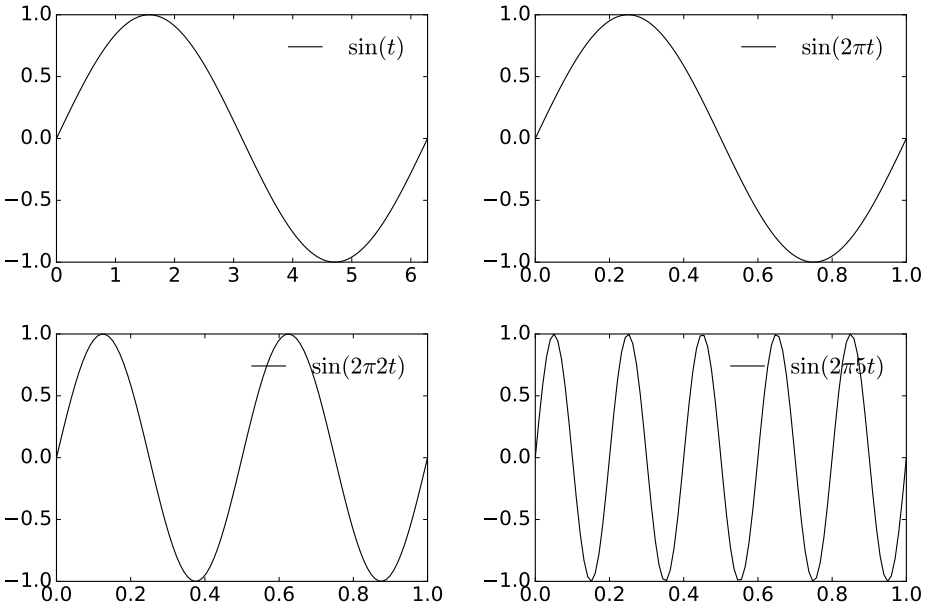

Figure 1.3: Versions of sin with different frequencies.

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.

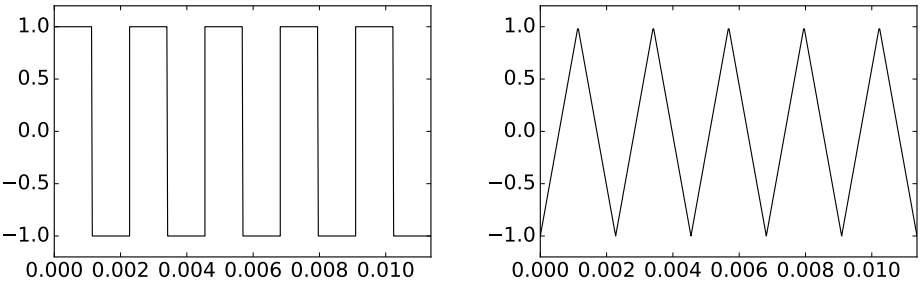

Figure 1.4: The first five periods of the square wave and the triangle wave, two functions with regular oscillations, but which are not simple, trigonometric functions.

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

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

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.

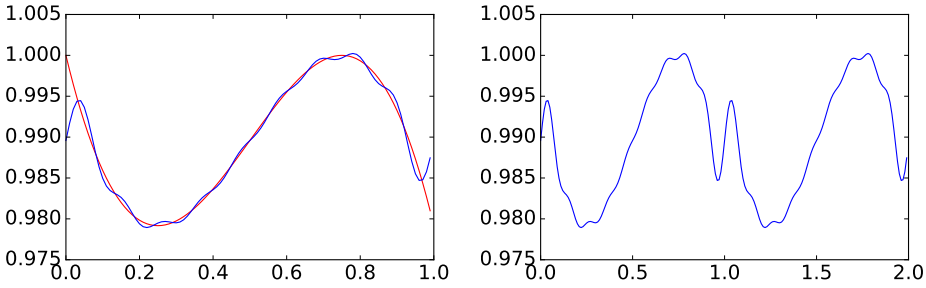

Figure 1.5: The cubic polynomial $f(x) = -\frac{1}{3}x^3 + \frac{1}{2}x^2 - \frac{3}{16}x + 1$ on the interval $[0, 1]$, together with its Fourier series approximation from $V_{9,1}$. The function and its Fourier series is shown left. The Fourier series on a larger interval is shown right.

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(shape(t))
for n in range(1,20,2):
    y = y + (4/(n*pi))*sin(2*pi*n*t/T)
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.

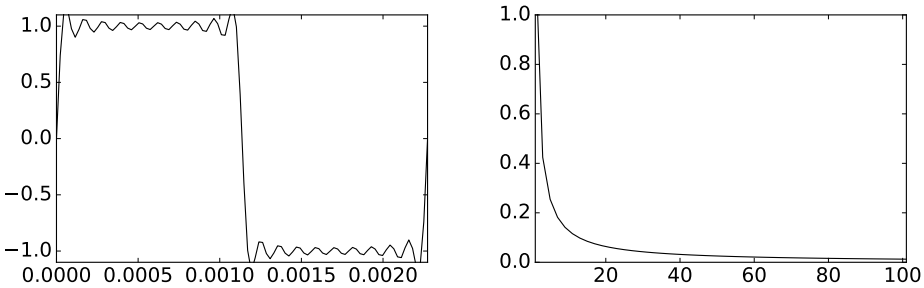

Figure 1.6: The Fourier series with $N = 20$ for the square wave of Example 1.17, and the values for the first 100 Fourier coefficients b_n.

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

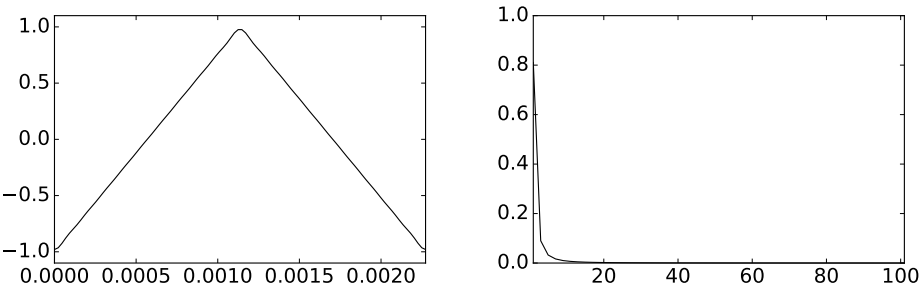

Figure 1.7: The Fourier series with $N = 20$ for the triangle wave of Example 1.18 and the values for the first 100 Fourier coefficients a_n.

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 $y_{-N}, \dots, y_0, \dots, y_N$ 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.

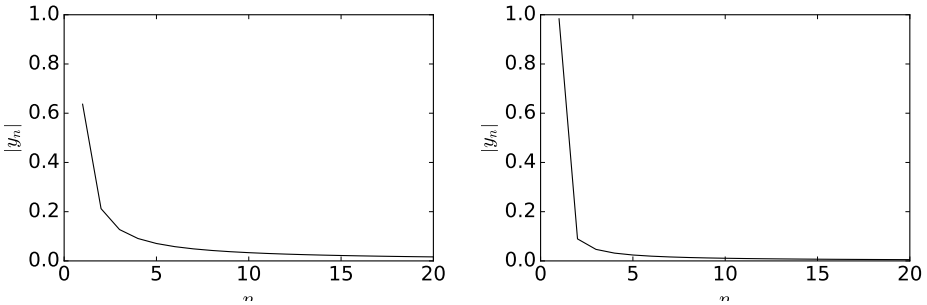

Figure 1.8: Plot of $|y_n|$ when $f(t) = e^{2\pi i t/T_2}$, and $T_2 > T$. Left: $T/T_2 = 0.5$. Right: $T/T_2 = 0.9$.

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.

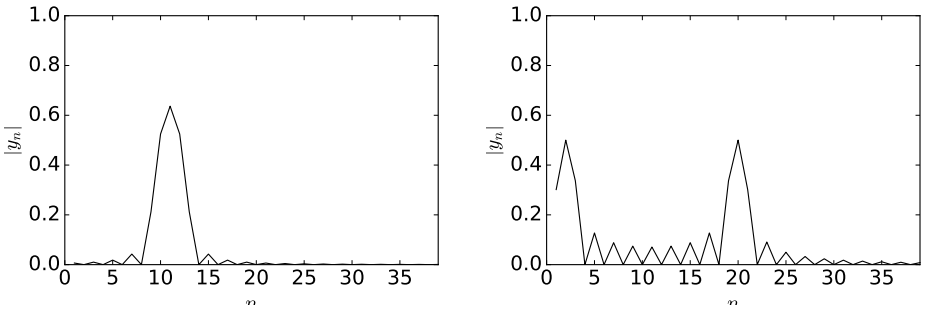

Figure 1.9: Plot of $|y_n|$ when we have two different pure tones at the different parts of a period. Left: $n_1 = 10$, $n_2 = 12$. Right: $n_1 = 2$, $n_2 = 20$.

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, 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) in the compendium, 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) in the compendium 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)?

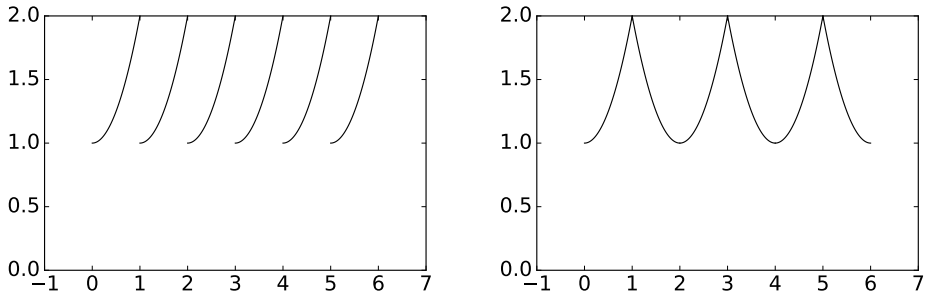

Figure 1.10: Two different extensions of f to a periodic function on the whole real line. Periodic extension (left) and symmetric extension (right).

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.

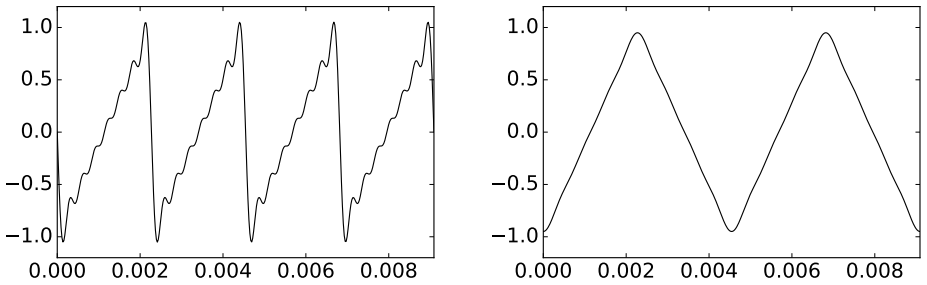

Figure 1.11: The Fourier series with $N = 7$ terms of the periodic (left) and symmetric (right) extensions of the function in Example 1.36.

- 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

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

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 one 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.ipynb`. Functionality for accessing sound are collected in a module called `sound`.

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 will use the function `play(x, fs)` in the module `sound` for this. 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:

```
play(x, fs)
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 = fs/f
oneperiod = hstack([ones((samplesperperiod/2),dtype=float), \
                    -ones((samplesperperiod/2),dtype=float)])
```

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 = tile(oneperiod, antsec*f)
play(x, fs)
```

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 = hstack([linspace(-1, 1, samplesperperiod/2), \
                    linspace(1, -1, 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

```
play(x[:, 1], fs);
```

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

```
play(x, fs)

play(x, 2*fs)

play(x, fs/2)
```

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

Example 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*random.random(shape(x))-1)
z /= abs(z).max()
```

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

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

This corresponds to the sound in the left plot of Figure 1.1, 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 right plot of Figure 1.1 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 $\boldsymbol{x}$.

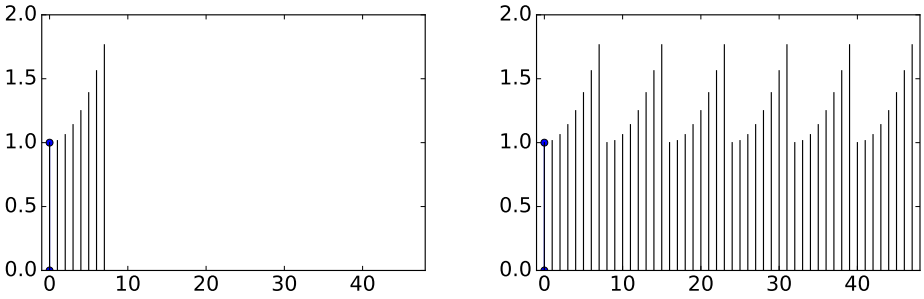

Figure 2.1: A vector and its periodic extension.

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 $\sqrt{N}$). 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

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

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

```
def DFTImpl(x):
    y = zeros_like(x).astype(complex)
    N = len(x)
    for n in xrange(N):
        D = exp(-2*pi*n*1j*arange(float(N))/N)
        y[n] = dot(D, x)
    return y
```

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 Python may have an optimized way for computing this. Note that `dot` in Python does not conjugate

any of the components, contrary to what we do in our definition of a complex inner product. 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. Python 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 $(\widehat{\boldsymbol{z}})_n = e^{-2\pi i dn/N}(\widehat{\boldsymbol{x}})_n$.
5. If d is an integer and $\boldsymbol{z}$ is the vector with components $z_k = e^{2\pi i dk/N}x_k$, 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

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

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

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

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 $e^{2\pi i k d/N}$ 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:

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

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

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

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

In particular, the total contribution in f from frequency n/T, for $0 \le n \le M$, 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

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

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.

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.

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

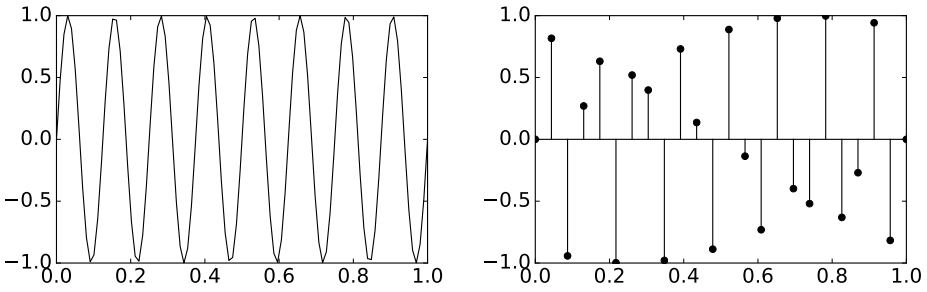

Figure 2.2: An example on how the samples are picked from an underlying continuous time function (left), and the samples on their own (right).

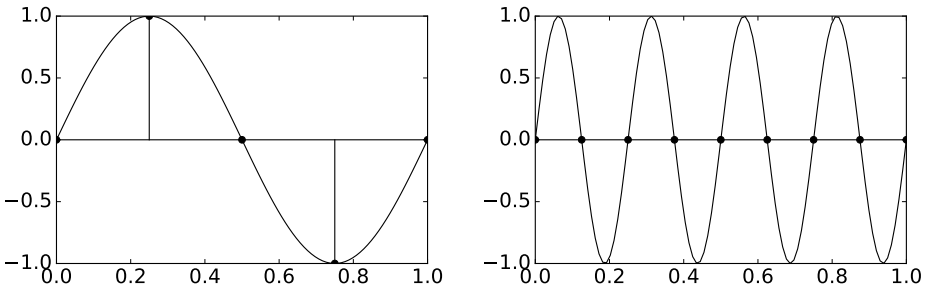

Figure 2.3: Sampling $\sin(2\pi t)$ with two points (left), and sampling $\sin(2\pi 4t)$ with eight points (right).

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.

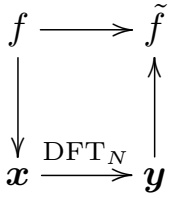

Figure 2.4: How we can interpolate f from $V_{M,T}$ with help of the DFT. The left vertical arrow represents sampling. The right vertical arrow represents interpolation, i.e. computing Equation (2.9).

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:

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

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

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

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). 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 = shape(x)[0]
# Zero out higher frequencies
y = fft.fft(x, None, 0)
y[(L+1):(N-L)] = 0;
newx = fft.ifft(y)
```

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. 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

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

$$|y_n| = \left|\sum_{k=0}^{N-1} e^{2\pi i k n_0/M} e^{-2\pi i k n/N}\right| = \left|\sum_{k=0}^{N-1} e^{2\pi i k(n_0/M - n/N)}\right|$$
$$= \left|\frac{1 - e^{2\pi i N(n_0/M - n/N)}}{1 - e^{2\pi i (n_0/M - n/N)}}\right| = \left|\frac{\sin(\pi N(n_0/M - n/N))}{\sin(\pi(n_0/M - n/N))}\right|. \tag{2.12}$$

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.

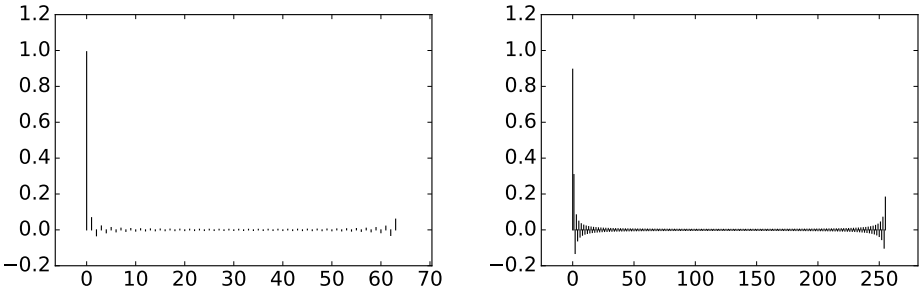

Figure 2.5: The frequency representation obtained when restricting to a block of size N of the signal, for $N = 64$ (left), and $N = 256$ (right)

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 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.fft(x, None, 0)
y = (abs(y) >= threshold)*y
newx = fft.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 for 512 sound samples from a song.

The samples of the sound and (the absolute value of) its DFT are shown at the top. At the bottom all values of the DFT with absolute value smaller than 0.02 are set to zero (52) values then remain), and the sound is reconstructed with the IDFT, and then shown in. The start and end signals look similar, even though the last signal can be represented with less than 10 % of the values from the first.

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.*

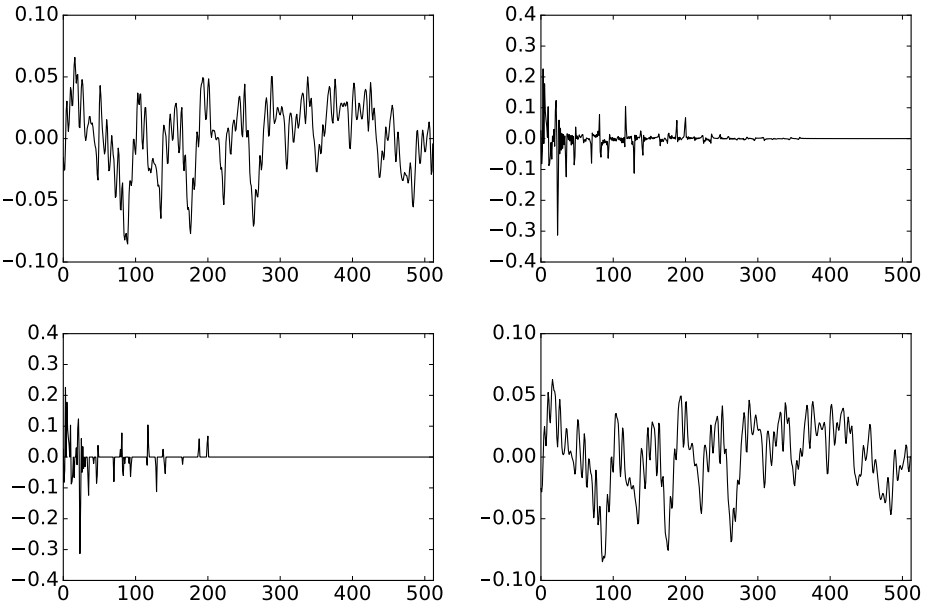

Figure 2.6: Experimenting with the DFT on a small part of a song.

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.fft(x, None, 0)
y *= 2**n
y = round_(y)
y /= float(2**n)
newx = fft.ifft(y)
```

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 = x[0:2**17]
y = fft.fft(x, None, 0)
y[(2**17/4):(3*2**17/4)] = 0
newx = abs(fft.ifft(y))
newx /= abs(newx).max()
play(newx, fs)
```

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, and that the number of sound samples is $N = 292570$. 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 Proposition 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.

FFT-based functionality is collected in a module called `fft`.

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 $(D_{N/2})_{n,n} = e^{-2\pi in/N}$ for $0 \le n < N/2$. 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 $0 \le n \le N/2 - 1$. 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 $0 \le n < N/2$ we obtain Equation (2.13). For the second half of the DFT coefficients, i.e. $\{y_{N/2+n}\}_{0\le n\le N/2-1}$, we similarly have

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

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

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

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

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

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.

```
def FFTImpl(x, FFTKernel):
    bitreverse(x)
    FFTKernel(x)
```

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

```
def FFTKernelStandard(x):
    N = len(x)
    if N > 1:
        xe, xo = x[0:(N/2)], x[(N/2):]
        FFTKernelStandard(xe)
        FFTKernelStandard(xo)
        D = exp(-2*pi*1j*arange(float(N/2))/N)
        xo *= D
        x[:] = concatenate([xe + xo, xe - xo])
```

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:

```
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.

Python has built-in functions for computing the DFT and the IDFT using the FFT algorithm. These reside in the module `numpy`. 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 $f(N)$, 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

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

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

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

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 $N = N_1N_2$. Any time-index k can be written uniquely on the form N_1k+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 $\boldsymbol{x} \in \mathbb{R}^{N_1N_2}$. We denote by $\boldsymbol{x}^{(p)}$ the vector in $\mathbb{R}^{N_2}$ with entries $(\boldsymbol{x}^{(p)})_k = x_{N_1k+p}$. $\boldsymbol{x}^{(p)}$ 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:

```
def FFTImpl(x, FFTKernel, forward = True):
def 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 function `time()` in the `time` module 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 $N = N_1N_2$, and that $\boldsymbol{x}^{(p)} = \boldsymbol{0}$ 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) in the compendium for indices n and $N/2 - n$ as

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

We see here that, if we have computed the terms in y_n (which needs an additional 4 real multiplications, since $e^{-2\pi in/N}$ 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 (2.18) in the compendium to write a kernel function `FFTKernelNonrec` for a non-recursive FFT implementation. In your code, perform the matrix multiplications in Equation (2.18) in the compendium 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 $\mathrm{DFT}_{N/2}$ in Equation (2.17) in the compendium by using this equation again, to obtain

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

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 $e^{-2\pi in/N}$ 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 in range(N-1):
    print j
    m = N/2
    while (m >= 1 and j >= m):
        j -= m
        m /= 2
    j += m
```

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.

Chapter 3

Operations on digital sound: digital filters

In Section 1.5 we defined analog filters as operations on sound which preserved different frequencies. Such operations are important since they can change the frequency content in many ways. Analog filters can not be used computationally, however, since they are defined for all instances in time. As when we defined the DFT to make Fourier series computable, we would like to define *digital filters*, in order to make analog filters computable. It turns out that what we will define as digital filters can be computed by the following procedure:

$$z_n = \frac{1}{4}(x_{n-1} + 2x_n + x_{n+1}), \quad \text{for } n = 0, 1, \ldots, N-1. \tag{3.1}$$

Here $\boldsymbol{x}$ denotes the *input vector*, and $\boldsymbol{z}$ the *output vector*. In other words, the output of a digital filter is constructed by combining several input elements linearly. The concrete filter defined by Equation (3.1) is called a *smoothing filter*. We will demonstrate that it smooths the variations in the sound, and this is where it gets its name from. We will start this chapter by by looking at matrix representations for operations as given by Equation (3.1). Then we will formally define digital filters in terms of preservation of frequencies as we did for analog filters, and show that the formal definition is equivalent to such operations.

3.1 Matrix representations of filters

Let us consider Equation (3.1) in some more detail to get more intuition about filters. As before we assume that the input vector is periodic with period N, so that $x_{n+N} = x_n$. Our first observation is that the output vector $\boldsymbol{z}$ is also periodic with period N since

$$z_{n+N} = \frac{1}{4}(x_{n+N-1} + 2x_{n+N} + x_{n+N+1}) = \frac{1}{4}(x_{n-1} + 2x_n + x_{n+1}) = z_n.$$

The filter is also clearly a linear transformation and may therefore be represented by an $N \times N$ matrix S that maps the vector $\boldsymbol{x} = (x_0, x_1, \ldots, x_{N-1})$ to the vector $\boldsymbol{z} = (z_0, z_1, \ldots, z_{N-1})$, i.e., we have $\boldsymbol{z} = S\boldsymbol{x}$. To find S, for $1 \leq n \leq N-2$ it is clear from Equation (3.1) that row n has the value $1/4$ in column $n-1$, the value $1/2$ in column n, and the value $1/4$ in column $n+1$. For row 0 we must be a bit more careful, since the index -1 is outside the legal range of the indices. This is where the periodicity helps us out so that

$$z_0 = \frac{1}{4}(x_{-1} + 2x_0 + x_1) = \frac{1}{4}(x_{N-1} + 2x_0 + x_1) = \frac{1}{4}(2x_0 + x_1 + x_{N-1}).$$

From this we see that row 0 has the value $1/4$ in columns 1 and $N-1$, and the value $1/2$ in column 0. In exactly the same way we can show that row $N-1$ has the entry $1/4$ in columns 0 and $N-2$, and the entry $1/2$ in column $N-1$. In summary, the matrix of the smoothing filter is given by

$$S = \frac{1}{4}\begin{pmatrix} 2 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ 1 & 2 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 2 & 1 \\ 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 2 \end{pmatrix}. \tag{3.2}$$

A matrix on this form is called a Toeplitz matrix. The general definition is as follows and may seem complicated, but is in fact quite straightforward:

Definition 3.1. *Toeplitz matrices.*

An $N \times N$-matrix S is called a Toeplitz matrix if its elements are constant along each diagonal. More formally, $S_{k,l} = S_{k+s,l+s}$ for all nonnegative integers k, l, and s such that both $k+s$ and $l+s$ lie in the interval $[0, N-1]$. A Toeplitz matrix is said to be circulant if in addition

$$S_{(k+s) \bmod N,(l+s) \bmod N} = S_{k,l}$$

for all integers k, l in the interval $[0, N-1]$, and all s (Here mod denotes the remainder modulo N).

Toeplitz matrices are very popular in the literature and have many applications. A Toeplitz matrix is constant along each diagonal, while the additional property of being circulant means that each row and column of the matrix 'wraps over' at the edges. Clearly the matrix given by Equation (3.2) satisfies Definition 3.1 and is a circulant Toeplitz matrix. A Toeplitz matrix is uniquely identified by the values on its nonzero diagonals, and a circulant Toeplitz matrix is uniquely identified by the values on the main diagonal, and on the diagonals above (or under) it. Toeplitz matrices show up here in the context of filters, but they will also show up later in the context of wavelets.

Equation (3.1) leads us to the more general expression

$$z_n = \sum_k t_k x_{n-k}. \tag{3.3}$$

If $\boldsymbol{t}$ has infinitely many nonzero entries, the sum is an infinite one, and may diverge. We will, however, mostly assume that $\boldsymbol{t}$ has a finite number of nonzero entries. This general expression opens up for defining many types of operations. The values t_k will be called *filter coefficients*. The range of k is not specified, but is typically an interval around 0, since z_n usually is calculated by combining x_k's with indices close to n. Both positive and negative indices are allowed. As an example, for formula (3.1) k ranges over $-1, 0$, and 1, and we have that $t_{-1} = t_1 = 1/4$, and $t_0 = 1/2$. Since Equation (3.3) needs to be computed for each n, if only $t_0, \ldots, t_{kmax}$ are nonzero, we need to go through the following `for`-loop to compute $z_{kmax}, \ldots, z_{N-1}$:

```
z = zeros_like(x)
for n in range(kmax,N):
    for k in range(kmax + 1):
        z[n] += t[k]*x[n - k]
```

It is clearly possible to vectorize the inner loop here, since it takes the form of a dot product. Another possible way to vectorize is to first change the order of summation, and then vectorize as follows

```
z = zeros_like(x)
for k in range(kmax + 1):
    z[kmax:N] += t[k]*x[(kmax-k):(N-k)]
```

Depending on how vectorization is supported, this code will in general execute faster, and is to prefer. The drawback, however, is that a filter often is applied in real time, with the output computed only when enough input is available, with the input becoming available continuously. This second approach then clearly fails, since it computes nothing before all input is available. In the exercise we will compare the computation times for the two approaches above, and compare them with a built-in function which computes the same.

Note that above we did not consider the first entries in $\boldsymbol{z}$, since this is where the circulation occurs. Taken this into account, the first filter we considered in this chapter can be implemented in vectorized form simply as

```
z[0]       = x[1]/4. + x[0]/2. + x[N-1]/4.
z[1:(N-1)] = x[2:N]/4. + x[1:(N-1)]/2. + x[0:(N-2)]/4.
z[N-1]     = x[0]/4. + x[N-1]/2. + x[N-2]/4.
```

In the following we will avoid such implementations, since `for`-loops can be very slow in Python. We will see that an efficient built-in function exists for computing this, and use this instead.

By following the same argument as above, the following is clear:

Proposition 3.2. *Filters as matrices.*

Any operation defined by Equation (3.3) is a linear transformation which transforms a vector of period N to another of period N. It may therefore be represented by an $N \times N$ matrix S that maps the vector $\boldsymbol{x} = (x_0, x_1, \ldots, x_{N-1})$ to the vector $\boldsymbol{z} = (z_0, z_1, \ldots, z_{N-1})$, i.e., we have $\boldsymbol{z} = S\boldsymbol{x}$. Moreover, the matrix S is a circulant Toeplitz matrix, and the first column $\boldsymbol{s}$ of this matrix is given by

$$s_k = \begin{cases} t_k, & \text{if } 0 \leq k < N/2; \\ t_{k-N} & \text{if } N/2 \leq k \leq N-1. \end{cases} \tag{3.4}$$

In other words, the first column of S can be obtained by placing the coefficients in (3.3) with positive indices at the beginning of $\boldsymbol{s}$, and the coefficients with negative indices at the end of $\boldsymbol{s}$.

This proposition will be useful for us, since it explains how to pass from the form (3.3), which is most common in practice, to the matrix form S.

Example 3.3. *Finding the matrix elements from the filter coefficients.*
Let us apply Proposition 3.2 to the operation defined by formula (3.1):

- for $k = 0$ Equation (3.4) gives $s_0 = t_0 = 1/2$.
- For $k = 1$ Equation (3.4) gives $s_1 = t_1 = 1/4$.
- For $k = N - 1$ Equation (3.4) gives $s_{N-1} = t_{-1} = 1/4$.

For all k different from 0, 1, and $N - 1$, we have that $s_k = 0$. Clearly this gives the matrix in Equation (3.2).

Proposition 3.2 is also useful when we have a circulant Toeplitz matrix S, and we want to find filter coefficients t_k so that $\boldsymbol{z} = S\boldsymbol{x}$ can be written on the form (3.3):

Example 3.4. *Finding the filter coefficients from the matrix.*
Consider the matrix

$$S = \begin{pmatrix} 2 & 1 & 0 & 3 \\ 3 & 2 & 1 & 0 \\ 0 & 3 & 2 & 1 \\ 1 & 0 & 3 & 2 \end{pmatrix}.$$

This is a circulant Toeplitz matrix with $N = 4$, and we see that $s_0 = 2$, $s_1 = 3$, $s_2 = 0$, and $s_3 = 1$. The first equation in (3.4) gives that $t_0 = s_0 = 2$, and $t_1 = s_1 = 3$. The second equation in (3.4) gives that $t_{-2} = s_2 = 0$, and $t_{-1} = s_3 = 1$. By including only the t_k which are nonzero, the operation can be written as

$$z_n = t_{-1}x_{n-(-1)} + t_0 x_n + t_1 x_{n-1} + t_2 x_{n-2} = x_{n+1} + 2x_0 + 3x_{n-1}.$$

Since the filter coefficients t_k uniquely define any $N \times N$-circulant Toeplitz matrix, we will establish the following shorthand notation for the filter matrix for a given set of filter coefficients. We will use this notation only when we have a finite set of nonzero filter coefficients (note however that many interesting filters in signal processing have infinitely many nonzero filter coefficients, see Section 3.5) Note also that we always choose N so large that the placement of the filter coefficients in the first column, as dictated by Proposition 3.2, never collide (as happens when N is smaller than the number of filter coefficients).

Definition 3.5. *Compact notation for filters.*

Let $k_{\min}$, $k_{\max}$ be the smallest and biggest index of a filter coefficient in Equation (3.3) so that $t_k \neq 0$ (if no such values exist, let $k_{\min} = k_{\max} = 0$), i.e.

$$z_n = \sum_{k=k_{min}}^{k_{max}} t_k x_{n-k}. \tag{3.5}$$

We will use the following compact notation for S:

$$S = \{t_{k_{min}}, \ldots, t_{-1}, \underline{t_0}, t_1, \ldots, t_{k_{max}}\}.$$

In other words, the entry with index 0 has been underlined, and only the nonzero t_k's are listed. k_{max} and k_{min} are also called the start and end indices of S. By the length of S, denoted $l(S)$, we mean the number $k_{max} - k_{min}$.

One seldom writes out the matrix of a filter, but rather uses this compact notation.

Example 3.6. *Writing down compact filter notation.*

Using the compact notation for a filter, we would write $S = \{1/4, \underline{1/2}, 1/4\}$ for the filter given by formula (3.1)). For the filter

$$z_n = x_{n+1} + 2x_0 + 3x_{n-1}$$

from Example 3.4, we would write $S = \{1, \underline{2}, 3\}$.

3.1.1 Convolution

Applying a filter to a vector $\boldsymbol{x}$ is also called taking the *convolution* of the two vectors $\boldsymbol{t}$ and $\boldsymbol{x}$. Convolution is usually defined without the assumption that the input vector is periodic, and without any assumption on the vector lengths (i.e. they may be sequences of inifinite length). The case where both vectors $\boldsymbol{t}$ and $\boldsymbol{x}$ have a finite number of nonzero elements dererves extra attention. Assume that $t_0, \ldots, t_{M-1}$ and $x_0, \ldots, x_{N-1}$ are the only nonzero elements in $\boldsymbol{t}$ and $\boldsymbol{x}$ (i.e. we can view them as vectors in $\mathbb{R}^M$ and $\mathbb{R}^N$, respectively). It is clear from the expression $z_n = \sum t_k x_{n-k}$ that only $z_0, \ldots, z_{M+N-2}$ can be nonzero. This motivates the following definition.

Definition 3.7. *Convolution of vectors.*

By the *convolution* of two vectors $\boldsymbol{t} \in \mathbb{R}^M$ and $\boldsymbol{x} \in \mathbb{R}^N$ we mean the vector $\boldsymbol{t} * \boldsymbol{x} \in \mathbb{R}^{M+N-1}$ defined by

$$(\boldsymbol{t} * \boldsymbol{x})_n = \sum_k t_k x_{n-k}, \tag{3.6}$$

where we only sum over k so that $0 \leq k < M$, $0 \leq n - k < N$.

Note that convolution in the literature usually assumes infinite vectors. Python has the built-in function `convolve` for computing $\boldsymbol{t} * \boldsymbol{x}$. As we shall see in the exercises this function is highly optimized, and is therefore much used in practice. Since convolution is not exactly the same as our definition of a filter (since we assume that a vector is repeated periodically), it would be a good idea to express our definition of filters in terms of convolution. This can be achieved with the next proposition, which is formulated for the case with equally many filter coefficients with negative and positive indices. The result is thus directly applicable for symmetric filters, which is the type of filters we will mostly concentrate on. It is a simple exercise to generalize the result to other filters, however.

Proposition 3.8. *Using convolution to compute filters.*

Assume that S is a filter on the form

$$S = \{t_{-L}, \ldots, \underline{t_0}, \ldots, t_L\}.$$

If $\boldsymbol{x} \in \mathbb{R}^N$, then $S\boldsymbol{x}$ can be computed as follows:

- Form the vector $\tilde{\boldsymbol{x}} = (x_{N-L}, \cdots, x_{N-1}, x_0, \cdots, x_{N-1}, x_0, \cdots, x_{L-1}) \in \mathbb{R}^{N+2L}$.
- Use the `convolve` function to compute $\tilde{\boldsymbol{z}} = \boldsymbol{t} * \tilde{\boldsymbol{x}} \in \mathbb{R}^{M+N+2L-1}$.
- We have that $S\boldsymbol{x} = (\tilde{z}_{2L}, \ldots, \tilde{z}_{M+N-2})$.

We will consider an implementation of this result using the `convolve` function in the exercises.

Proof. When $\boldsymbol{x} \in \mathbb{R}^N$, the operation $\boldsymbol{x} \to \boldsymbol{t} * \boldsymbol{x}$ can be represented by an $(M + N - 1) \times N$ matrix. It is easy to see that this matrix has element $(i + s, i)$ equal to t_s, for $0 \leq i < M$, $0 \leq s < N$. In Figure 3.1 such a matrix is shown for $M = 5$. The nonzero diagonals are shown as diagonal lines.

Now, form the vector $\tilde{\boldsymbol{x}} \in \mathbb{R}^{N+2L}$ as in the text of the theorem. Convolving $(t_{-L}, \ldots, t_L)$ with vectors in $\mathbb{R}^{N+2L}$ can similarly be represented by an $(M+N+2L-1) \times (N+2L)$-matrix. The rows from $2L$ up to and including $M + N - 2$ in this matrix (we have marked these with horizontal lines above) make up a new matrix $\tilde{S}$, and this is shown in Figure 3.2 ($\tilde{S}$ is an $N \times (N + 2L)$ matrix).

We need to show that $S\boldsymbol{x} = \tilde{S}\tilde{\boldsymbol{x}}$. We have that $\tilde{S}\tilde{\boldsymbol{x}}$ equals the matrix shown in Figure 3.3 multiplied with $(x_{N-L}, \ldots, x_{N-1}, x_0, \ldots, x_{N-1}, x_0, \ldots, x_{L-1})$ (we

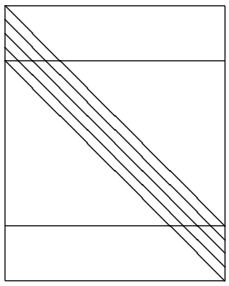

Figure 3.1: A $(M+N-1)\times N$ matrix representing the operation $\boldsymbol{x}\to\boldsymbol{t}*\boldsymbol{x}$.

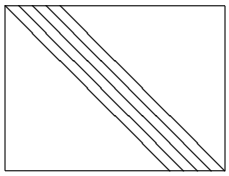

Figure 3.2: The $N\times(N+2L)$-matrix $\tilde{S}$.

inserted extra vertical lines in the matrix where circulation occurs), which equals the matrix shown in Figure 3.4 multiplied with $(x_0,\ldots,x_{N-1})$. We see that this is $S\boldsymbol{x}$, and the proof is complete.

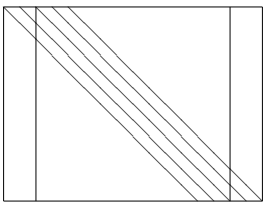

Figure 3.3: The matrix we multiply with $(x_{N-L},\ldots,x_{N-1},x_0,\ldots,x_{N-1},x_0,\ldots,x_{L-1})$.

□

There is also a very nice connection between convolution and polynomials:

Proposition 3.9. *Convolution and polynomials.*

Assume that $p(x)=a_Nx^N+a_{N-1}x_{N-1}+\ldots,a_1x+a_0$ and $q(x)=b_Mx^M+b_{M-1}x_{M-1}+\ldots,b_1x+b_0$ are polynomials of degree N and M respectively.

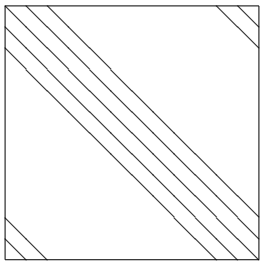

Figure 3.4: The matrix we multiply with $(x_0, \ldots, x_{N-1})$.

Then the coefficients of the polynomial pq can be obtained by computing `convolve(a,b)`.

We can thus interpret a filter as a polynomial. In this setting, clearly the length $l(S)$ of the filter can be interpreted as the degree of the polynomial. If $\boldsymbol{t} \in \mathbb{R}^M$ and $\boldsymbol{x} \in \mathbb{R}^N$, then they can be associated with polynomials of degree $M-1$ and $N-1$, respectively. Also, their convolution, which is in $\mathbb{R}^{M+N-1}$, can be associated with a polynomial of degree $M+N-2$, which is the sum of the degrees of the individual polynomials. Of course we can make the same addition of degrees when we multiply polynomials. Clearly the polynomial associated with $\boldsymbol{t}$ is the frequency response, when we insert $x = e^{-i\omega}$. Also, applying two filters in succession is equivalent to applying the convolution of the filters, so that two filtering operations can be combined to one.

Since the number of nonzero filter coefficients is typically much less than N (the period of the input vector), the matrix S have many entries which are zero. Multiplication with such matrices requires less additions and multiplications than for other matrices: If S has k nonzero filter coefficients, S has Nk nonzero entries, so that kN multiplications and $(k-1)N$ additions are needed to compute $S\boldsymbol{x}$. This is much less than the N^2 multiplications and $(N-1)N$ additions needed in the general case. Perhaps more important is that we need not form the entire matrix, we can perform the matrix multiplication directly in a loop. For large N we risk running into out of memory situations if we had to form the entire matrix.

What you should have learned in this section.

- How to write down the circulant Toeplitz matrix from a digital filter expression, and vice versa.
- How to find the first column of this matrix ($\boldsymbol{s}$) from the filter coefficients ($\boldsymbol{t}$), and vice versa.
- The compact filter notation for filters with a finite number of filter coefficients.

- The definition of convolution, its connection with filters, and the `conv`-function for computing convolution.
- Connection between applying a filter and multiplying polynomials.

Exercise 3.1: Finding the filter coefficients and the matrix

Assume that the filter S is defined by the formula

$$z_n = \frac{1}{4}x_{n+1} + \frac{1}{4}x_n + \frac{1}{4}x_{n-1} + \frac{1}{4}x_{n-2}.$$

Write down the filter coefficients t_k, and the matrix for S when $N = 8$.

Exercise 3.2: Finding the filter coefficients from the matrix

Given the circulant Toeplitz matrix

$$S = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 2 \\ 2 & 0 & 0 & 1 \end{pmatrix},$$

write down the filter coefficients t_k.

Exercise 3.3: Convolution and polynomials

Compute the convolution of $\{\underline{1}, 2, 1\}$ with itself. interpret the result in terms of two polynomials.

Exercise 3.4: Implementation of convolution

Implement code which computes $\boldsymbol{t} * \boldsymbol{x}$ in the two ways described after Equation (3.3) in the compendium, i.e. as a double for loop, and as a simple for loop in `k`, with `n` vectorized. As your $\boldsymbol{t}$, take k randomly generated numbers. Compare execution times for these two methods and the `convolve` function, for different values of k. Present the result as a plot where k runs along the x-axis, and execution times run along the y-axis. Your result will depend on how Python performs vectorization.

Exercise 3.5: Filters with a different number of coefficients with positive and negative indices

Assume that $S = \{t_{-E}, \ldots, \underline{t_0}, \ldots, t_F\}$. Formulate a generalization of Proposition 3.8 for such filters, i.e. to filters where there may be a different number of filter coefficients with positive and negative indices. You should only need to make some small changes to the proof of Proposition 3.8 to achieve this.

Exercise 3.6: Implementing filtering with convolution

Implement a function `filterS` which uses Proposition 3.8 and the `convolve` function $S\boldsymbol{x}$ when $S = \{t_{-L}, \ldots, \underline{t_0}, \ldots, t_L\}$ The function should take the vectors $(t_{-L}, \ldots, t_0, \ldots, t_L)$ and $\boldsymbol{x}$ as input.

3.2 Formal definition of filters and the vector frequency response

Let us now define digital filters formally, and establish their relationship to Toeplitz matrices. We have seen that a sound can be decomposed into different frequency components, and we would like to define filters as operations which adjust these frequency components in a predictable way. One such example is provided in Example 2.27, where we simply set some of the frequency components to 0. The natural starting point is to require for a filter that the output of a pure tone is a pure tone with the same frequency.

Definition 3.10. *Digital filters and vector frequency response.*

A linear transformation $S : \mathbb{R}^N \mapsto \mathbb{R}^N$ is a said to be a digital filter, or simply a filter, if, for any integer n in the range $0 \leq n \leq N-1$ there exists a value $\lambda_{S,n}$ so that

$$S(\phi_n) = \lambda_{S,n}\phi_n, \tag{3.7}$$

i.e., the N Fourier vectors are the eigenvectors of S. The vector of (eigen)values $\boldsymbol{\lambda}_S = (\lambda_{S,n})_{n=0}^{N-1}$ is often referred to as the *(vector) frequency response* of S.

Since the Fourier basis vectors are orthogonal vectors, S is clearly orthogonally diagonalizable. Since also the Fourier basis vectors are the columns in $(F_N)^H$, we have that

$$S = F_N^H D F_N \tag{3.8}$$

whenever S is a digital filter, where D has the frequency response (i.e. the eigenvalues) on the diagonal [1]. We could also use DFT_N to diagonalize filters, but it is customary to use an orthogonal matrix (i.e. F_N) when the matrix is orthogonally diagonalizable. In particular, if S_1 and S_2 are digital filters, we can write $S_1 = F_N^H D_1 F_N$ and $S_2 = F_N^H D_2 F_N$, so that

$$S_1 S_2 = F_N^H D_1 F_N F_N^H D_2 F_N = F_N^H D_1 D_2 F_N.$$

Since $D_1 D_2 = D_2 D_1$ for any diagonal matrices, we get the following corollary:

Corollary 3.11. *The product of two filters is a filter.*

The product of two digital filters is again a digital filter. Moreover, all digital filters commute, i.e. if S_1 and S_2 are digital filters, $S_1 S_2 = S_2 S_1$.

[1] Recall that the orthogonal diagonalization of S takes the form $S = PDP^T$, where P contains as columns an orthonormal set of eigenvectors, and D is diagonal with the eigenvectors listed on the diagonal (see Section 7.1 in [20]).

Clearly also $S_1 + S_2$ is a filter when S_1 and S_2 are. The set of all filters is thus a vector space, which also is closed under multiplication. Such a space is called an *algebra*. Since all filters commute, this algebra is also called a *commutative algebra*.

The next result states three equivalent characterizations of a digital filter. The first one is simply the definition in terms of having the Fourier basis as eigenvectors. The second is that the matrix is circulant Toeplitz, i.e. that the operations we started this chapter with actually are filters. The third characterization is in terms of a new concept which we now define.

Definition 3.12. *Time-invariance.*

Assume that S is a linear transformation from $\mathbb{R}^N$ to $\mathbb{R}^N$. Let $\boldsymbol{x}$ be input to S, and $\boldsymbol{y} = S\boldsymbol{x}$ the corresponding output. Let also $\boldsymbol{z}$, $\boldsymbol{w}$ be delays of $\boldsymbol{x}$, $\boldsymbol{y}$ with d elements (i.e. $z_n = x_{n-d}$, $w_n = y_{n-d}$). S is said to be *time-invariant* if, for any d and $\boldsymbol{x}$, $S\boldsymbol{z} = \boldsymbol{w}$ (i.e. S sends the delayed input vector to the delayed output vector).

With this notation, it is clear that time-delay with d elements, i.e. the operation $\boldsymbol{x} \to \boldsymbol{z}$, is a filter, since the time-delay of $\boldsymbol{x} = \phi_n = \frac{1}{\sqrt{N}} e^{2\pi i k n/N}$ is $\frac{1}{\sqrt{N}} e^{2\pi i (k-d) n/N} = e^{-2\pi i d n/N}\boldsymbol{x}$, and the Fourier basis are thus eigenvectors. If we denote the *time-delay filter* with E_d, the definition of time-invariance demands that $SE_d\boldsymbol{x} = E_d S\boldsymbol{x}$ for any $\boldsymbol{x}$ and d, i.e. $SE_d = E_d S$ for any d. We can now prove the following.

Theorem 3.13. *Characterizations of digital filters.*

The following are equivalent characterizations of a digital filter:

- $S = (F_N)^H D F_N$ for a diagonal matrix D, i.e. the Fourier basis is a basis of eigenvectors for S.
- S is a circulant Toeplitz matrix.
- S is linear and time-invariant.

Proof. If S is a filter, then $SE_d = E_d S$ for all d since all filters commute, so that S is time-invariant. This proves 1. $\to$ 3..

Assume that S is time-invariant. Note that $E_d\boldsymbol{e}_0 = \boldsymbol{e}_d$, and since $SE_d\boldsymbol{e}_0 = E_d S\boldsymbol{e}_0$ we have that $S\boldsymbol{e}_d = E_d\boldsymbol{s}$, where $\boldsymbol{s}$ is the first column of S. This also says that column d of S can be obtained by delaying the first column of S with d elements. But then d is a circulant Toeplitz matrix. This proves 3. $\to$ 2..

Finally, any circulant Toeplitz matrix can be written on the form $\sum_{d=0}^{N-1} s_d E_d$ (by splitting the matrix into a sum of its diagonals). Since all E_d are filters, it is clear that any circulant Toeplitz matrix is a filter. This proves 2. $\to$ 1.. □

Due to this result, filters are also called *LTI filters*, LTI standing for Linear, Time-Invariant. Also, operations defined by (3.3) are digital filters, when restricted to vectors with period N. The following results enables us to compute the eigenvalues/frequency response easily, so that we do not need to form the characteristic polynomial and find its roots:

Theorem 3.14. *Connection between frequency response and the matrix.*

Any digital filter is uniquely characterized by the values in the first column of its matrix. Moreover, if $\boldsymbol{s}$ is the first column in S, the frequency response of S is given by

$$\boldsymbol{\lambda}_S = \mathrm{DFT}_N \boldsymbol{s}. \tag{3.9}$$

Conversely, if we know the frequency response $\boldsymbol{\lambda}_S$, the first column $\boldsymbol{s}$ of S is given by

$$\boldsymbol{s} = \mathrm{IDFT}_N \boldsymbol{\lambda}_S. \tag{3.10}$$

Proof. If we replace S by $(F_N)^H D F_N$ we find that

$$\begin{aligned}\mathrm{DFT}_N \boldsymbol{s} = \sqrt{N} F_N \boldsymbol{s} = \sqrt{N} F_N S \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \sqrt{N} F_N F_N^H D F_N \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \\ = \sqrt{N} D F_N \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = D \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \boldsymbol{\lambda}_S,\end{aligned}$$

where we have used that the first column in F_N has all entries equal to $1/\sqrt{N}$, and that the diagonal matrix D has all the eigenvalues of S on its diagonal, so that the last expression is the vector of eigenvalues $\boldsymbol{\lambda}_S$. This proves (3.9). Equation (3.10) follows directly by applying the inverse DFT to (3.9). □

The first column $\boldsymbol{s}$, which thus characterizes the filter, is also called the *impulse response*. This name stems from the fact that we can write $\boldsymbol{s} = S\boldsymbol{e}_0$, i.e. the vector $\boldsymbol{s}$ is the output (often called response) to the vector $\boldsymbol{e}_0$ (often called an impulse). Equation (3.9) states that the frequency response can be written as

$$\lambda_{S,n} = \sum_{k=0}^{N-1} s_k e^{-2\pi i n k/N}, \quad \text{for } n = 0,\, 1,\, \ldots,\, N-1, \tag{3.11}$$

where s_k are the components of $\boldsymbol{s}$.

Example 3.15. *The identity is a filter.*

The identity matrix is a digital filter since $I = (F_N)^H I F_N$. Since $\boldsymbol{e}_0$ is the first column, it has impulse response $\boldsymbol{s} = \boldsymbol{e}_0$. Its frequency response has 1 in all components and therefore preserves all frequencies, as expected.

Example 3.16. *Frequency response of a simple filter.*

When only few of the coefficients s_k are nonzero, it is possible to obtain nice expressions for the frequency response. To see this, let us compute the frequency response of the filter defined from formula (3.1). We saw that the first column of the corresponding Toeplitz matrix satisfied $s_0 = 1/2$, and $s_{N-1} = s_1 = 1/4$. The frequency response is thus

$$\begin{aligned}\lambda_{S,n} &= \frac{1}{2}e^0 + \frac{1}{4}e^{-2\pi in/N} + \frac{1}{4}e^{-2\pi in(N-1)/N} \\ &= \frac{1}{2}e^0 + \frac{1}{4}e^{-2\pi in/N} + \frac{1}{4}e^{2\pi in/N} = \frac{1}{2} + \frac{1}{2}\cos(2\pi n/N).\end{aligned}$$

Equations (3.8), (3.9), and (3.10) are important relations between the matrix- and frequency representations of a filter. We see that the DFT is a crucial ingredient in these relations. A consequence is that, once you recognize a matrix as circulant Toeplitz, you do not need to make the tedious calculation of eigenvectors and eigenvalues which you are used to. Let us illustrate this with an example.

Example 3.17. *Matrix form.*

Let us compute the eigenvalues and eigenvectors of the simple matrix

$$S = \begin{pmatrix} 4 & 1 \\ 1 & 4 \end{pmatrix}.$$

It is straightforward to compute the eigenvalues and eigenvectors of this matrix the way you learnt in your first course in linear algebra. However, this matrix is also a circulant Toeplitz matrix, so that we can use the results in this section to compute the eigenvalues and eigenvectors. Since here $N = 2$, we have that $e^{2\pi ink/N} = e^{\pi ink} = (-1)^{nk}$. This means that the Fourier basis vectors are $(1, 1)/\sqrt{2}$ and $(1, -1)/\sqrt{2}$, which also are the eigenvectors of S. The eigenvalues are the frequency response of S, which can be obtained as

$$\sqrt{N}F_N \boldsymbol{s} = \sqrt{2}\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 3 \end{pmatrix}$$

The eigenvalues are thus 3 and 5. You could have obtained the same result with your computer. Note that the computer may not return the eigenvectors exactly as the Fourier basis vectors, since the eigenvectors are not unique (the multiple of an eigenvector is also an eigenvector). The computer may for instance switch the signs of the eigenvectors. We have no control over what the computer actually chooses to do, since it uses some underlying numerical algorithm for computing eigenvectors which we can't influence.

In signal processing, the frequency content of a vector (i.e., its DFT) is also referred to as its spectrum. This may be somewhat confusing from a linear algebra perspective, because in this context the term spectrum is used to denote the eigenvalues of a matrix. But because of Theorem 3.14 this is not so confusing

after all if we interpret the spectrum of a vector (in signal processing terms) as the spectrum of the corresponding digital filter (in linear algebra terms).

Certain vectors are easy to express in terms of the Fourier basis. This enables us to compute the output of such vectors from a digital filter easily, as the following example shows.

Example 3.18. *Computing the output of a filter.*

Let us consider the filter S defined by $z_n = \frac{1}{6}(x_{n+2} + 4x_{n+1} + 6x_n + 4x_{n-1} + x_{n-2})$, and see how we can compute $S\boldsymbol{x}$ when

$$\boldsymbol{x} = (\cos(2\pi 5 \cdot 0/N), \cos(2\pi 5 \cdot 1/N), \ldots, \cos(2\pi 5 \cdot (N-1)/N)),$$

where N is the length of the vector. We note first that

$$\begin{aligned}\sqrt{N}\phi_5 &= \left(e^{2\pi i 5\cdot 0/N}, e^{2\pi i 5\cdot 1/N}, \ldots, e^{2\pi i 5\cdot (N-1)/N}\right)\\ \sqrt{N}\phi_{N-5} &= \left(e^{-2\pi i 5\cdot 0/N}, e^{-2\pi i 5\cdot 1/N}, \ldots, e^{-2\pi i 5\cdot (N-1)/N}\right),\end{aligned}$$

Since $e^{2\pi i 5k/N} + e^{-2\pi i 5k/N} = 2\cos(2\pi 5k/N)$, we get by adding the two vectors that $\boldsymbol{x} = \frac{1}{2}\sqrt{N}(\phi_5 + \phi_{N-5})$. Since the ϕ_n are eigenvectors, we have expressed $\boldsymbol{x}$ as a sum of eigenvectors. The corresponding eigenvalues are given by the vector frequency response, so let us compute this. If $N = 8$, computing $S\boldsymbol{x}$ means to multiply with the 8×8 circulant Toeplitz matrix

$$\frac{1}{6}\begin{pmatrix} 6 & 4 & 1 & 0 & 0 & 0 & 1 & 4\\ 4 & 6 & 4 & 1 & 0 & 0 & 0 & 1\\ 1 & 4 & 6 & 4 & 1 & 0 & 0 & 0\\ 0 & 1 & 4 & 6 & 4 & 1 & 0 & 0\\ 0 & 0 & 1 & 4 & 6 & 4 & 1 & 0\\ 0 & 0 & 0 & 1 & 4 & 6 & 4 & 1\\ 1 & 0 & 0 & 0 & 1 & 4 & 6 & 4\\ 4 & 1 & 0 & 0 & 0 & 1 & 4 & 6 \end{pmatrix}$$

We now see that

$$\begin{aligned}\lambda_{S,n} &= \frac{1}{6}(6 + 4e^{-2\pi i n/N} + e^{-2\pi i 2n/N} + e^{-2\pi i (N-2)n/N} + 4e^{-2\pi i (N-1)n/N})\\ &= \frac{1}{6}(6 + 4e^{2\pi i n/N} + 4e^{-2\pi i n/N} + e^{2\pi i 2n/N} + e^{-2\pi i 2n/N})\\ &= 1 + \frac{4}{3}\cos(2\pi n/N) + \frac{1}{3}\cos(4\pi n/N).\end{aligned}$$

The two values of this we need are

$$\begin{aligned}\lambda_{S,5} &= 1 + \frac{4}{3}\cos(2\pi 5/N) + \frac{1}{3}\cos(4\pi 5/N)\\ \lambda_{S,N-5} &= 1 + \frac{4}{3}\cos(2\pi(N-5)/N) + \frac{1}{3}\cos(4\pi(N-5)/N)\\ &= 1 + \frac{4}{3}\cos(2\pi 5/N) + \frac{1}{3}\cos(4\pi 5/N).\end{aligned}$$

Since these are equal, $\boldsymbol{x}$ is a sum of eigenvectors with equal eigenvalues. This means that $\boldsymbol{x}$ itself also is an eigenvector, with the same eigenvalue, so that

$$S\boldsymbol{x} = \left(1 + \frac{4}{3}\cos(2\pi 5/N) + \frac{1}{3}\cos(4\pi 5/N)\right)\boldsymbol{x}.$$

3.2.1 Using digital filters to approximate analog filters

The formal definition of digital filters resembles that of analog filters, the difference being that the Fourier basis is now discrete. From this one may think that one can construct digital filters from analog filters. The following result clarifies this:

Theorem 3.19. *Connection with analog frequency response.*

Let s be an analog filter with frequency response $\lambda_s(f)$, and assume that $f \in V_{M,T}$ (so that also $s(f) \in V_{M,T}$). Let

$$\begin{aligned}\boldsymbol{x} &= (f(0\cdot T/N), f(1\cdot T/N), \ldots, f((N-1)T/N))\\ \boldsymbol{z} &= (s(f)(0\cdot T/N), s(f)(1\cdot T/N), \ldots, s(f)((N-1)T/N))\end{aligned}$$

be vectors of $N = 2M+1$ uniform samples from f and $s(f)$. Then the operation $S : \boldsymbol{x} \to \boldsymbol{z}$ (i.e. the operation which sends the samples of the input to the samples of the output) is well-defined on $\mathbb{R}^N$, and is an $N \times N$-digital filter with frequency response $\lambda_{S,n} = \lambda_s(n/T)$.

Proof. With $N = 2M+1$ we know that $f \in V_{M,T}$ is uniquely determined from $\boldsymbol{x}$. This means that $s(f)$ also is uniquely determined from $\boldsymbol{x}$, so that $\boldsymbol{z}$ also is uniquely determined from $\boldsymbol{x}$. The operation $S : \boldsymbol{x} \to \boldsymbol{z}$ is therefore well-defined on $\mathbb{R}^N$.

Clearly also $s(e^{2\pi int/T}) = \lambda_s(n/T)e^{2\pi int/T}$. Since the samples of $e^{2\pi int/T}$ is the vector $e^{2\pi ikn/N}$, and the samples of $\lambda_s(n/T)e^{2\pi int/T}$ is $\lambda_s(n/T)e^{2\pi ikn/N}$, the vector $e^{2\pi ikn/N}$ is an eigenvector of S with eigenvalue $\lambda_s(n/T)$. Clearly then S is a digital filter with frequency response $\lambda_{S,n} = \lambda_s(n/T)$. □

It is interesting that the digital frequency response above is obtained by sampling the analog frequency response. In this way we also see that it is easy to realize any digital filter as the restriction of an analog filter: any analog filter s will do where the frequency response has the values $\lambda_{S,n}$ at the points n/T.

In the theorem it is essential that $f \in V_{M,T}$. There are many functions with the same samples, but where the samples of the output from the analog filter are different. When we restrict to $V_{M,T}$, however, the output samples are always determined from the input samples.

Theorem 3.19 explains how digital filters can occur in practice. In the real world, a signal is modeled as a continuous function $f(t)$, and an operation on signals as an analog filter s. We can't compute the entire output $s(f)$ of the analog filter, but it is possible to apply the digital filter from Theorem 3.19 to the samples $\boldsymbol{x}$ of f. In general $f(t)$ may not lie in $V_{M,T}$, but we can denote by $\tilde{f}$ the unique function in $V_{M,T}$ with the same samples as f (as in Section 2.3). By definition, $S\boldsymbol{x}$ are the samples of $s(\tilde{f}) \in V_{M,T}$. $s(\tilde{f})$ can finally be found from these samples by using the procedure from Figure 2.4 for finding $s(\tilde{f})$. This procedure for finding $s(\tilde{f})$ is illustrated in Figure 3.5.

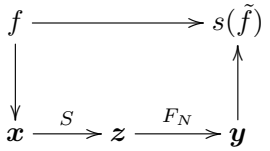

Figure 3.5: The connections between analog and digital filters, sampling and interpolation, provided by Theorem 3.19. The left vertical arrow represents sampling, the right vertical arrow represents interpolation.

Clearly, $s(\tilde{f})$ is an approximation to $s(f)$, since $\tilde{f}$ is an approximation to f, and since s is continuous. Let us summarize this as follows:

Idea 3.20. *Approximating an analog filter.*

An analog filter s can be approximated through sampling, a digital filter, the DFT, and interpolation, as illustrated in Figure 3.5. S is the digital filter with frequency response $\lambda_{S,n} = \lambda_s(n/T)$. When $f \in V_{M,T}$, this approximation equals $s(f)$. When we increase the number of sample points/the size of the filter, the approximation becomes better. If there is a bound on the highest frequency in f, there exists an N so that when sampling of that size, the approximation equals $s(f)$.

Let us comment on why the last statements here are true. That the approximation equals $s(f)$ when $f \in V_{M,T}$ is obvious, since both f and $s(f) \in V_{M,T}$ are determined from their samples then. If there is a bound on the highest frequency in f, then f lies in $V_{M,T}$ for large enough M, so that we recover $s(f)$ as our approximation using $N = 2M + 1$. Finally, what happens when there is no bound on the highest frequency? We know that $s(f_N) = (s(f))_N$. Since f_N is a good approximation to f, the samples $\boldsymbol{x}$ of f are close to the samples of f_N. By continuity of the digital filter, $\boldsymbol{z} = S\boldsymbol{x}$ will also be close to the samples of $(s(f))_N = s(f_N)$, so that (also by continuity) interpolating with $\boldsymbol{z}$ gives a good approximation to $(s(f))_N$, which is again a good approximation to $s(f)$). From this it follows that the digital filter is a better approximation when N is high.

What you should have learned in this section.

- The formal definition of a digital filter in terms of having the Fourier vectors as eigenvectors.
- The definition of the vector frequency response in terms of the corresponding eigenvalues.
- The definition of time-invariance and the three equivalent characterizations of a filter.
- For filters, eigenvalues can be computed by taking the DFT of the first column $\boldsymbol{s}$, and there is no need to compute eigenvectors explicitly.
- How to apply a digital filter to a sum of sines or cosines, by splitting these into a sum of eigenvectors.

Exercise 3.7: Time reversal is not a filter

In Example 2.6 we looked at time reversal as an operation on digital sound. In $\mathbb{R}^N$ this can be defined as the linear mapping which sends the vector $\boldsymbol{e}_k$ to $\boldsymbol{e}_{N-1-k}$ for all $0 \leq k \leq N-1$.

a) Write down the matrix for the time reversal linear mapping, and explain from this why time reversal is not a digital filter.

b) Prove directly that time reversal is not a time-invariant operation.

Exercise 3.8: When is a filter symmetric?

Let S be a digital filter. Show that S is symmetric if and only if the frequency response satisfies $\lambda_{S,n} = \lambda_{S,N-n}$ for all n.

Exercise 3.9: Eigenvectors and eigenvalues

Consider the matrix

$$S = \begin{pmatrix} 4 & 1 & 3 & 1 \\ 1 & 4 & 1 & 3 \\ 3 & 1 & 4 & 1 \\ 1 & 3 & 1 & 4 \end{pmatrix}.$$

a) Compute the eigenvalues and eigenvectors of S using the results of this section. You should only need to perform one DFT in order to achieve this.

b) Verify the result from a) by computing the eigenvectors and eigenvalues the way you taught in your first course in linear algebra. This should be a much more tedious task.

c) Use a computer to compute the eigenvectors and eigenvalues of S also. For some reason some of the eigenvectors seem to be different from the Fourier basis vectors, which you would expect from the theory in this section. Try to find an explanation for this.

Exercise 3.10: Composing filters

Assume that S_1 and S_2 are two circulant Toeplitz matrices.

a) How can you express the eigenvalues of $S_1 + S_2$ in terms of the eigenvalues of S_1 and S_2?

b) How can you express the eigenvalues of $S_1 S_2$ in terms of the eigenvalues of S_1 and S_2?

c) If A and B are general matrices, can you find a formula which expresses the eigenvalues of $A + B$ and AB in terms of those of A and B? If not, can you find a counterexample to what you found in a) and b)?

Exercise 3.11: Keeping every second component

Consider the linear mapping S which keeps every second component in $\mathbb{R}^N$, i.e. $S(\boldsymbol{e}_{2k}) = \boldsymbol{e}_{2k}$, and $S(\boldsymbol{e}_{2k-1}) = \mathbf{0}$. Is S a digital filter?

3.3 The continuous frequency response and properties

If we make the substitution $\omega = 2\pi n/N$ in the formula for $\lambda_{S,n}$, we may interpret the frequency response as the values on a continuous function on $[0, 2\pi)$.

Theorem 3.21. *Connection between vector- and continuous frequency response.*
The function $\lambda_S(\omega)$ defined on $[0, 2\pi)$ by

$$\lambda_S(\omega) = \sum_k t_k e^{-ik\omega}, \tag{3.12}$$

where t_k are the filter coefficients of S, satisfies

$$\lambda_{S,n} = \lambda_S(2\pi n/N) \text{ for } n = 0,\, 1,\, \ldots,\, N-1$$

for any N. In other words, regardless of N, the vector frequency response lies on the curve λ_S.

Proof. For any N we have that

$$\begin{aligned}
\lambda_{S,n} &= \sum_{k=0}^{N-1} s_k e^{-2\pi ink/N} = \sum_{0\le k<N/2} s_k e^{-2\pi ink/N} + \sum_{N/2\le k\le N-1} s_k e^{-2\pi ink/N} \\
&= \sum_{0\le k<N/2} t_k e^{-2\pi ink/N} + \sum_{N/2\le k\le N-1} t_{k-N} e^{-2\pi ink/N} \\
&= \sum_{0\le k<N/2} t_k e^{-2\pi ink/N} + \sum_{-N/2\le k\le -1} t_k e^{-2\pi in(k+N)/N} \\
&= \sum_{0\le k<N/2} t_k e^{-2\pi ink/N} + \sum_{-N/2\le k\le -1} t_k e^{-2\pi ink/N} \\
&= \sum_{-N/2\le k<N/2} t_k e^{-2\pi ink/N} = \lambda_S(2\pi n/N).
\end{aligned}$$

where we have used Equation (3.4). □

Both $\lambda_S(\omega)$ and $\lambda_{S,n}$ will be referred to as frequency responses in the following. To distinguish the two, while $\lambda_{S,n}$ is called the vector frequency response of S, $\lambda_S(\omega)$) is called the *continuous frequency response* of S. ω is called *angular frequency*.

The difference in the definition of the continuous- and the vector frequency response lies in that one uses the filter coefficients t_k, while the other uses the impulse response s_k. While these contain the same values, they are ordered differently. Had we used the impulse response to define the continuous frequency response, we would have needed to compute $\sum_{k=0}^{N-1} s_k e^{-\pi i\omega}$, which does not converge when $N \to \infty$ (although it gives the right values at all points $\omega = 2\pi n/N$ for all N)! The filter coefficients avoid this convergence problem, however, since we assume that only t_k with $|k|$ small are nonzero. In other words, filter coefficients are used in the definition of the continuous frequency response so that we can find a continuous curve where we can find the vector frequency response values for all N.

The frequency response contains the important characteristics of a filter, since it says how it behaves for the different frequencies. When analyzing a filter, we therefore often plot the frequency response. Often we plot only the absolute value (or the magnitude) of the frequency response, since this is what explains how each frequency is amplified or attenuated. Since λ_S is clearly periodic with period 2π, we may restrict angular frequency to the interval $[0, 2\pi)$. The conclusion in Observation 2.22 was that the low frequencies in a vector correspond to DFT indices close to 0 and $N-1$, and high frequencies correspond to DFT indices close to $N/2$. This observation is easily translated to a statement about angular frequencies:

Observation 3.22. *Plotting the frequency response.*

When plotting the frequency response on $[0, 2\pi)$, angular frequencies near 0 and 2π correspond to low frequencies, angular frequencies near π correspond to high frequencies

λ_S may also be viewed as a function defined on the interval $[-\pi, \pi)$. Plotting on $[-\pi, \pi]$ is often done in practice, since it makes clearer what corresponds to lower frequencies, and what corresponds to higher frequencies:

Observation 3.23. *Higher and lower frequencies.*
When plotting the frequency response on $[-\pi, \pi)$, angular frequencies near 0 correspond to low frequencies, angular frequencies near $\pm\pi$ correspond to high frequencies.

The following holds:

Theorem 3.24. *Connection between analog and digital filters.*
Assume that s is an analog filter, and that we sample a periodic function at rate f_s over one period, and denote the corresponding digital filter by S. The analog and digital frequency responses are related by $\lambda_s(f) = \lambda_S(2\pi f f_s)$.

To see this, note first that S has frequency response $\lambda_{S,n} = \lambda_s(n/T) = \lambda_s(f)$, where $f = n/T$. We then rewrite $\lambda_{S,n} = \lambda_S(2\pi n/N) = \lambda_S(2\pi f T/N) = \lambda_S(2\pi f f_s)$.

Example 3.25. *Plotting a simple frequency response.*
In Example 3.16 we computed the vector frequency response of the filter defined in formula (3.1). The filter coefficients are here $t_{-1} = 1/4$, $t_0 = 1/2$, and $t_1 = 1/4$. The continuous frequency response is thus

$$\lambda_S(\omega) = \frac{1}{4}e^{i\omega} + \frac{1}{2} + \frac{1}{4}e^{-i\omega} = \frac{1}{2} + \frac{1}{2}\cos\omega.$$

Clearly this matches the computation from Example 3.16. Figure 3.6 shows plots of this frequency response, plotted on the intervals $[0, 2\pi)$ and $[-\pi, \pi)$.

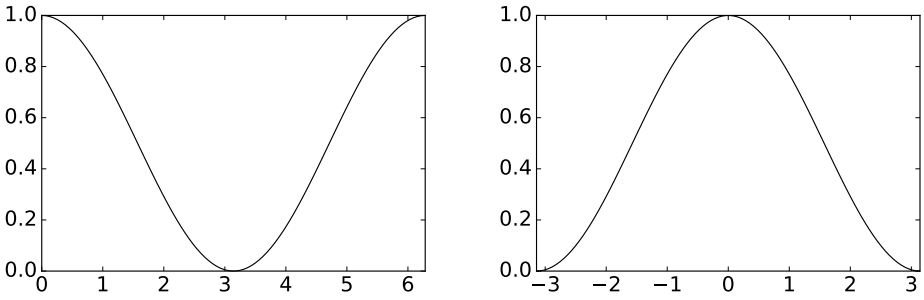

Figure 3.6: The (absolute value of the) frequency response of the moving average filter of Formula (3.1) from the beginning of this chapter, plotted over $[0, 2\pi]$ and $[-\pi, \pi]$.

Both the continuous frequency response and the vector frequency response for $N = 51$ are shown. Figure (b) shows clearly how the high frequencies are softened by the filter.

Since the frequency response is essentially a DFT, it inherits several properties from Theorem 2.18. We will mostly use the continuous frequency response to express these properties.

Theorem 3.26. *Properties of the frequency response.*
We have that

- The continuous frequency response satisfies $\lambda_S(-\omega) = \overline{\lambda_S(\omega)}$.
- If S is a digital filter, S^T is also a digital filter. Moreover, if the frequency response of S is $\lambda_S(\omega)$, then the frequency response of S^T is $\overline{\lambda_S(\omega)}$.
- If S is symmetric, λ_S is real. Also, if S is antisymmetric (the element on the opposite side of the diagonal is the same, but with opposite sign), λ_S is purely imaginary.
- A digital filter S is an invertible if and only if $\lambda_{S,n} \neq 0$ for all n. In that case S^{-1} is also a digital filter, and $\lambda_{S^{-1},n} = 1/\lambda_{S,n}$.
- If S_1 and S_2 are digital filters, then S_1S_2 also is a digital filter, and $\lambda_{S_1S_2}(\omega) = \lambda_{S_1}(\omega)\lambda_{S_2}(\omega)$.

Proof. Property 1. and 3. follow directly from Theorem 2.18. Transposing a matrix corresponds to reversing the first column of the matrix and thus also the filter coefficients. Due to this Property 2. also follows from Theorem 2.18. If $S = (F_N)^H D F_N$, and all $\lambda_{S,n} \neq 0$, we have that $S^{-1} = (F_N)^H D^{-1} F_N$, where D^{-1} is a diagonal matrix with the values $1/\lambda_{S,n}$ on the diagonal. Clearly then S^{-1} is also a digital filter, and its frequency response is $\lambda_{S^{-1},n} = 1/\lambda_{S,n}$, which proves 4. The last property follows in the same was as we showed that filters commute:

$$S_1S_2 = (F_N)^H D_1 F_N (F_N)^H D_2 F_N = (F_N)^H D_1 D_2 F_N.$$

The frequency response of S_1S_2 is thus obtained by multiplying the frequency responses of S_1 and S_2. □

In particular the frequency response may not be real, although this was the case in the first example of this section. Theorem 3.26 applies also for the vector frequency response. Since the vector frequency response are the eigenvalues of the filter, the last property above says that, for filters, multiplication of matrices corresponds to multiplication of eigenvalues. Clearly this is an important property which is shared with all other matrices which have the same eigenvectors.

Example 3.27. *Computing a composite filter.*
Assume that the filters S_1 and S_2 have the frequency responses $\lambda_{S_1}(\omega) = \cos(2\omega)$, $\lambda_{S_2}(\omega) = 1 + 3\cos\omega$. Let us see how we can use Theorem 3.26 to compute the filter coefficients and the matrix of the filter $S = S_1S_2$. We first

notice that, since both frequency responses are real, all S_1, S_2, and $S = S_1S_2$ are symmetric. We rewrite the frequency responses as

$$\lambda_{S_1}(\omega) = \frac{1}{2}(e^{2i\omega} + e^{-2i\omega}) = \frac{1}{2}e^{2i\omega} + \frac{1}{2}e^{-2i\omega}$$
$$\lambda_{S_2}(\omega) = 1 + \frac{3}{2}(e^{i\omega} + e^{-i\omega}) = \frac{3}{2}e^{i\omega} + 1 + \frac{3}{2}e^{-i\omega}.$$

We now get that

$$\lambda_{S_1S_2}(\omega) = \lambda_{S_1}(\omega)\lambda_{S_2}(\omega) = \left(\frac{1}{2}e^{2i\omega} + \frac{1}{2}e^{-2i\omega}\right)\left(\frac{3}{2}e^{i\omega} + 1 + \frac{3}{2}e^{-i\omega}\right)$$
$$= \frac{3}{4}e^{3i\omega} + \frac{1}{2}e^{2i\omega} + \frac{3}{4}e^{i\omega} + \frac{3}{4}e^{-i\omega} + \frac{1}{2}e^{-2i\omega} + \frac{3}{4}e^{-3i\omega}$$

From this expression we see that the filter coefficients of S are $t_{\pm 1} = 3/4$, $t_{\pm 2} = 1/2$, $t_{\pm 3} = 3/4$. All other filter coefficients are 0. Using Theorem 3.2, we get that $s_1 = 3/4$, $s_2 = 1/2$, and $s_3 = 3/4$, while $s_{N-1} = 3/4$, $s_{N-2} = 1/2$, and $s_{N-3} = 3/4$ (all other s_k are 0). This gives us the matrix representation of S.

3.3.1 Windowing operations

In this section we will take a look at a very important, and perhaps surprising, application of the continuous frequency response. Let us return to the computations from Example 2.27. There we saw that, when we restricted to a block of the signal, this affected the frequency representation. If we substitute with the angular frequencies $\omega = 2\pi n/N$ and $\omega_0 = 2\pi n_0/M$ in Equation (2.12), we get

$$y_n = \frac{1}{N}\sum_{k=0}^{N-1} e^{ik\omega_0}e^{-ik\omega} = \frac{1}{N}\sum_{k=0}^{N-1} e^{-ik(\omega-\omega_0)}$$

(here y_n were the DFT components of the sound after we had restricted to a block). This expression states that, when we restrict to a block of length N in the signal by discarding the other samples, a pure tone of angular frequency ω_0 suddenly gets a frequency contribution at angular frequency ω also, and the contribution is given by this formula. The expression is seen to be the same as the frequency response of the filter $\frac{1}{N}\{\underline{1}, 1, \ldots, 1\}$ (where 1 is repeated N times), evaluated at $\omega - \omega_0$. This filter is nothing but a (delayed) moving average filter. The frequency response of a moving average filter thus governs how the different frequencies pollute when we limit ourselves to a block of the signal. Since this frequency response has its peak at 0, angular frequencies ω close to ω_0 have biggest values, so that the pollution is mostly from frequencies close to ω_0. But unfortunately, other frequencies also pollute.

One can also ask the question if there are better ways to restrict to a block of size N of the signal. We formulate the following idea.

Idea 3.28. *Windows.*

Let $\boldsymbol{x} = (x_0, \ldots, x_M)$ be a sound of length M. We would like to find values $\boldsymbol{w} = \{\underline{w_0}, \ldots, w_{N-1}\}$ so that the new sound $(w_0 x_0, \ldots, w_{N-1} x_{N-1})$ of length $N < M$ has a frequency representation similar to that of $\boldsymbol{x}$. $\boldsymbol{w}$ is called a *window* of length N, and the new sound is called the *windowed signal*.

Above we encountered the window $\boldsymbol{w} = \{\underline{1}, 1, \ldots, 1\}$. This is called the rectangular window. To see how we can find a good window, note first that the DFT values in the windowed signal of length N is

$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} w_k e^{ik\omega_0} e^{-ik\omega} = \frac{1}{N} \sum_{k=0}^{N-1} w_k e^{-ik(\omega-\omega_0)}.$$

This is the frequency response of $\frac{1}{N}\boldsymbol{w}$. In order to limit the pollution from other frequencies, we thus need to construct a window with a frequency response with smaller values than that of the rectangular window away from 0. Let us summarize our findings as follows:

Observation 3.29. *Constructing a window.*

Assume that we would like to construct a window of length N. It is desirable that the frequency response of the window has small values away from zero.

We will not go into techniques for how such frequency responses can be constructed, but only consider one example different from the rectangular window. We define the Hamming window by

$$w_n = 2(0.54 - 0.46\cos(2\pi n/(N-1))). \tag{3.13}$$

The frequency responses of the rectangular window and the Hamming window are compared in Figure 3.7 for $N = 32$.

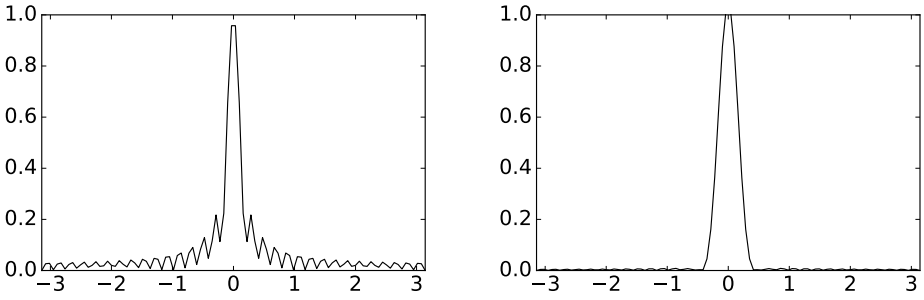

Figure 3.7: The frequency responses of the rectangular and Hamming windows, which we considered for restricting to a block of the signal.

We see that the Hamming window has much smaller values away from 0, so that it is better suited as a window. However, the width of the "main lobe"

(i.e. the main structure at the center), seems to be bigger. The window coefficients themselves are shown in Figure 3.8. It is seen that the frequency response of the Hamming window attenuates more and more as we get close to the boundaries.

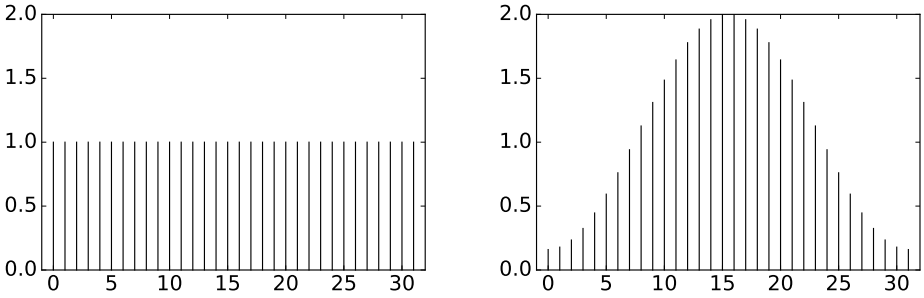

Figure 3.8: The coefficients of the rectangular and Hamming windows, which we considered for restricting to a block of the signal.

Many other windows are used in the literature. The concrete window from Exercise 3.21 is for instance used in the MP3 standard. It is applied to the sound, and after this an FFT is applied to the windowed sound in order to make a frequency analysis of that part of the sound. The effect of the window is that there is smaller loss in the frequency representation of the sound when we restrict to a block of sound samples. This is a very important part of the psychoacoustic model used in the MP3 encoder, since it has to make compression decisions based on the frequency information in the sound.

What you should have learned in this section.

- The definition of the continuous frequency response in terms of the filter coefficients $\boldsymbol{t}$.
- Connection with the vector frequency response.
- Properties of the continuous frequency response, in particular that the product of two frequency responses equals the frequency response of the product.
- How to compute the frequency response of the product of two filters,.
- How to find the filter coefficients when the continuous frequency response is known.

Exercise 3.12: Plotting a simple frequency response

Let again S be the filter defined by the equation

$$z_n = \frac{1}{4}x_{n+1} + \frac{1}{4}x_n + \frac{1}{4}x_{n-1} + \frac{1}{4}x_{n-2},$$

as in Exercise 3.1. Compute and plot (the magnitude of) $\lambda_S(\omega)$.

Exercise 3.13: Low-pass and high-pass filters

A filter S is defined by the equation

$$z_n = \frac{1}{3}(x_n + 3x_{n-1} + 3x_{n-2} + x_{n-3}).$$

a) Compute and plot the (magnitude of the continuous) frequency response of the filter, i.e. $|\lambda_S(\omega)|$. Is the filter a low-pass filter or a high-pass filter?

b) Find an expression for the vector frequency response $\lambda_{S,2}$. What is $S\boldsymbol{x}$ when $\boldsymbol{x}$ is the vector of length N with components $e^{2\pi i 2k/N}$?

Exercise 3.14: Circulant matrices

A filter S_1 is defined by the equation

$$z_n = \frac{1}{16}(x_{n+2} + 4x_{n+1} + 6x_n + 4x_{n-1} + x_{n-2}).$$

a) Write down an 8×8 circulant Toeplitz matrix which corresponds to applying S_1 on a periodic signal with period $N = 8$.

b) Compute and plot (the continuous) frequency response of the filter. Is the filter a low-pass filter or a high-pass filter?

c) Another filter S_2 has (continuous) frequency response $\lambda_{S_2}(\omega) = (e^{i\omega} + 2 + e^{-i\omega})/4$. Write down the filter coefficients for the filter S_1S_2.

Exercise 3.15: Composite filters

Assume that the filters S_1 and S_2 have the frequency responses $\lambda_{S_1}(\omega) = 2 + 4\cos(\omega)$, $\lambda_{S_2}(\omega) = 3\sin(2\omega)$.

a) Compute and plot the frequency response of the filter S_1S_2.

b) Write down the filter coefficients t_k and the impulse response $\boldsymbol{s}$ for the filter S_1S_2.

Exercise 3.16: Maximum and minimum

Compute and plot the continuous frequency response of the filter $S = \{1/4, \underline{1/2}, 1/4\}$. Where does the frequency response achieve its maximum and minimum value, and what are these values?

Exercise 3.17: Plotting a simple frequency response

Plot the continuous frequency response of the filter $T = \{1/4, -\underline{1/2}, 1/4\}$. Where does the frequency response achieve its maximum and minimum value, and what are these values? Can you write down a connection between this frequency response and that from Exercise 3.16?

Exercise 3.18: Continuous- and vector frequency responses

Define the filter S by $S = \{1, 2, \underline{3}, 4, 5, 6\}$. Write down the matrix for S when $N = 8$. Plot (the magnitude of) $\lambda_S(\omega)$, and indicate the values $\lambda_{S,n}$ for $N = 8$ in this plot.

Exercise 3.19: Starting with circulant matrices

Given the circulant Toeplitz matrix

$$S = \frac{1}{5}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 1 \\ 1 & 1 & 0 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix}$$

Write down the compact notation for this filter. Compute and plot (the magnitude) of $\lambda_S(\omega)$.

Exercise 3.20: When the filter coefficients are powers

Assume that $S = \{\underline{1}, c, c^2, \ldots, c^k\}$. Compute and plot $\lambda_S(\omega)$ when $k = 4$ and $k = 8$. How does the choice of k influence the frequency response? How does the choice of c influence the frequency response?

Exercise 3.21: The Hanning window

The Hanning window is defined by $w_n = 1 - \cos(2\pi n/(N-1))$. Compute and plot the window coefficients and the continuous frequency response of this window for $N = 32$, and compare with the window coefficients and the frequency responses for the rectangular- and the Hamming window.

3.4 Some examples of filters

We have now established the basic theory of filters, and it is time to study some specific examples.

Example 3.30. *Time delay filters.*

We have already encountered the time-delay filter $S = E_d$. With only one nonzero diagonal, this is the simplest possible type of filters. Since $\boldsymbol{s} = E_d\boldsymbol{e}_0 = \boldsymbol{e}_d$, we can write $E_d = \{\underline{0}, \ldots, 1\}$, where the 1 occurs at position d. Intuitively, we would expect that time-delay does not change the frequencies in sounds we hear. This is confirmed by the fact that the frequency response of the time delay filter is $\lambda_S(\omega) = e^{-id\omega}$, which has magnitude 1, so that the filter does not change the magnitude of the different frequencies.

Fact 3.31. *Adding echo.*

Let $\boldsymbol{x}$ be a digital sound. Then the sound $\boldsymbol{z}$ with samples given by

```
N, nchannels = shape(x)
z = zeros((N,nchannels))
z[0:d] = x[0:d]
z[d:N] = x[d:N] + c*x[0:(N-d)]
```

will include an echo of the original sound. `d` is the delay in samples, and is an integer. c is a constant called the damping factor, and is usually smaller than 1.

Example 3.32. *Adding echo.*

An echo is a copy of the sound that is delayed and softer than the original sound. The sample that comes t seconds before sample i has index $i - tf_s$ where f_s is the sampling rate. This also makes sense even if t is not an integer so we can use this to produce delays that are less than one second. The one complication with this is that the number tf_s may not be an integer. We can get round this by rounding it to the nearest integer. This corresponds to adjusting the echo slightly. The following holds:

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

Using our compact filter notation, the filter which adds echo can be written as

$$S = \{\underline{1}, 0, \ldots, 0, c\},$$

where the damping factor c appears after the delay d. The frequency response of this is $\lambda_S(\omega) = 1 + ce^{-id\omega}$. This frequency response is not real, which means that the filter is not symmetric. In Figure 3.9 we have plotted the magnitude of this frequency response with $c = 0.1$ and $d = 10$.

We see that the response varies between 0.9 and 1.1, so that the deviation from 1 is controlled by the damping factor c. Also, we see that the oscillation in the frequency response, as visible in the plot, is controlled by the delay d.

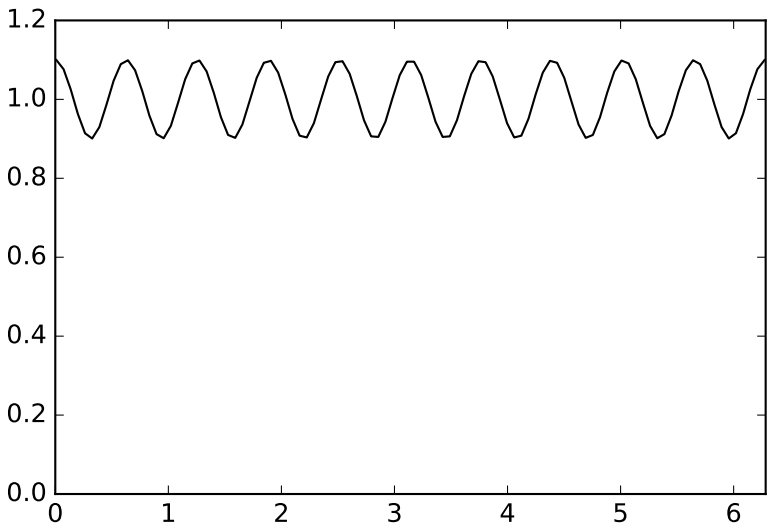

Figure 3.9: The frequency response of a filter which adds an echo with damping factor $c = 0.1$ and delay $d = 10$.

Let us now take a look at some filters which adjust the bass and treble in sound. The fact that the filters are useful for these purposes will be clear when we plot the frequency response.

Example 3.33. *Reducing the treble with moving average filters.*

The treble in a sound is generated by the fast oscillations (high frequencies) in the signal. If we want to reduce the treble we have to adjust the sample values in a way that reduces those fast oscillations. A general way of reducing variations in a sequence of numbers is to replace one number by the average of itself and its neighbors, and this is easily done with a digital sound signal. If $\boldsymbol{z} = (z_i)_{i=0}^{N-1}$ is the sound signal produced by taking the average of three successive samples, we have that

$$z_n = \frac{1}{3}(x_{n+1} + x_n + x_{n-1}),$$

i.e. $S = \{1/3, 1/3, 1/3\}$. This filter is also called a *moving average filter* (with three elements), and it can be written in compact form as. If we set $N = 4$, the corresponding circulant Toeplitz matrix for the filter is

$$S = \frac{1}{3}\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

The frequency response is

$$\lambda_S(\omega) = (e^{i\omega} + 1 + e^{-i\omega})/3 = (1 + 2\cos(\omega))/3.$$

More generally we can construct the moving average filter of $2L+1$ elements, which is $S = \{1, \cdots, \underline{1}, \cdots, 1\}/(2L+1)$, where there is symmetry around 0. Clearly then the first column of S is $\boldsymbol{s} = (\underbrace{1, \ldots, 1}_{L+1 \text{ times}}, 0, \ldots, 0, \underbrace{1, \ldots, 1}_{L \text{ times}})/(2L+1)$. In Example 2.15 we computed that the DFT of the vector $\boldsymbol{x} = (\underbrace{1, \ldots, 1}_{L+1 \text{ times}}, 0, \ldots, 0, \underbrace{1, \ldots, 1}_{L \text{ times}})$ had components

$$y_n = \frac{\sin(\pi n(2L+1)/N)}{\sin(\pi n/N)}.$$

Since $\boldsymbol{s} = \boldsymbol{x}/(2L+1)$ and $\lambda_S = \text{DFT}_N \boldsymbol{s}$, the frequency response of S is

$$\lambda_{S,n} = \frac{1}{2L+1} \frac{\sin(\pi n(2L+1)/N)}{\sin(\pi n/N)},$$

so that

$$\lambda_S(\omega) = \frac{1}{2L+1} \frac{\sin((2L+1)\omega/2)}{\sin(\omega/2)}.$$

We clearly have

$$0 \leq \frac{1}{2L+1} \frac{\sin((2L+1)\omega/2)}{\sin(\omega/2)} \leq 1,$$

and this frequency response approaches 1 as $\omega \to 0$. The frequency response thus peaks at 0, and this peak gets narrower and narrower as L increases, i.e. as we use more and more samples in the averaging process. This filter thus "keeps" only the lowest frequencies. When it comes to the highest frequencies it is seen that the frequency response is small for $\omega \approx \pi$. In fact it is straightforward to see that $|\lambda_S(\pi)| = 1/(2L+1)$. In Figure 3.10 we have plotted the frequency response for moving average filters with $L = 1$, $L = 5$, and $L = 20$.

Unfortunately, the frequency response is far from a filter which keeps some frequencies unaltered, while annihilating others: Although the filter distinguishes between high and low frequencies, it slightly changes the small frequencies. Moreover, the higher frequencies are not annihilated, even when we increase L to high values.

In the previous example we mentioned filters which favor certain frequencies of interest, while annihilating the others. This is a desirable property for filters, so let us give names to such filters:

Definition 3.34. *Lowpass and highpass filters.*
A filter S is called

- a *lowpass filter* if $\lambda_S(\omega)$ is large when ω is close to 0, and $\lambda_S(\omega) \approx 0$ when ω is close to π (i.e. S keeps low frequencies and annhilates high frequencies),

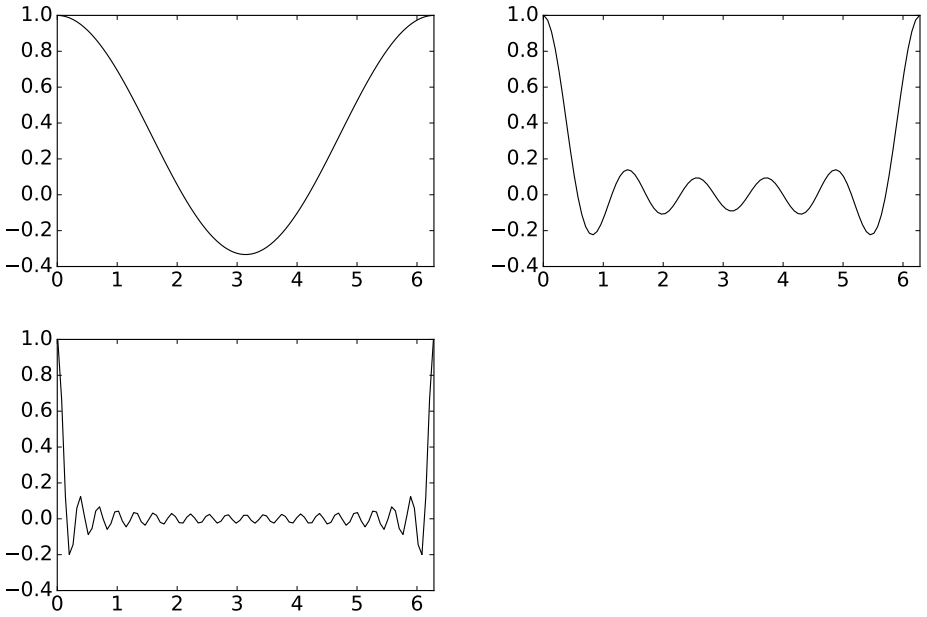

Figure 3.10: The frequency response of moving average filters with $L = 1$, $L = 5$, and $L = 20$.

- a *highpass filter* if $\lambda_S(\omega)$ is large when ω is close to π, and $\lambda_S(\omega) \approx 0$ when ω is close to 0 (i.e. S keeps high frequencies and annhilates low frequencies),
- a *bandpass filter* if $\lambda_S(\omega)$ is large within some interval $[a, b] \subset [0, 2\pi]$, and $\lambda_S(\omega) \approx 0$ outside this interval.

This definition should be considered rather vague when it comes to what we mean by "ω close to $0, \pi$", and "$\lambda_S(\omega)$ is large": in practice, when we talk about lowpass and highpass filters, it may be that the frequency responses are still quite far from what is commonly refered to as *ideal lowpass or highpass filters*, where the frequency response only assumes the values 0 and 1 near 0 and π. The next example considers an ideal lowpass filter.

Example 3.35. *Ideal lowpass filters.*

By definition, the ideal lowpass filter keeps frequencies near 0 unchanged, and completely removes frequencies near π. We now have the theory in place in order to find the filter coefficients for such a filter: In Example 2.27 we implemented the ideal lowpass filter with the help of the DFT. Mathematically you can see that this code is equivalent to computing $(F_N)^H D F_N$ where D is the diagonal matrix with the entries $0, \ldots, L$ and $N - L, \ldots, N - 1$ being 1, the rest being 0. Clearly this is a digital filter, with frequency response as stated. If the filter should keep the angular frequencies $|\omega| \leq \omega_c$ only, where ω_c describes the highest frequency we should keep, we should choose L so that $\omega_c = 2\pi L/N$. Again,

in Example 2.15 we computed the DFT of this vector, and it followed from Theorem 2.18 that the IDFT of this vector equals its DFT. This means that we can find the filter coefficients by using Equation (3.10), i.e. we take an IDFT. We then get the filter coefficients

$$\frac{1}{N}\frac{\sin(\pi k(2L+1)/N)}{\sin(\pi k/N)}.$$

This means that the filter coefficients lie as N points uniformly spaced on the curve $\frac{1}{N}\frac{\sin(\pi t(2L+1)/2)}{\sin(\pi t/2)}$ between 0 and 1. This curve has been encountered many other places in these notes. The filter which keeps only the frequency $\omega_c = 0$ has all filter coefficients being $\frac{1}{N}$ (set $L = 1$), and when we include all frequencies (set $L = N$) we get the filter where $x_0 = 1$ and all other filter coefficients are 0. When we are between these two cases, we get a filter where s_0 is the biggest coefficient, while the others decrease towards 0 along the curve we have computed. The bigger L and N are, the quicker they decrease to zero. All filter coefficients are usually nonzero for this filter, since this curve is zero only at certain points. This is unfortunate, since it means that the filter is time-consuming to compute.

The two previous examples show an important duality between vectors which are 1 on some elements and 0 on others (also called window vectors), and the vector $\frac{1}{N}\frac{\sin(\pi k(2L+1)/N)}{\sin(\pi k/N)}$ (also called a sinc): filters of the one type correspond to frequency responses of the other type, and vice versa. The examples also show that, in some cases only the filter coefficients are known, while in other cases only the frequency response is known. In any case we can deduce the one from the other, and both cases are important.

Filters are much more efficient when there are few nonzero filter coefficients. In this respect the second example displays a problem: in order to create filters with particularly nice properties (such as being an ideal lowpass filter), one may need to sacrifice computational complexity by increasing the number of nonzero filter coefficients. The trade-off between computational complexity and desirable filter properties is a very important issue in *filter design theory*.

Example 3.36. *Dropping filter coefficients.*

In order to decrease the computational complexity for the ideal lowpass filter in Example 3.35, one can for instance include only the first filter coefficients, i.e.

$$\left\{\frac{1}{N}\frac{\sin(\pi k(2L+1)/N)}{\sin(\pi k/N)}\right\}_{k=-N_0}^{N_0},$$

ignoring the last ones. Hopefully this gives us a filter where the frequency reponse is not that different from the ideal lowpass filter. In Figure 3.11 we show the corresponding frequency responses. In the figure we have set $N = 128$, $L = 32$, so that the filter removes all frequencies $\omega > \pi/2$. N_0 has been chosen so that the given percentage of all coefficients are included.

Clearly the figure shows that we should be careful when we omit filter coefficients: if we drop too many, the frequency response is far away from that of

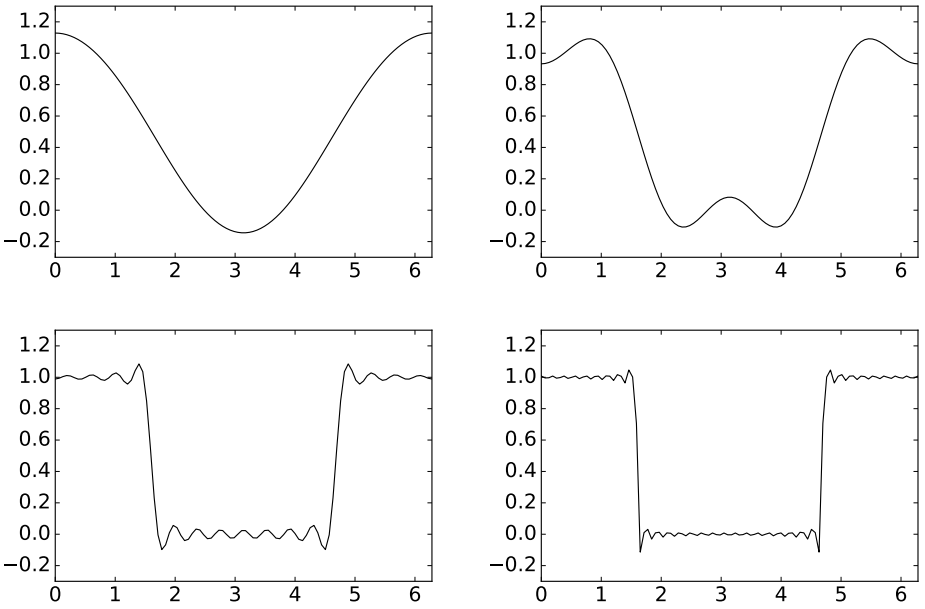

Figure 3.11: The frequency response which results by including the first 1/32, the first 1/16, the first 1/4, and and all of the filter coefficients for the ideal lowpass filter.

an ideal bandpass filter. In particular, we see that the new frequency response oscillates wildly near the discontinuity of the ideal lowpass filter. Such oscillations are called *Gibbs oscillations*.

Example 3.37. *Filters and the MP3 standard.*

We mentioned previously that the MP3 standard splits the sound into frequency bands. This splitting is actually performed by particular filters, which we will consider now.

In the example above, we saw that when we dropped the last filter coefficients in the ideal lowpass filter, there were some undesired effects in the frequency response of the resulting filter. Are there other and better approximations to the ideal lowpass filter which uses the same number of filter coefficients? This question is important, since the ear is sensitive to certain frequencies, and we would like to extract these frequencies for special processing, using as low computational complexity as possible. In the MP3-standard, such filters have been constructed. These filters are more advanced than the ones we have seen upto now. They have as many as 512 filter coefficients! We will not go into the details on how these filters are constructed, but only show how their frequency responses look. In the left plot in Figure 3.12, the "prototype filter" which is used in the MP3 standard is shown. We see that this is very close to an ideal lowpass filter. Moverover, many of the undesirable effect from the previous

example have been eliminated: The oscillations near the discontinuities are much smaller, and the values are lower away from 0. Using Property 4 in Theorem 2.18, it is straightforward to construct filters with similar frequency responses, but centered around different frequencies: We simply need to multiply the filter coefficients with a complex exponential, in order to obtain a filter where the frequency response has been shifted to the left or right. In the MP3 standard, this observation is used to construct 32 filters, each having a frequency response which is a shifted copy of that of the prototype filter, so that all filters together cover the entire frequency range. 5 of these frequency responses are shown in the right plot in Figure 3.12.

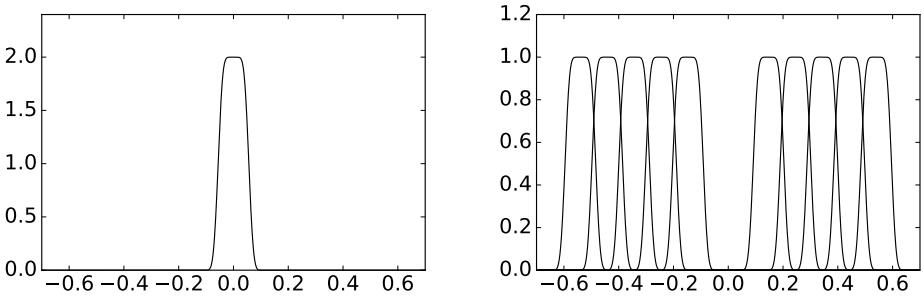

Figure 3.12: Frequency responses of some filters used in the MP3 standard. The prototype filter is shown left. The other frequency responses at right are simply shifted copies of this.

To understand the effects of the different filters, let us apply them to our sample sound. If you apply all filters in the MP3 standard in successive order with the most lowpass filters first, the result will sound like this. You should interpret the result as low frequencies first, followed by the high frequencies. π corresponds to the frequency 22.05KHz (i.e. the highest representable frequency equals half the sampling rate on 44.1KHz. The different filters are concentrated on 1/32 of these frequencies each, so that the angular frequencies you here are $[\pi/64, 3\pi/64]$, $[3\pi/64, 5\pi/64]$, $[5\pi/64, 7\pi/64]$, and so on, in that order.

In Section 3.3.1 we mentioned that the psycoacoustic model of the MP3 standard applied a window the the sound data, followed by an FFT to that data. This is actually performed in parallel on the same sound data. Applying two different operations in parallel to the sound data may seem strange. In the MP3 standard [16] (p. 109) this is explained by "the lack of spectral selectivity obtained at low frequencies" by the filters above. In other words, the FFT can give more precise frequency information than the filters can. This more precise information is then used to compute psychoacoustic information such as masking thresholds, and this information is applied to the output of the filters.

Example 3.38. *Reducing the treble II.*

When reducing the treble it is reasonable to let the middle sample x_i count more than the neighbors in the average, so an alternative is to compute the average by instead writing

$$z_n = (x_{n-1} + 2x_n + x_{n+1})/4$$

The coefficients $1, 2, 1$ here have been taken from row 2 in Pascal's triangle. It turns out that this is a good choice of coefficients. Also if we take averages of more numbers it will turn out that higher rows of Pascals triangle are good choices. Let us take a look at why this is the case. Let S be the moving average filter of two elements, i.e.

$$(S\boldsymbol{x})_n = \frac{1}{2}(x_{n-1} + x_n).$$

In Example 3.33 we had an odd number of filter coefficients. Here we have only two. We see that the frequency response in this case is

$$\lambda_S(\omega) = \frac{1}{2}(1 + e^{-i\omega}) = e^{-i\omega/2}\cos(\omega/2).$$

The frequency response is complex now, since the filter is not symmetric in this case. Let us now apply this filter k times, and denote by S_k the resulting filter. Theorem 3.26 gives us that the frequency response of S_k is

$$\lambda_{S^k}(\omega) = \frac{1}{2^k}(1 + e^{-i\omega})^k = e^{-ik\omega/2}\cos^k(\omega/2),$$

which is a polynomial in $e^{-i\omega}$ with the coefficients taken from Pascal's triangle (remember that the values in Pascals triangle are the coefficients of x in the expression $(1 + x)^k$, i.e. the binomial coefficients $\binom{k}{r}$ for $0 \leq r \leq k$). At least, this partially explains how filters with coefficients taken from Pascal's triangle appear. The reason why these are more desirable than moving average filters, and are used much for smoothing abrupt changes in images and in sound, is the following: Since $(1+e^{-i\omega})^k$ is a factor in $\lambda_{S^k}(\omega)$, it has a zero of multiplicity of k at $\omega = \pi$. In other words, when k is large, λ_{S^k} has a zero of high multiplicity at $\omega = \pi$, and this implies that the frequency response is very flat for $\omega \approx \pi$ when k increases, i.e. the filter is good at removing the highest frequencies. This can be seen in Figure 3.13, where we have plotted the magnitude of the frequency response when $k = 5$, and when $k = 30$. Clearly the latter frequency response is much flatter for $\omega \approx \pi$. On the other side, it is easy to show that the moving average filters of Example 3.33 had a zero of multiplicity one at $\omega = \pi$, regardless of L. Clearly, the corresponding frequency responses, shown in Figure 3.10, were not as flat for $\omega \approx \pi$, when compared to the ones in Figure 3.13.

While using S^k gives a desirable behaviour for $\omega \approx \pi$, we see that the behaviour is not so desirable for small frequencies $\omega \approx 0$: Only frequencies very close to 0 are kept unaltered. It should be possible to produce better lowpass filters than this also, and the frequency responses we plotted for the filters used in the MP3 standard gives an indication to this.

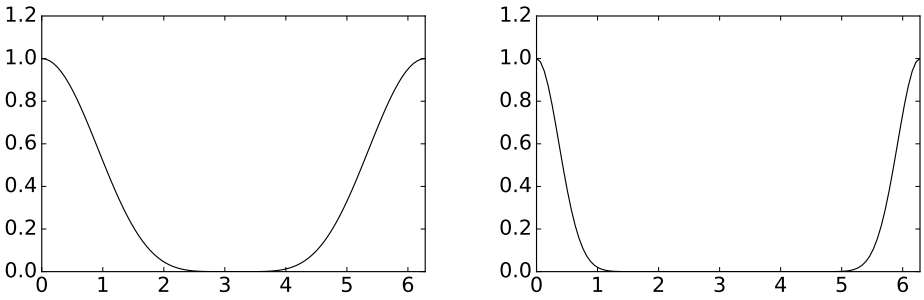

Figure 3.13: The frequency response of filters corresponding to iterating the moving average filter $\{\underline{1/2}, 1/2\}$ $k = 5$ and $k = 30$ times (i.e. using row k in Pascal's triangle).

Let us now see how to implement the filters S^k. Since convolution corresponds to multiplication of polynomials, we can obtain their filter coefficients with the following code

```
t = [1.]
for kval in range(k):
    t = convolve(t, [1/2., 1/2.])
```

Note that S^k has $k+1$ filter coefficients, and that S^k corresponds to the filter coefficients of a symmetric filter when k is even. Having computed `t`, we can simply compute the convolution of the input `x` and `t`. In using `conv` we disregard the circularity of S, and we introduce a time delay. These issues will, however, not be audible when we listen to the output. An example of the result of smoothing is shown in Figure 3.14.

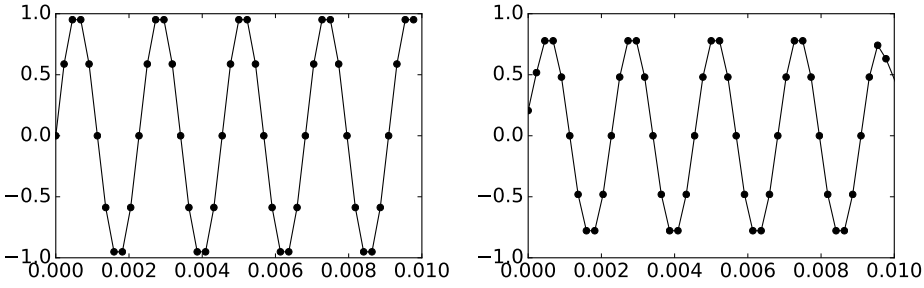

Figure 3.14: Reducing the treble. The original sound signal is shown left, the result after filtering using row 4 in Pascal's triangle is shown right.

The left plot shows the samples of the pure sound with frequency 440Hz (with sampling frequency $f_s = 4400$Hz). The right plot shows the result of applying the averaging process by using row 4 of Pascals triangle. We see that the oscillations have been reduced. In Exercise 3.25 you will be asked to implement reducing

the treble in our sample audio file. If you do this you should hear that the sound gets softer when you increase k: For $k = 32$ the sound will be like this, for $k = 256$ it will be like this.

Picking coefficients from a row in Pascals triangle works better the longer the filter is:

Observation 3.39. *Reducing the treble.*
Let $\boldsymbol{x}$ be the samples of a digital sound, and let S be a filter with coefficients taken from row k of Pascals triangle. Then $S\boldsymbol{x}$ has reduced treble when compared to $\boldsymbol{x}$.

Another common option in an audio system is reducing the bass. This corresponds to reducing the low frequencies in the sound, or equivalently, the slow variations in the sample values. It turns out that this can be accomplished by simply changing the sign of the coefficients used for reducing the treble. Let us explain why this is the case. Let S_1 be a filter with filter coefficients t_k, and let us consider the filter S_2 with filter coefficient $(-1)^k t_k$. The frequency response of S_2 is

$$\begin{aligned}\lambda_{S_2}(\omega) &= \sum_k (-1)^k t_k e^{-i\omega k} = \sum_k (e^{-i\pi})^k t_k e^{-i\omega k} \\ &= \sum_k e^{-i\pi k} t_k e^{-i\omega k} = \sum_k t_k e^{-i(\omega+\pi)k} = \lambda_{S_1}(\omega+\pi).\end{aligned}$$

where we have set $e^{-i\pi} = -1$ (note that this is nothing but Property 4. in Theorem 2.18, with $d = N/2$). Now, for a lowpass filter S_1, $\lambda_{S_1}(\omega)$ has large values when ω is close to 0 (the low frequencies), and values near 0 when ω is close to π (the high frequencies). For a highpass filter S_2, $\lambda_{S_2}(\omega)$ has values near 0 when ω is close to 0 (the low frequencies), and large values when ω is close to π (the high frequencies). When S_2 is obtained by adding an alternating sign to the filter coefficicents of S_1, The relation $\lambda_{S_2}(\omega) = \lambda_{S_1}(\omega+\pi)$ thus says that S_2 is a highpass filter when S_1 is a lowpass filter, and vice versa:

Observation 3.40. *Passing between lowpass- and highpass filters.*
Assume that S_2 is obtained by adding an alternating sign to the filter coefficicents of S_1. If S_1 is a lowpass filter, then S_2 is a highpass filter. If S_1 is a highpass filter, then S_2 is a lowpass filter.

The following example elaborates further on this.

Example 3.41. *Reducing the bass.*
Consider the bass-reducing filter deduced from the fourth row in Pascals triangle:

$$z_n = \frac{1}{16}(x_{n-2} - 4x_{n-1} + 6x_n - 4x_{n+1} + x_{n+2})$$

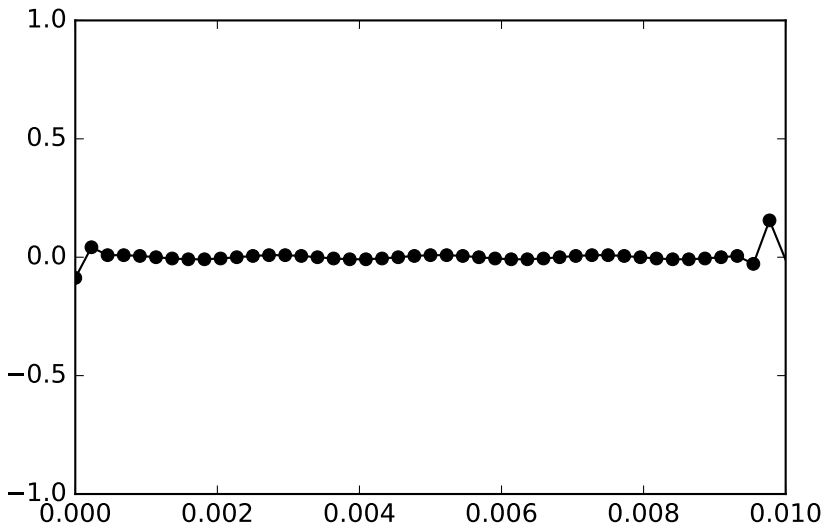

Figure 3.15: The result of applying the bass-reducing filter deduced from row 4 in Pascals triangle to the pure sound in the left plot of Figure 3.14.

The result of applying this filter to the sound in Figure 3.14 is shown in Figure 3.15.

We observe that the samples oscillate much more than the samples of the original sound. In Exercise 3.25 you will be asked to implement reducing the bass in our sample audio file. The new sound will be difficult to hear for large k, and we will explain why later. For $k = 1$ the sound will be like this, for $k = 2$ it will be like this. Even if the sound is quite low, you can hear that more of the bass has disappeared for $k = 2$.

The frequency response we obtain from using row 5 of Pascal's triangle is shown in Figure 3.16. It is just the frequency response of the corresponding treble-reducing filter shifted with π. The alternating sign can also be achieved if we write the frequency response $\frac{1}{2^k}(1+e^{-i\omega})^k$ from Example 3.38 as $\frac{1}{2^k}(1-e^{-i\omega})^k$, which corresponds to applying the filter $S(\boldsymbol{x}) = \frac{1}{2}(-x_{n-1} + x_n)$ k times.

If we play a sound after such a bass-reducing filter has been applied to it, the bass will be reduced:

Observation 3.42. *Pascals triangle and reducing the bass.*

Let $\boldsymbol{x}$ be the samples of a digital sound, and let S be a filter with filter coefficients taken from row k of Pascal's triangle, and add an alternating sign to the filter coefficients. Then $S\boldsymbol{x}$ has reduced bass when compared to $\boldsymbol{x}$.

What you should have learned in this section.

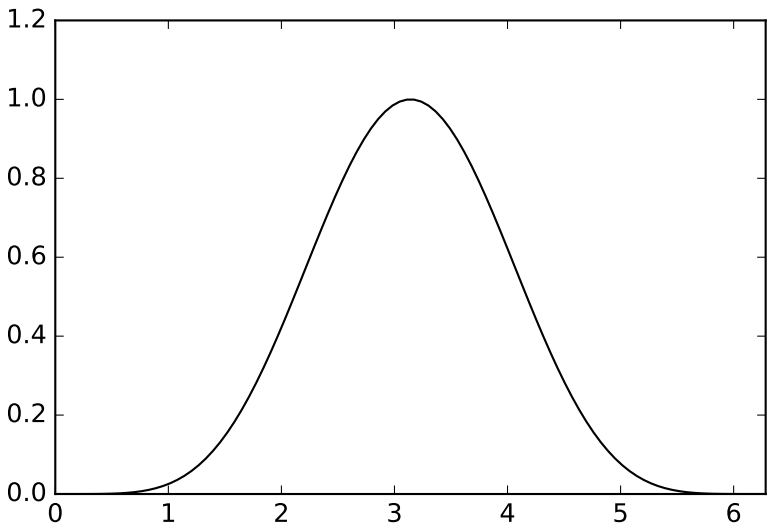

Figure 3.16: The frequency response of the bass reducing filter, which corresponds to row 5 of Pascal's triangle.

- Simple examples of filters, such as time delay filters and filters which add echo.
- Lowpass and highpass filters and their frequency responses, and their interpretation as treble- and bass-reducing filters. Moving average filters, and filters arising from rows in Pascal's triangle, as examples of such filters.
- How to pass between lowpass and highpass filters by adding an alternating sign to the filter cofficients.

Exercise 3.22: Composing time delay filters

Let E_{d_1} and E_{d_2} be two time delay filters. Show that $E_{d_1}E_{d_2} = E_{d_1+d_2}$ (i.e. that the composition of two time delays again is a time delay) in two different ways:

a) Give a direct argument which uses no computations.

b) By using Property 3 in Theorem 2.18, i.e. by using a property for the Discrete Fourier Transform.

Exercise 3.23: Adding echo

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

a) Write a function `play_with_echo` which takes the sound samples, the sample rate, a damping constant `c`, and a delay `d` as input, and plays the sound samples with an echo added, as described in Example 3.32. Recall that you have to ensure that the sound samples lie in $[-1, 1]$.

b) Generate the sound from Example 3.32, and verify that it is the same as the one you heard there.

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

Exercise 3.24: Adding echo filters

Consider the two filters $S_1 = \{\underline{1}, 0, \ldots, 0, c\}$ and $S_2 = \{\underline{1}, 0, \ldots, 0, -c\}$. Both of these can be interpreted as filters which add an echo. Show that $\frac{1}{2}(S_1 + S_2) = I$. What is the interpretation of this relation in terms of echos?

Exercise 3.25: Reducing bass and treble

In this exercise, we will experiment with increasing and reducing the treble and bass in a signal as in examples 3.38 and 3.41.

a) Write functions `play_with_reduced_treble` and `play_with_reduced_bass` which take a data vector, sampling rate, and k as input, and which reduce bass and treble, respectively, in the ways described above, and plays the result, when row number $2k$ in Pascal' triangle is used to construct the filters. Use the function `convolve` to help you to find the values in Pascal's triangle. You can use the `convolve` function also to compute the output of the filter, but note that this disregards the circularity of the filter. If you solved Exercise 3.6, you can also use the function `filterS` you implemented there, since row $2k$ in Pascal's triangle has an odd number of values, and thus corresponds to a symmetric filter.

b) Generate the sounds you heard in examples 3.38 and 3.41, and verify that they are the same.

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

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

Exercise 3.26: Constructing a highpass filter

Consider again Example 3.35. Find an expression for a filter so that only frequencies so that $|\omega - \pi| < \omega_c$ are kept, i.e. the filter should only keep angular frequencies close to π (i.e. here we construct a highpass filter).

Exercise 3.27: Combining lowpass and highpass filters

In this exercise we will investigate how we can combine lowpass and highpass filters to produce other filters

a) Assume that S_1 and $S2$ are lowpass filters. What kind of filter is S_1S_2? What if both S_1 and S_2 are highpass filters?

b) Assume that one of S_1, S_2 is a highpass filter, and that the other is a lowpass filter. What kind of filter S_1S_2 in this case?

Exercise 3.28: Composing filters

A filter S_1 has the frequency response $\frac{1}{2}(1+\cos\omega)$, and another filter has the frequency response $\frac{1}{2}(1+\cos(2\omega))$.

a) Is S_1S_2 a lowpass filter, or a highpass filter?

b) What does the filter S_1S_2 do with angular frequencies close to $\omega = \pi/2$.

c) Find the filter coefficients of S_1S_2.

Hint. Use Theorem 3.26 to compute the frequency response of S_1S_2 first.

d) Write down the matrix of the filter S_1S_2 for $N = 8$.

Exercise 3.29: Composing filters

An operation describing some transfer of data in a system is defined as the composition of the following three filters:

- First a time delay filter with delay $d_1 = 2$, due to internal transfer of data in the system,
- then the treble-reducing filter $T = \{1/4, \underline{1/2}, 1/4\}$,
- finally a time delay filter with delay $d_2 = 4$ due to internal transfer of the filtered data.

We denote by $T_2 = E_{d_2}TE_{d_1} = E_4TE_2$ the operation which applies these filters in succession.

a) Explain why T_2 also is a digital filter. What is (the magnitude of) the frequency response of E_{d_1}? What is the connection between (the magnitude of) the frequency response of T and T_2?

b) Show that $T_2 = \{\underline{0}, 0, 0, 0, 0, 1/4, 1/2, 1/4\}$.

Hint. Use the expressions $(E_{d_1}\boldsymbol{x})_n = x_{n-d_1}$, $(T\boldsymbol{x})_n = \frac{1}{4}x_{n+1} + \frac{1}{2}x_n + \frac{1}{4}x_{n-1}$, $(E_{d_2}\boldsymbol{x})_n = x_{n-d_2}$, and compute first $(E_{d_1}\boldsymbol{x})_n$, then $(TE_{d_1}\boldsymbol{x})_n$, and finally $(T_2\boldsymbol{x})_n = (E_{d_2}TE_{d_1}\boldsymbol{x})_n$. From the last expression you should be able to read out the filter coefficients.

c) Assume that $N = 8$. Write down the 8×8-circulant Toeplitz matrix for the filter T_2.

Exercise 3.30: Filters in the MP3 standard

In Example 3.37, we mentioned that the filters used in the MP3-standard were constructed from a lowpass prototype filter by multiplying the filter coefficients with a complex exponential. Clearly this means that the new frequency response is a shift of the old one. The disadvantage is, however, that the new filter coefficients are complex. It is possible to address this problem as follows. Assume that t_k are the filter coefficients of a filter S_1, and that S_2 is the filter with filter coefficients $\cos(2\pi kn/N)t_k$, where $n \in \mathbb{N}$. Show that

$$\lambda_{S_2}(\omega) = \frac{1}{2}(\lambda_{S_1}(\omega - 2\pi n/N) + \lambda_{S_1}(\omega + 2\pi n/N)).$$

In other words, when we multiply (modulate) the filter coefficients with a cosine, the new frequency response can be obtained by shifting the old frequency response with $2\pi n/N$ in both directions, and taking the average of the two.

Exercise 3.31: Explain code

a) Explain what the code below does, line by line.

```
x, fs = audioread('sounds/castanets.wav')
N, nchannels = shape(x)
z = zeros((N, nchannels))
for n in range(1,N-1):
    z[n] = 2*x[n+1] + 4*x[n] + 2*x[n-1]
z[0] = 2*x[1] + 4*x[0] + 2*x[N-1]
z[N-1] = 2*x[0] + 4*x[N-1] + 2*x[N-2]
z = z/abs(z).max()
play(z, fs)
```

Comment in particular on what happens in the three lines directly after the `for`-loop, and why we do this. What kind of changes in the sound do you expect to hear?

b) Write down the compact filter notation for the filter which is used in the code, and write down a 5×5 circulant Toeplitz matrix which corresponds to this filter. Plot the (continuous) frequency response. Is the filter a lowpass- or a highpass filter?

c) Another filter is given by the circulant Toeplitz matrix

$$\begin{pmatrix} 4 & -2 & 0 & 0 & -2 \\ -2 & 4 & -2 & 0 & 0 \\ 0 & -2 & 4 & -2 & 0 \\ 0 & 0 & -2 & 4 & -2 \\ -2 & 0 & 0 & -2 & 4 \end{pmatrix}.$$

Express a connection between the frequency responses of this filter and the filter from b. Is the new filter a lowpass- or a highpass filter?

3.5 More general filters

The starting point for defining filters at the beginning of this chapter was equations on the form

$$z_n = \sum_k t_k x_{n-k}.$$

For most filters we have looked at, we had a limited number of nonzero t_k, and this enabled us to compute them on a computer using a finite number of additions and multiplications. Filters which have a finite number of nonzero filter coefficients are also called *FIR-filters* (FIR is short for Finite Impulse Response. Recall that the impulse response of a filter can be found from the filter coefficients). However, there exist many useful filters which are not FIR filters, i.e. where the sum above is infinite. The ideal lowpass filter from Example 3.35 was one example. It turns out that many such cases can be made computable if we change our procedure slightly. The old procedure for computing a filter is to compute $\boldsymbol{z} = S\boldsymbol{x}$. Consider the following alternative:

Idea 3.43. *More general filters (1).*
Let $\boldsymbol{x}$ be the input to a filter, and let T be a filter. By solving the system $T\boldsymbol{z} = \boldsymbol{x}$ for $\boldsymbol{z}$ we get another filter, which we denote by S.

Of course T must then be the inverse of S (which also is a filter), but the point is that the inverse of a filter may have a finite number of filter coefficicents, even if the filter itself does not. In such cases this new procedure is more attractive that the old one, since the equation system can be solved with few arithmetic operations when T has few filter coefficients.

It turns out that there also are highly computable filters where neither the filter nor its inverse have a finite number of filter coefficients. Consider the following idea:

Idea 3.44. *More general filters (2).*
Let $\boldsymbol{x}$ be the input to a filter, and let U and V be filters. By solving the system $U\boldsymbol{z} = V\boldsymbol{x}$ for $\boldsymbol{z}$ we get another filter, which we denote by S. The filter S can be implemented in two steps: first we compute the right hand side $\boldsymbol{y} = V\boldsymbol{x}$, and then we solve the equation $U\boldsymbol{z} = \boldsymbol{y}$.

If both U and V are invertible we have that the filter is $S = U^{-1}V$, and this is invertible with inverse $S^{-1} = V^{-1}U$. The point is that, when U and V have a finite number of filter coefficicents, both S and its inverse will typically have an infinite number of filter coefficients. The filters from this idea are thus more general than the ones from the previous idea, and the new idea makes a wider class of filters implementable using row reduction of sparse matrices. Computing

a filter by solving $U\boldsymbol{z} = V\boldsymbol{x}$ may also give meaning when the matrices U and V are singular: The matrix system can have a solution even if U is singular. Therefore we should be careful in using the form $T = U^{-1}V$.

We have the following result concerning the frequency responses:

Theorem 3.45. *Frequency response of IIR filters.*

Assume that S is the filter defined from the equation $U\boldsymbol{z} = V\boldsymbol{x}$. Then we have that $\lambda_S(\omega) = \frac{\lambda_V(\omega)}{\lambda_U(\omega)}$ whenever $\lambda_U(\omega) \neq 0$.

Proof. Set $\boldsymbol{x} = \phi_n$. We have that $U\boldsymbol{z} = \lambda_{U,n}\lambda_{S,n}\phi_n$, and $V\boldsymbol{x} = \lambda_{V,n}\phi_n$. If the expressions are equal we must have that $\lambda_{U,n}\lambda_{S,n} = \lambda_{V,n}$, so that $\lambda_{S,n} = \frac{\lambda_{V,n}}{\lambda_{U,n}}$ for all n. By the definition of the continuous frequency response this means that $\lambda_S(\omega) = \frac{\lambda_V(\omega)}{\lambda_U(\omega)}$ whenever $\lambda_U(\omega) \neq 0$. □

The following example clarifies the points made above, and how one may construct U and V from S. The example also shows that, in addition to making some filters with infinitely many filter coefficients computable, the procedure $U\boldsymbol{z} = V\boldsymbol{x}$ for computing a filter can also reduce the complexity in some filters where we already have a finite number of filter coefficients.

Example 3.46. *Moving average filter.*

Consider again the moving average filter S from Example 3.33:

$$z_n = \frac{1}{2L+1}(x_{n+L} + \cdots + x_n + \cdots + x_{n-L}).$$

If we implemented this directly, $2L$ additions would be needed for each n, so that we would need a total of $2NL$ additions. However, we can also write

$$\begin{aligned} z_{n+1} &= \frac{1}{2L+1}(x_{n+1+L} + \cdots + x_{n+1} + \cdots + x_{n+1-L}) \\ &= \frac{1}{2L+1}(x_{n+L} + \cdots + x_n + \cdots + x_{n-L}) + \frac{1}{2L+1}(x_{n+1+L} - x_{n-L}) \\ &= z_n + \frac{1}{2L+1}(x_{n+1+L} - x_{n-L}). \end{aligned}$$

This means that we can also compute the output from the formula

$$z_{n+1} - z_n = \frac{1}{2L+1}(x_{n+1+L} - x_{n-L}),$$

which can be written on the form $U\boldsymbol{z} = V\boldsymbol{x}$ with $U = \{1, \underline{-1}\}$ and $V = \frac{1}{2L+1}\{1, 0, \ldots, 0, -1\}$ where the 1 is placed at index $-L-1$ and the -1 is placed at index L. We now perform only $2N$ additions in computing the right hand side, and solving the equation system requires only $2(N-1)$ additions. The total number of additions is thus $2N + 2(N-1) = 4N - 2$, which is much less than the previous $2LN$ when L is large.

A perhaps easier way to find U and V is to consider the frequency response of the moving average filter, which is

$$\frac{1}{2L+1}(e^{-Li\omega}+\ldots+e^{Li\omega}) = \frac{1}{2L+1}e^{-Li\omega}\frac{1-e^{(2L+1)i\omega}}{1-e^{i\omega}}$$
$$= \frac{\frac{1}{2L+1}\left(-e^{(L+1)i\omega}+e^{-Li\omega}\right)}{1-e^{i\omega}},$$

where we have used the formula for the sum of a geometric series. From here we easily see the frequency responses of U and V from the numerator and the denominator.

Filters with an infinite number of filter coefficients are also called *IIR filters* (IIR stands for *Infinite Impulse Response*). Thus, we have seen that some IIR filters may still have efficient implementations.

Exercise 3.32: A concrete IIR filter

A filter is defined by demanding that $z_{n+2} - z_{n+1} + z_n = x_{n+1} - x_n$.

a) Compute and plot the frequency response of the filter.

b) Use a computer to compute the output when the input vector is $\boldsymbol{x} = (1, 2, \ldots, 10)$. In order to do this you should write down two 10×10-circulant Toeplitz matrices.

3.6 Implementation of filters

As we saw in Example 3.46, a filter with many filter coefficients could be factored into the application of two simpler filters, and this could be used as a basis for an efficient implementation. There are also several other possible efficient implementations of filters. In this section we will consider two such techniques. The first technique considers how we can use the DFT to speed up the computation of filters. The second technique considers how we can factorize a filter into a product of simpler filters.

3.6.1 Implementation of filters using the DFT

If there are k filter coefficients, a direct implementation of a filter would require kN multiplications. Since filters are diagonalized by the DFT, one can also compute the filter as the product $S = F_N^H D F_N$. This would instead require $O(N \log_2 N)$ complex multiplications when we use the FFT algorithm, which may be a higher number of multiplications. We will however see that, by slightly changing our algorithm, we may end up with a DFT-based implementation of the filter which requires fewer multiplications.

The idea is to split the computation of the filter into smaller parts. Assume that we compute M elements of the filter at a time. If the nonzero filter coefficients of S are $t_{-k_0}, \ldots, t_{k-k_0-1}$, we have that

$$(S\boldsymbol{x})_t = \sum_r t_r x_{s-r} = t_{-k_0} x_{t+k_0} + .. + t_{k-k_0-1} x_{t-(k-k_0-1)}.$$

From this it is clear that $(S\boldsymbol{x})_t$ only depends on $x_{t-(k-k_0-1)}, \ldots, x_{t+k_0}$. This means that, if we restrict the computation of S to $x_{t-(k-k_0-1)}, \ldots, x_{t+M-1+k_0}$, the outputs $x_t, \ldots, x_{t+M-1}$ will be the same as without this restriction. This means that we can compute the output M elements at a time, at each step multiplying with a circulant Toeplitz matrix of size $(M+k-1) \times (M+k-1)$. If we choose M so that $M+k-1 = 2^r$, we can use the FFT and IFFT algorithms to compute $S = F_N^H D F_N$, and we require $O(r2^r)$ multiplications for every block of length M. The total number of multiplications is $\frac{Nr2^r}{M} = \frac{Nr2^r}{2^r-k+1}$. If $k = 128$, you can check on your calculator that the smallest value is for $r = 10$ with value $11.4158 \times N$. Since the direct implementation gives kN multiplications, this clearly gives a benefit for the new approach, it gives a 90% decrease in the number of multiplications.

3.6.2 Factoring a filter into several filters

In practice, filters are often applied in hardware, and applied in real-time scenarios where performance is a major issue. The most CPU-intensive tasks in such applications often have few memory locations available. These tasks are thus not compatible with filters with many filter coefficients, since for each output sample we then need access to many input samples and filter coefficients. A strategy which addresses this is to factorize the filter into the product of several smaller filters, and then applying each filter in turn. Since the frequency response of the product of filters equals the product of the frequency responses, we get the following idea:

Idea 3.47. *Factorizing a filter.*
Let S be a filter with real coefficients. Assume that

$$\lambda_S(\omega) = Ke^{ik\omega}(e^{i\omega} - a_1)\ldots(e^{i\omega} - a_m)(e^{2i\omega} + b_1 e^{i\omega} + c_1)\ldots(e^{2i\omega} + b_n e^{i\omega} + c_n). \tag{3.14}$$

Then we can write $S = KE_k A_1 \ldots A_m B_1 \ldots B_n$, where $A_i = \{1, \underline{-a_i}\}$ and $B_i = \{1, b_i, \underline{c_i}\}$.

Note that in Equation (3.14) a_i correspond to the real roots of the frequency response, while b_i, c_i are obtained by pairing the complex conjugate roots. Clearly the frequency responses of A_i, B_i equal the factors in the frequency response of S, which in any case can be factored into the product of filters with 2 and 3 filter coefficients, followed by a time-delay.

Note that, even though this procedure factorizes a filter into smaller parts (which is attractive for hardware implementations since smaller filters require fewer locations in memory), the number of of arithmetic operations is usually not reduced. However, consider Example 3.38, where we factorized the treble-reducing filters into a product of moving average filters of length 2 (all roots in

the previous idea are real, and equal). Each application of a moving average filter of length 2 does not really require any multiplications, since multiplication with $\frac{1}{2}$ corresponds to a bitshift. Therefore, the factorization of Example 3.38 removes the need for doing any multiplications at all, while keeping the number of additions the same. There are computational savings in this case, due to the special filter structure here.

Exercise 3.33: Implementing the factorization

Write a function `filterdftimpl`, which takes the filter coefficients $\boldsymbol{t}$ and the value k_0 from this section, computes the optimal M, and implements the filter as here.

Exercise 3.34: Factoring concrete filter

Factor the filter $S = \{\underline{1}, 5, 10, 6\}$ into a product of two filters, one with two filter coefficients, and one with three filter coefficients.

3.7 Summary

We defined digital filters, which do the same job for digital sound as analog filters do for (continuous) sound. Digital filters turned out to be linear transformations diagonalized by the DFT. We proved several other equivalent characterizations of digital filters as well, such as being time-invariant, and having a matrix which is circulant and Toeplitz. Just as for continuous sound, digital filters are characterized by their frequency response, which explains how the filter treats the different frequencies. We also went through several important examples of filters, some of which corresponded to meaningful operations on sound, such as adjustmest of bass and treble, and adding echo. We also explained that there exist filters with useful implementations which have an infinite number of filter coefficients, and we considered techniques for implementing filters efficiently. Most of the topics covered on that can also be found in [28]. We also took a look at the role of filters in the MP3 standard for compression of sound.

In signal processing literature, the assumption that vectors are periodic is often not present, and filters are thus not defined as finite-dimensional operations. With matrix notation they would then be viewed as infinite matrices which have the Toeplitz structure (i.e. constant values on the diagonals), but with no circulation. The circulation in the matrices, as well as the restriction to finite vectors, come from the assumption of a periodic vector. There are, however, also some books which view filters as circulant Toeplits matrices as we have done, such as [13].

Chapter 4

Symmetric filters and the DCT

In Chapter 1 we approximated a signal of finite duration with trigonometric functions. Since these are all periodic, there are some undesirable effects near the boundaries of the signal (at least when the values at the boundaries are different), and this resulted in a slowly converging Fourier series. This was addressed by instead considering the symmetric extension of the function, for which we obtained a more precise Fourier representation, as fewer Fourier basis vectors were needed in order to get a precise approximation.

This chapter is dedicated to addressing these thoughts for vectors. We will start by defining symmetric extensions of vectors, similarly to how we defined these for functions. Just as the Fourier series of a symmetric function was a cosine series, we will see that the symmetric extension can be viewed as a cosine vector. This gives rise to a different change of coordinates than the DFT, which we will call the DCT, which enables us to express a symmetric vector as a sum of cosine-vectors (instead of the non-symmetric complex exponentials). Since a cosine also can be associated with a given frequency, the DCT is otherwise similar to the DFT, in that it extracts the frequency information in the vector. The advantage is that the DCT can give more precise frequency information than the DFT, since it avoids the discontinuity problem of the Fourier series. This makes the DCT very practical for applications, and we will explain some of these applications. We will also show that the DCT has a a very efficient implementation, comparable with the FFT.

In this chapter we will also see that the DCT has a very similar role as the DFT when it comes to filters: just as the DFT diagonalized filters, we will see that symmetric filters can be diagonalized by the DCT, when we apply the filter to the symmetric extension of the input. We will actually show that the filters which preserve our symmetric extensions are exactly the symmetric filters.

4.1 Symmetric vectors and the DCT

As in Chapter 1, vectors can also be extended in a symmetric manner, besides the simple periodic extension procedure from Figure 2.1. In Figure 4.1 we have shown such an extension of a vector $\boldsymbol{x}$. It has $\boldsymbol{x}$ as its first half, and a copy of $\boldsymbol{x}$ in reverse order as its second half.

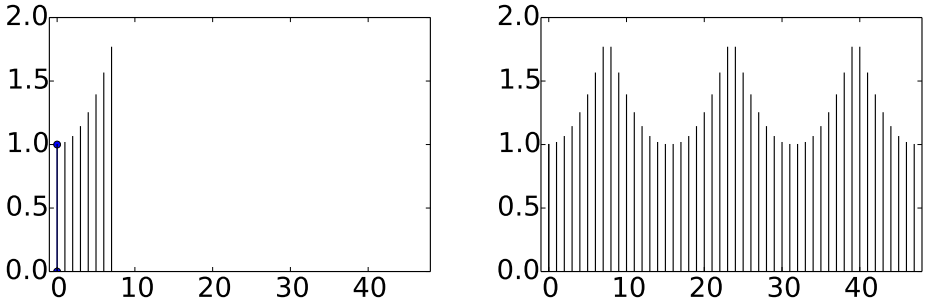

Figure 4.1: A vector and its symmetric extension.

We will call this the symmetric extension of $\boldsymbol{x}$:

Definition 4.1. *Symmetric extension of a vector.*

By the *symmetric extension* of $\boldsymbol{x} \in \mathbb{R}^N$, we mean the symmetric vector $\breve{\boldsymbol{x}} \in \mathbb{R}^{2N}$ defined by

$$\breve{\boldsymbol{x}}_k = \begin{cases} x_k & 0 \leq k < N \\ x_{2N-1-k} & N \leq k < 2N-1 \end{cases} \tag{4.1}$$

Clearly, the symmetric extension is symmetric around $N - 1/2$. This is not the only way to construct a symmetric extension, as we will return to later. As shown in Figure 4.1, but not included in Definition 4.1, we also repeat $\breve{\boldsymbol{x}} \in \mathbb{R}^{2N}$ in order to obtain a periodic vector. Creating a symmetric extension is thus a two-step process:

- First, "mirror" the vector to obtain a vector in $\mathbb{R}^{2N}$,
- repeat this periodically to obtain a periodic vector.

The result from the first step lies in an N-dimensional subspace of all vectors in $\mathbb{R}^{2N}$, which we will call *the space of symmetric vectors*. To account for the fact that a periodic vector can have a different symmetry point than $N - 1/2$, let us make the following general definition:

Definition 4.2. *Symmetric vector.*

We say that a periodic vector $\boldsymbol{x}$ is *symmetric* if there exists a number d so that $x_{d+k} = x_{d-k}$ for all k so that $d + k$ and $d - k$ are integers. d is called the *symmetry point* of $\boldsymbol{x}$

Due to the inherent periodicity of $\boldsymbol{x}$, it is clear that N must be an even number for symmetric vectors to exist at all. d can take any value, and it may not be an integer: It can also be an odd multiple of $1/2$, because then both $d+k$ and $d-k$ are integers when k also is an odd multiple of $1/2$. The symmetry point in symmetric extensions as defined in Definition 4.1 was $d = N - 1/2$. This is very common in the literature, and this is why we concentrate on this in this chapter. Later we will also consider symmetry around $N-1$, as this also is much used.

We would like to find a basis for the N-dimensional space of symmetric vectors, and we would like this basis to be similar to the Fourier basis. Since the Fourier basis corresponds to the standard basis in the frequency domain, we are lead to studying the DFT of a symmetric vector. If the symmetry point is an integer, it is straightforward to prove the following:

Theorem 4.3. *Symmetric vectors with integer symmetry points.*
Let d be an integer. The following are equivalent

- $\boldsymbol{x}$ is real and symmetric with d as symmetry point.
- $(\widehat{\boldsymbol{x}})_n = z_n e^{-2\pi i d n/N}$ where z_n are real numbers so that $z_n = z_{N-n}$.

Proof. Assume first that $d = 0$. It follows in this case from property 2(a) of Theorem 2.18 that $(\widehat{\boldsymbol{x}})_n$ is a real vector. Combining this with property 1 of Theorem 2.18 we see that $\widehat{\boldsymbol{x}}$, just as $\boldsymbol{x}$, also must be a real vector symmetric about 0. Since the DFT is one-to-one, it follows that $\boldsymbol{x}$ is real and symmetric about 0 if and only if $\widehat{\boldsymbol{x}}$ is. From property 3 of Theorem 2.18it follows that, when d is an integer, $\boldsymbol{x}$ is real and symmetric about d if and only if $(\widehat{\boldsymbol{x}})_n = z_n e^{-2\pi i d n/N}$, where z_n is real and symmetric about 0. This completes the proof. □

Symmetric extensions were here defined by having the non-integer symmetry point $N - 1/2$, however. For these we prove the following, which is slightly more difficult.

Theorem 4.4. *Symmetric vectors with non-integer symmetry points.*
Let d be an odd multiple of $1/2$. The following are equivalent

- $\boldsymbol{x}$ is real and symmetric with d as symmetry point.
- $(\widehat{\boldsymbol{x}})_n = z_n e^{-2\pi i d n/N}$ where z_n are real numbers so that $z_{N-n} = -z_n$.

Proof. When $\boldsymbol{x}$ is as stated we can write

$$\begin{aligned}
(\widehat{\boldsymbol{x}})_n &= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} x_k e^{-2\pi i k n/N} \\
&= \frac{1}{\sqrt{N}} \left(\sum_{s\geq 0} x_{d+s} e^{-2\pi i (d+s) n/N} + \sum_{s\geq 0} x_{d-s} e^{-2\pi i (d-s) n/N} \right) \\
&= \frac{1}{\sqrt{N}} \sum_{s\geq 0} x_{d+s} \left(e^{-2\pi i (d+s) n/N} + e^{-2\pi i (d-s) n/N} \right) \\
&= \frac{1}{\sqrt{N}} e^{-2\pi i d n/N} \sum_{s\geq 0} x_{d+s} \left(e^{-2\pi i s n/N} + e^{2\pi i s n/N} \right) \\
&= \frac{1}{\sqrt{N}} e^{-2\pi i d n/N} \sum_{s\geq 0} 2x_{d+s} \cos(2\pi s n/N).
\end{aligned}$$

Here s runs through odd multiples of $1/2$. Since $z_n = \frac{1}{\sqrt{N}} \sum_{s\geq 0} 2x_{d+s} \cos(2\pi s n/N)$ is a real number, we can write the result as $z_n e^{-2\pi i d n/N}$. Substituting $N - n$ for n, we get

$$\begin{aligned}
(\widehat{\boldsymbol{x}})_{N-n} &= \frac{1}{\sqrt{N}} e^{-2\pi i d (N-n)/N} \sum_{s\geq 0} 2x_{d+s} \cos(2\pi s (N-n)/N) \\
&= \frac{1}{\sqrt{N}} e^{-2\pi i d (N-n)/N} \sum_{s\geq 0} 2x_{d+s} \cos(-2\pi s n/N + 2\pi s) \\
&= -\frac{1}{\sqrt{N}} e^{-2\pi i d (N-n)/N} \sum_{s\geq 0} 2x_{d+s} \cos(2\pi s n/N) = -z_n e^{-2\pi i d (N-n)/N}.
\end{aligned}$$

This shows that $z_{N-n} = -z_n$, and this completes one way of the proof. The other way, we can write

$$x_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} (\widehat{\boldsymbol{x}})_n \, e^{2\pi i k n/N}$$

if $(\widehat{\boldsymbol{x}})_n = z_n e^{-2\pi i d n/N}$ and $(\widehat{\boldsymbol{x}})_{N-n} = -z_n e^{-2\pi i d (N-n)/N}$, the sum of the n'th term and the $N - n$'th term in the sum is

$$\begin{aligned}
&z_n e^{-2\pi i d n/N} e^{2\pi i k n/N} - z_n e^{2-\pi i d (N-n)/N} e^{2\pi i k (N-n)/N} \\
&= z_n (e^{2\pi i (k-d) n/N} - e^{-2\pi i d + 2\pi i d n/N - 2\pi i k n/N}) \\
&= z_n (e^{2\pi i (k-d) n/N} + e^{2\pi i (d-k) n/N}) = 2z_n \cos(2\pi (k-d) n/N).
\end{aligned}$$

This is real, so that all x_k are real. If we set $k = d + s$, $k = d - s$ here we get

$$2z_n \cos(2\pi((d+s)-d)n/N) = 2z_n \cos(2\pi sn/N)$$
$$2z_n \cos(2\pi((d-s)-d)n/N) = 2z_n \cos(-2\pi sn/N) = 2z_n \cos(2\pi sn/N).$$

By adding terms together and comparing we must have that $x_{d+s} = x_{d-s}$, and the proof is done. □

Now, let us specialize to symmetric extensions as defined in Definition 4.1, i.e. where $d = N - 1/2$. The following result gives us an orthonormal basis for the symmetric extensions, which are very simple in the frequency domain:

Theorem 4.5. *Orthonormal basis for symmetric vectors.*

The set of all $\boldsymbol{x}$ symmetric around $N - 1/2$ is a vector space of dimension N, and we have that

$$\left\{ \boldsymbol{e}_0, \left\{ \frac{1}{\sqrt{2}} \left(e^{\pi i n/(2N)} \boldsymbol{e}_n + e^{-\pi i n/(2N)} \boldsymbol{e}_{2N-n} \right) \right\}_{n=1}^{N-1} \right\}$$

is an orthonormal basis for $\widehat{\boldsymbol{x}}$ where $\boldsymbol{x}$ is symmetric around $N - 1/2$.

Proof. For a vector $\boldsymbol{x}$ symmetric about $d = N - 1/2$ we know that

$$(\widehat{\boldsymbol{x}})_n = z_n e^{-2\pi i(N-1/2)n/(2N)},$$

and the only requirement on the vector $\boldsymbol{z}$ is the antisymmetry condition $z_{2N-n} = -z_n$. The vectors $\boldsymbol{z}_i = \frac{1}{\sqrt{2}}(\boldsymbol{e}_i - \boldsymbol{e}_{2N-i})$, $1 \leq i \leq N-1$, together with the vector $\boldsymbol{z}_0 = \boldsymbol{e}_0$, are clearly orthonormal and satisifes the antisymmetry condition. From these we obtain that

$$\left\{ \boldsymbol{e}_0, \left\{ \frac{1}{\sqrt{2}} \left(e^{-2\pi i(N-1/2)n/(2N)} \boldsymbol{e}_n - e^{-2\pi i(N-1/2)(2N-n)/(2N)} \boldsymbol{e}_{2N-n} \right) \right\}_{n=1}^{N-1} \right\}$$

is an orthonormal basis for the $\widehat{\boldsymbol{x}}$ with $\boldsymbol{x}$ symmetric. We can write

$$\begin{aligned}
&\frac{1}{\sqrt{2}} \left(e^{-2\pi i(N-1/2)n/(2N)} \boldsymbol{e}_n - e^{-2\pi i(N-1/2)(2N-n)/(2N)} \boldsymbol{e}_{2N-n} \right) \\
&= \frac{1}{\sqrt{2}} \left(e^{-\pi i n} e^{\pi i n/(2N)} \boldsymbol{e}_n + e^{\pi i n} e^{-\pi i n/(2N)} \boldsymbol{e}_{2N-n} \right) \\
&= \frac{1}{\sqrt{2}} e^{\pi i n} \left(e^{\pi i n/(2N)} \boldsymbol{e}_n + e^{-\pi i n/(2N)} \boldsymbol{e}_{2N-n} \right).
\end{aligned}$$

This also means that

$$\left\{ \boldsymbol{e}_0, \left\{ \frac{1}{\sqrt{2}} \left(e^{\pi i n/(2N)} \boldsymbol{e}_n + e^{-\pi i n/(2N)} \boldsymbol{e}_{2N-n} \right) \right\}_{n=1}^{N-1} \right\}$$

is an orthonormal basis. □

We immediately get the following result:

Theorem 4.6. *Orthonormal basis for symmetric vectors.*
We have that

$$\left\{ \frac{1}{\sqrt{2N}} \cos\left(2\pi \frac{0}{2N}\left(k+\frac{1}{2}\right)\right), \left\{ \frac{1}{\sqrt{N}} \cos\left(2\pi \frac{n}{2N}\left(k+\frac{1}{2}\right)\right)\right\}_{n=1}^{N-1} \right\} \quad (4.2)$$

is an orthonormal basis for the set of vectors symmetric around $N-1/2$ in $\mathbb{R}^{2N}$. Moreover, the n'th vector in this basis has frequency contribution only from the indices n and $2N-n$.

Proof. Since the IDFT is unitary, the IDFT applied to the vectors above gives an orthonormal basis for the set of symmetric extensions. We get that

$$(F_{2N})^H(\boldsymbol{e}_0) = \left(\frac{1}{\sqrt{2N}}, \frac{1}{\sqrt{2N}}, \ldots, \frac{1}{\sqrt{2N}}\right) = \frac{1}{\sqrt{2N}} \cos\left(2\pi \frac{0}{2N}\left(k+\frac{1}{2}\right)\right).$$

We also get that

$$\begin{aligned}
&(F_{2N})^H \left(\frac{1}{\sqrt{2}}\left(e^{\pi i n/(2N)}\boldsymbol{e}_n + e^{-\pi i n/(2N)}\boldsymbol{e}_{2N-n}\right)\right) \\
&= \frac{1}{\sqrt{2}}\left(e^{\pi i n/(2N)}\frac{1}{\sqrt{2N}}e^{2\pi i n k/(2N)} + e^{-\pi i n/(2N)}\frac{1}{\sqrt{2N}}e^{2\pi i (2N-n)k/(2N)}\right) \\
&= \frac{1}{\sqrt{2}}\left(e^{\pi i n/(2N)}\frac{1}{\sqrt{2N}}e^{2\pi i n k/(2N)} + e^{-\pi i n/(2N)}\frac{1}{\sqrt{2N}}e^{-2\pi i n k/(2N)}\right) \\
&= \frac{1}{2\sqrt{N}}\left(e^{2\pi i (n/(2N))(k+1/2)} + e^{-2\pi i (n/(2N))(k+1/2)}\right) = \frac{1}{\sqrt{N}} \cos\left(2\pi \frac{n}{2N}\left(k+\frac{1}{2}\right)\right).
\end{aligned}$$

Since F_{2N} is unitary, and thus preserves the scalar product, the given vectors are orthonormal. □

We need to address one final thing before we can define the DCT: The vector $\boldsymbol{x}$ we start with is in $\mathbb{R}^N$, but the vectors above are in $\mathbb{R}^{2N}$. We would like to have orthonormal vectors in $\mathbb{R}^N$, so that we can use them to decompose $\boldsymbol{x}$. It is possible to show with a direct argument that, when we restrict the vectors above to the first N elements, they are still orthogonal. We will, however, apply a more instructive argument to show this, which gives us some intuition into the connection with symmetric filters. We start with the following result, which shows that a filter preserves symmetric vectors if and only if the filter is symmetric.

Theorem 4.7. *Criteria for preserving symmetric vectors.*
Let S be a filter. The following are equivalent

- S preserves symmetric vectors (i.e. $S\boldsymbol{x}$ is a symmetric vector whenever $\boldsymbol{x}$ is).
- The set of filter coefficients of S is a symmetric vector.

Also, when S preserves symmetric vectors, the following hold:

- The vector of filter coefficients has an integer symmetry point if and only if the input and output have the same type (integer or non-integer) of symmetry point.
- The input and output have the same symmetry point if and only if the filter is symmetric.

Proof. Assume that the filter S maps a symmetric vector with symmetry at d_1 to another symmetric vector. Let $\boldsymbol{x}$ be the symmetric vector so that $(\widehat{\boldsymbol{x}})_n = e^{-2\pi i d_1 n/N}$ for $n < N/2$. Since the output is a symmetric vector, we must have that

$$\lambda_{S,n} e^{-2\pi i d_1 n/N} = z_n e^{-2\pi i d_2 n/N}$$

for some d_2, z_n and for $n < N/2$. But this means that $\lambda_{S,n} = y_n e^{-2\pi i (d_2 - d_1) n/N}$. Similar reasoning applies for $n > N/2$, so that $\lambda_{S,n}$ clearly equals $\widehat{s}$ for some symmetric vector $\boldsymbol{s}$ from Theorems 4.3 and 4.4. This vector equals (up to multiplication with $\sqrt{N}$) the filter coefficients of S, which therefore is a symmetric. Moreover, it is clear that the filter coefficients have an integer symmetry point if and only if the input and output vector either both have an integer symmetry point, or both a non-integer symmetry point. □

Since the filter coefficients of a filter which preserves symmetric vectors also is a symmetric vector, this means that its frequency response takes the form $\lambda_{S,n} = z_n e^{-2\pi i d n/N}$, where $\boldsymbol{z}$ is a real vector. This means that the phase (argument) of the freqency response is $-2\pi dn/N$ or $\pi - 2\pi dn/N$, depending on the sign of z_n. In other words, the phase is linear in n. Filters which preserve symmetric vectors are therefore also called *linear phase filters*

.

Note also that the case $d = 0$ or $d = N - 1/2$ corresponds to symmetric filters. An example of linear phase filters which are not symmetric are smoothing filters where the coefficients are taken from odd rows in Pascal's triangle.

When S is symmetric, it preserves symmetric extensions, so that it makes sense to restrict S to symmetric vectors. We therefore make the following definition.

Definition 4.8. *Symmetric restriction.*

Assume that $S : \mathbb{R}^{2N} \to \mathbb{R}^{2N}$ is a symmetric filter. We define $S_r : \mathbb{R}^N \to \mathbb{R}^N$ as the mapping which sends $\boldsymbol{x} \in \mathbb{R}^N$ to the first N components of the vector $S\breve{\boldsymbol{x}}$. S_r is also called the symmetric restriction of S.

S_r is clearly linear, and the restriction of S to vectors symmetric about $N-1/2$ is characterized by S_r. We continue with the following result:

Theorem 4.9. *Expression for S_r.*
Assume that $S : \mathbb{R}^{2N} \to \mathbb{R}^{2N}$ is a symmetric filter, and that

$$S = \begin{pmatrix} S_1 & S_2 \\ S_3 & S_4 \end{pmatrix}.$$

Then S_r is symmetric, and $S_r = S_1 + (S_2)^f$, where $(S_2)^f$ is the matrix S_2 with the columns reversed.

Proof. With S as in the text of the theorem, we compute

$$S_r\boldsymbol{x} = \begin{pmatrix} S_1 & S_2 \end{pmatrix} \left(\begin{array}{c} x_0 \\ \vdots \\ x_{N-1} \\ \hline x_{N-1} \\ \vdots \\ x_0 \end{array}\right) = S_1 \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} + S_2 \begin{pmatrix} x_{N-1} \\ \vdots \\ x_0 \end{pmatrix}$$
$$= S_1 \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} + (S_2)^f \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} = (S_1 + (S_2)^f)\boldsymbol{x},$$

so that $S_r = S_1 + (S_2)^f$. Since S is symmetric, S_1 is also symmetric. $(S_2)^f$ is also symmetric, since it is constant on anti-diagonals. It follows then that S is also symmetric. This completes the proof. □

Note that S_r is not a digital filter, since its matrix is not circulant. In particular, its eigenvectors are not pure tones. In the block matrix factorization of S, S_2 contains the circulant part of the matrix, and forming $(S_2)^f$ means that the circulant parts switch corners. With the help of Theorem 4.9 we can finally establish the orthogonality of the cosine-vectors in $\mathbb{R}^N$.

Corollary 4.10. *Basis of eigenvectors for S_r.*
Let S be a symmetric filter, and let S_r be the mapping defined in Theorem 4.9. Define

$$d_{n,N} = \begin{cases} \sqrt{\frac{1}{N}} & , n = 0 \\ \sqrt{\frac{2}{N}} & , 1 \leq n < N \end{cases}$$

and $\boldsymbol{d}_n = d_{n,N} \cos\left(2\pi \frac{n}{2N}\left(k + \frac{1}{2}\right)\right)$ for $0 \leq n \leq N-1$, then $\{\boldsymbol{d}_0, \boldsymbol{d}_1, \ldots, \boldsymbol{d}_{N-1}\}$ is an orthonormal basis of eigenvectors for S_r.

Proof. Let S be a symmetric filter of length $2N$. We know then that $\lambda_{S,n} = \lambda_{S,2N-n}$, so that

$$\begin{aligned}
&S\left(\cos\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right)\right)\\
&= S\left(\frac{1}{2}\left(e^{2\pi i(n/(2N))(k+1/2)} + e^{-2\pi i(n/(2N))(k+1/2)}\right)\right)\\
&= \frac{1}{2}\left(e^{\pi in/(2N)}S\left(e^{2\pi ink/(2N)}\right) + e^{-\pi in/(2N)}S\left(e^{-2\pi ink/(2N)}\right)\right)\\
&= \frac{1}{2}\left(e^{\pi in/(2N)}\lambda_{S,n}e^{2\pi ink/(2N)} + e^{-\pi in/(2N)}\lambda_{S,2N-n}e^{-2\pi ink/(2N)}\right)\\
&= \frac{1}{2}\left(\lambda_{S,n}e^{2\pi i(n/(2N))(k+1/2)} + \lambda_{S,2N-n}e^{-2\pi i(n/(2N))(k+1/2)}\right)\\
&= \lambda_{S,n}\frac{1}{2}\left(e^{2\pi i(n/(2N))(k+1/2)} + e^{-2\pi i(n/(2N))(k+1/2)}\right)\\
&= \lambda_{S,n}\cos\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right),
\end{aligned}$$

where we have used that $e^{2\pi ink/(2N)}$ is an eigenvector of S with eigenvalue $\lambda_{S,n}$, and $e^{-2\pi ink/(2N)} = e^{2\pi i(2N-n)k/(2N)}$ is an eigenvector of S with eigenvalue $\lambda_{S,2N-n}$. This shows that the vectors are eigenvectors for symmetric filters of length $2N$. It is also clear that the first half of the vectors must be eigenvectors for S_r with the same eigenvalue, since when $\boldsymbol{y} = S\boldsymbol{x} = \lambda_{S,n}\boldsymbol{x}$, we also have that

$$(y_0, y_1, \ldots, y_{N-1}) = S_r(x_0, x_1, \ldots, x_{N-1}) = \lambda_{S,n}(x_0, x_1, \ldots, x_{N-1}).$$

To see why these vectors are orthogonal, choose at the outset a symmetric filter where $\{\lambda_{S,n}\}_{n=0}^{N-1}$ are distinct. Then the cosine-vectors of length N are also eigenvectors with distinct eigenvalues, and they must be orthogonal since S_r is symmetric. Moreover, since

$$
\begin{aligned}
&\sum_{k=0}^{2N-1} \cos^2\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) \\
&= \sum_{k=0}^{N-1} \cos^2\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) + \sum_{k=N}^{2N-1} \cos^2\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) \\
&= \sum_{k=0}^{N-1} \cos^2\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) + \sum_{k=0}^{N-1} \cos^2\left(2\pi\frac{n}{2N}\left(k+N+\frac{1}{2}\right)\right) \\
&= \sum_{k=0}^{N-1} \cos^2\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) + (-1)^{2n} \sum_{k=0}^{N-1} \cos^2\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) \\
&= 2\sum_{k=0}^{N-1} \cos^2\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right),
\end{aligned}
$$

where we used that $\cos(x+n\pi) = (-1)^n \cos x$. This means that

$$
\left\| \left\{ \cos\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) \right\}_{k=0}^{2N-1} \right\| = \sqrt{2} \left\| \left\{ \cos\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) \right\}_{k=0}^{N-1} \right\|.
$$

Thus, in order to make the vectors orthonormal when we consider the first N elements instead of all $2N$ elements, we need to multiply with $\sqrt{2}$. This gives us the vectors $\boldsymbol{d}_n$ as defined in the text of the theorem. This completes the proof. □

We now clearly see the analogy between symmetric functions and vectors: while the first can be written as a sum of cosine-functions, the second can be written as a sum of cosine-vectors. The orthogonal basis we have found is given its own name:

Definition 4.11. *DCT basis.*
We denote by $\mathcal{D}_N$ the orthogonal basis $\{\boldsymbol{d}_0, \boldsymbol{d}_1, \ldots, \boldsymbol{d}_{N-1}\}$. We also call $\mathcal{D}_N$ the N-point *DCT* basis.

Using the DCT basis instead of the Fourier basis we can make the following definitions, which parallel those for the DFT:

Definition 4.12. *Discrete Cosine Transform.*
The change of coordinates from the standard basis of $\mathbb{R}^N$ to the DCT basis $\mathcal{D}_N$ is called the *discrete cosine transform* (or DCT). The $N \times N$ matrix DCT_N that represents this change of basis is called the (N-point) DCT matrix. If $\boldsymbol{x}$ is a vector in R^N, its coordinates $\boldsymbol{y} = (y_0, y_1, \ldots, y_{N-1})$ relative to the DCT basis are called the DCT coefficients of $\boldsymbol{x}$ (in other words, $\boldsymbol{y} = \mathrm{DCT}_N \boldsymbol{x}$).

Note that we can also write

$$\mathrm{DCT}_N = \sqrt{\frac{2}{N}} \begin{pmatrix} 1/\sqrt{2} & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \left(\cos\left(2\pi \tfrac{n}{2N}(k+1/2)\right)\right). \tag{4.3}$$

Since this matrix is orthogonal, it is immediate that

$$\left(\cos\left(2\pi \tfrac{n}{2N}(k+1/2)\right)\right)^{-1} = \frac{2}{N}\left(\cos\left(2\pi \tfrac{n+1/2}{2N}k\right)\right) \begin{pmatrix} 1/2 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \tag{4.4}$$

$$\left(\cos\left(2\pi \tfrac{n+1/2}{2N}k\right)\right)^{-1} = \frac{2}{N} \begin{pmatrix} 1/2 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \left(\cos\left(2\pi \tfrac{n}{2N}(k+1/2)\right)\right). \tag{4.5}$$

In other words, not only can DCT_N be directly expressed in terms of a cosine-matrix, but our developments helped us to express the inverse of a cosine matrix in terms of other cosine-matrices. In the literature different types of cosine-matrices have been useful:

I Cosine-matrices with entries $\cos(2\pi nk/(2(N-1)))$.

II Cosine-matrices with entries $\cos(2\pi n(k+1/2)/(2N))$.

III Cosine-matrices with entries $\cos(2\pi(n+1/2)k/(2N))$.

IV Cosine-matrices with entries $\cos(2\pi(n+1/2)(k+1/2)/(2N))$.

We will call these type-I, type-II, type-III, and type-IV cosine-matrices, respectively. What we did above handles the case of type-II cosine-matrices. It will turn out that not all of these cosine-matrices are orthogonal, but that we in all cases, as we did above for type-II cosine matrices, can express the inverse of a cosine-matrix of one type in terms of a cosine-matrix of another type, and that any cosine-matrix is easily expressed in terms of an orthogonal matrix. These orthogonal matrices will be called $\mathrm{DCT}_N^{(I)}$, $\mathrm{DCT}_N^{(II)}$, $\mathrm{DCT}_N^{(III)}$, and $\mathrm{DCT}_N^{(IV)}$, respectively, and they are all called DCT-matrices. The DCT_N we constructed abobe is thus $\mathrm{DCT}_N^{(II)}$. The type-II DCT matrix is the most commonly used, and the type is therefore often dropped when refering to these. We will consider the other cases of cosine-matrices at different places in this book: In Section 5.6 we will run into type-I cosine matrices, in connection with a different extension

strategy used for wavelets. Type-IV cosine-matrices will be encountered in exercises 4.4 and 4.5 at the end of this section.

As with the Fourier basis vectors, the DCT basis vectors are called synthesis vectors, since we can write

$$\boldsymbol{x} = y_0\boldsymbol{d}_0 + y_1\boldsymbol{d}_1 + \cdots + y_{N-1}\boldsymbol{d}_{N-1} \tag{4.6}$$

in the same way as for the DFT. Following the same reasoning as for the DFT, DCT_N^{-1} is the matrix where the $\boldsymbol{d}_n$ are columns. But since these vectors are real and orthonormal, DCT_N must be the matrix where the $\boldsymbol{d}_n$ are rows. Moreover, since Theorem 4.9 also states that the same vectors are eigenvectors for filters which preserve symmetric extensions, we can state the following:

Theorem 4.13. *The DCT is orthogonal.*

DCT_N is the orthogonal matrix where the rows are $\boldsymbol{d}_n$. Moreover, for any digital filter S which preserves symmetric extensions, $(\text{DCT}_N)^T$ diagonalizes S_r, i.e. $S_r = \text{DCT}_N^T D \text{DCT}_N$ where D is a diagonal matrix.

Let us also make the following definition:

Definition 4.14. *IDCT.*

We will call $\boldsymbol{x} = (\text{DCT}_N)^T\boldsymbol{y}$ the inverse DCT or (IDCT) of $\boldsymbol{x}$.

Example 4.15. *Computing lower order DCTs.*

As with Example 2.16, exact expressions for the DCT can be written down just for a few specific cases. It turns out that the case $N = 4$ as considered in Example 2.16 does not give the same type of nice, exact values, so let us instead consider the case $N = 2$. We have that

$$\text{DCT}_4 = \begin{pmatrix} \frac{1}{\sqrt{2}}\cos(0) & \frac{1}{\sqrt{2}}\cos(0) \\ \cos\left(\frac{\pi}{2}\left(0+\frac{1}{2}\right)\right) & \cos\left(\frac{\pi}{2}\left(1+\frac{1}{2}\right)\right) \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

The DCT of the same vector as in Example 2.16 can now be computed as:

$$\text{DCT}_2 \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} \frac{3}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix}.$$

Matlab's functions for computing the DCT and IDCT are called `dct`, and `idct`, respectively. These are defined exactly as they are here, contrary to the case for the FFT (where a different normalizing factor was used).

With these functions we can repeat examples 2.27- 2.29, by simply replacing the calls to `DFTImpl` with calls to the DCT counterparts. You may not here much improvements in these simple experiments, but in theory the DCT should be able to approximate sound better.

Similarly to the DFT, one can think of the DCT as a least squares approximation and the unique representation of a function having the same sample values, but this time in terms of sinusoids instead of complex exponentials:

Theorem 4.16. *Interpolation with the DCT basis.*

Let f be a function defined on the interval $[0, T]$, and let $\boldsymbol{x}$ be the sampled vector given by

$$x_k = f((2k+1)T/(2N)) \quad \text{for } k = 0,\, 1,\, \ldots,\, N-1.$$

There is exactly one linear combination $g(t)$ on the form

$$\sum_{n=0}^{N-1} y_n d_{n,N} \cos(2\pi(n/2)t/T)$$

which satisfies the conditions

$$g((2k+1)T/(2N)) = f((2k+1)T/(2N)), \quad k = 0,\, 1,\, \ldots,\, N-1,$$

and its coefficients are determined by $\boldsymbol{y} = \mathrm{DCT}_N \boldsymbol{x}$.

Proof. This follows by inserting $t = (2k+1)T/(2N)$ in the equation

$$g(t) = \sum_{n=0}^{N-1} y_n d_{n,N} \cos(2\pi(n/2)t/T)$$

to arrive at the equations

$$f(kT/N) = \sum_{n=0}^{N-1} y_n d_{n,N} \cos\left(2\pi \frac{n}{2N}\left(k + \frac{1}{2}\right)\right) \qquad 0 \le k \le N-1.$$

This gives us an equation system for finding the y_n with the invertible DCT matrix as coefficient matrix, and the result follows. □

Due to this there is a slight difference to how we applied the DFT, due to the subtle change in the sample points, from kT/N for the DFT, to $(2k+1)T/(2N)$ for the DCT. The sample points for the DCT are thus the midpoints on the intervals in a uniform partition of $[0, T]$ into N intervals, while they for the DFT are the start points on the intervals. Also, the frequencies are divided by 2. In Figure 4.2 we have plotted the sinusoids of Theorem 4.16 for $T = 1$, as well as the sample points used in that theorem.

The sample points in the upper left plot correspond to the first column in the DCT matrix, the sample points in the upper right plot to the second column of the DCT matrix, and so on (up to normalization with $d_{n,N}$). As n increases, the functions oscillate more and more. As an example, y_5 says how much content of maximum oscillation there is. In other words, the DCT of an audio signal shows the proportion of the different frequencies in the signal, and the two formulas $\boldsymbol{y} = \mathrm{DCT}_N \boldsymbol{x}$ and $\boldsymbol{x} = (\mathrm{DCT}_N)^T \boldsymbol{y}$ allow us to switch back and forth between the time domain representation and the frequency domain representation of the sound. In other words, once we have computed $\boldsymbol{y} = \mathrm{DCT}_N \boldsymbol{x}$, we can analyse

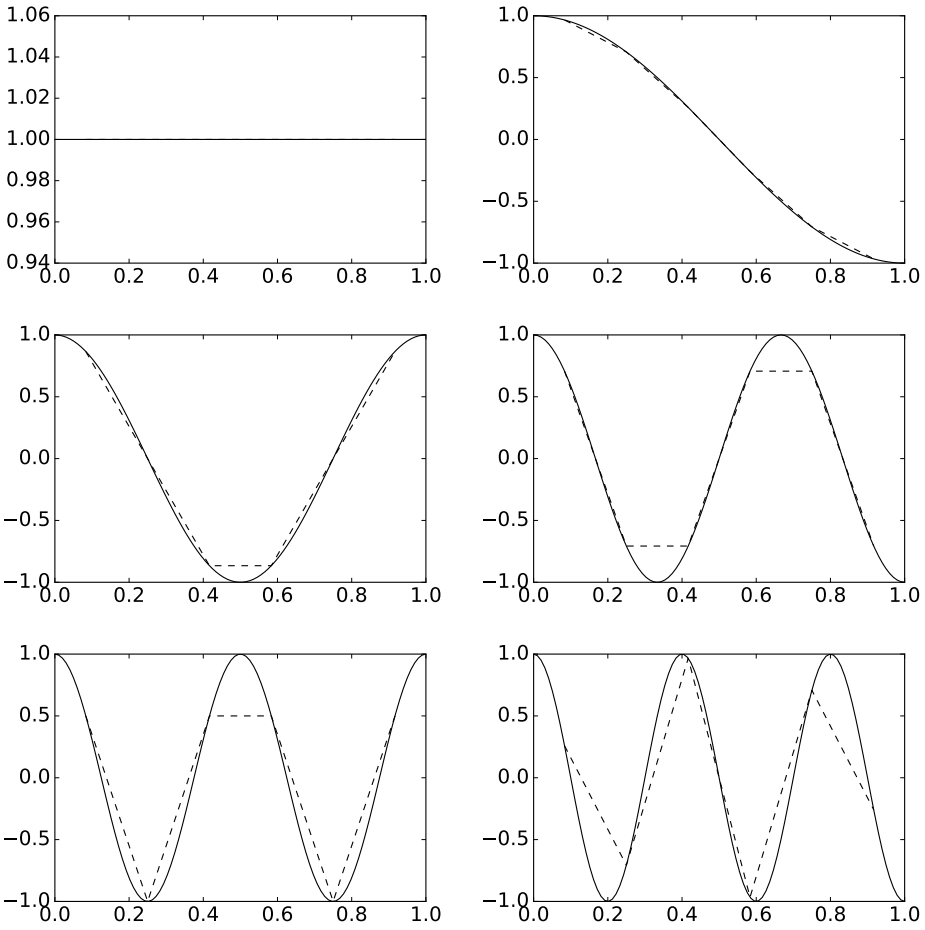

Figure 4.2: The 6 different sinusoids used in DCT for $N = 6$, i.e. $\cos(2\pi(n/2)t)$, $0 \leq n < 6$. The plots also show piecewise linear functions (in red) between the sample points $\frac{2k+1}{2N}$ $0 \leq k < 6$, since only the values at these points are used in Theorem 4.16.

the frequency content of $\boldsymbol{x}$. If we want to reduce the bass we can decrease the $\boldsymbol{y}$-values with small indices and if we want to increase the treble we can increase the $\boldsymbol{y}$-values with large indices.

Exercise 4.1: Computing eigenvalues

Consider the matrix

$$S = \frac{1}{3}\begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix}$$

a) Compute the eigenvalues and eigenvectors of S using the results of this section. You should only need to perform one DFT or one DCT in order to achieve this.

b) Use a computer to compute the eigenvectors and eigenvalues of S also. What are the differences from what you found in a)?

c) Find a filter T so that $S = T_r$. What kind of filter is T?

Exercise 4.2: Writing down lower order S_r

Consider the averaging filter $S = \{\frac{1}{4}, \underline{\frac{1}{2}}, \frac{1}{4}\}$. Write down the matrix S_r for the case when $N = 4$.

Exercise 4.3: Writing down lower order DCTs

As in Example 4.15, state the exact cartesian form of the DCT matrix for the case $N = 3$.

Exercise 4.4: DCT-IV

Show that the vectors $\left\{\cos\left(2\pi\frac{n+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)\right\}_{n=0}^{N-1}$ in $\mathbb{R}^N$ are orthogonal, with lengths $\sqrt{N/2}$. This means that the matrix with entries $\sqrt{\frac{2}{N}}\cos\left(2\pi\frac{n+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)$ is orthogonal. Since this matrix also is symmetric, it is its own inverse. This is the DCT-IV, which we denote by $\mathrm{DCT}_N^{(\mathrm{IV})}$. Although we will not consider this, the DCT-IV also has an efficient implementation.

Hint. Compare with the orthogonal vectors $\boldsymbol{d}_n$, used in the DCT.

Exercise 4.5: MDCT

The MDCT is defined as the $N\times(2N)$-matrix M with elements $M_{n,k} = \cos(2\pi(n+1/2)(k+1/2+N/2)/(2N))$. This exercise will take you through the details of the transformation which corresponds to multiplication with this matrix. The MDCT is very useful, and is also used in the MP3 standard and in more recent standards.

a) Show that

$$M = \sqrt{\frac{N}{2}}\mathrm{DCT}_N^{(\mathrm{IV})} \begin{pmatrix} \mathbf{0} & A \\ B & \mathbf{0} \end{pmatrix}$$

where A and B are the $(N/2) \times N$-matrices

$$A = \begin{pmatrix} \cdots & \cdots & 0 & -1 & -1 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & -1 & \cdots & \cdots & \cdots & \cdots & -1 & 0 \\ -1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 & -1 \end{pmatrix} = \begin{pmatrix} -I_{N/2}^f & -I_{N/2} \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 & -1 \\ 0 & 1 & \cdots & \cdots & \cdots & \cdots & -1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & 0 & 1 & -1 & 0 & \cdots & \cdots \end{pmatrix} = \begin{pmatrix} I_{N/2} & -I_{N/2}^f \end{pmatrix}.$$

Due to this expression, any algorihtm for the DCT-IV can be used to compute the MDCT.

b) The MDCT is not invertible, since it is not a square matrix. We will show here that it still can be used in connection with invertible transformations. We first define the IMDCT as the matrix M^T/N. Transposing the matrix expression we obtained in a) gives

$$\frac{1}{\sqrt{2N}} \begin{pmatrix} \mathbf{0} & B^T \\ A^T & \mathbf{0} \end{pmatrix} \mathrm{DCT}_N^{(\mathrm{IV})}$$

for the IMDCT, which thus also has an efficient implementation. Show that if

$$\boldsymbol{x}_0 = (x_0, \ldots, x_{N-1}) \qquad \boldsymbol{x}_1 = (x_N, \ldots, x_{2N-1}) \qquad \boldsymbol{x}_2 = (x_{2N}, \ldots, x_{3N-1})$$

and

$$\boldsymbol{y_{0,1}} = M \begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \end{pmatrix} \qquad\qquad \boldsymbol{y_{1,2}} = M \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix}$$

(i.e. we compute two MDCT's where half of the data overlap), then

$$\boldsymbol{x}_1 = \{\mathrm{IMDCT}(\boldsymbol{y_{0,1}})\}_{k=N}^{2N-1} + \{\mathrm{IMDCT}(\boldsymbol{y_{1,2}})\}_{k=0}^{N-1}.$$

Even though the MDCT itself is not invertible, the input can still be recovered from overlapping MDCT's.

4.2 Improvements from using the DCT to interpolate functions and approximate analog filters

Recall that, in Section 3.2.1, we explained how to approximate an analog filter from the samples. It turns out that, when an analog filter is symmetric, we can use symmetric extensions to create a better approximation from the samples.

Assume that s is an analog filter, and that we apply it to a general function f. Denote as before the symmetric extension of f by $\breve{f}$. We start with the following observation, which follows from the continuity of s.

Observation 4.17. *Using symmetric extensions for approximations.*
Since $(\breve{f})_N$ is a better approximation to $\breve{f}$, compared to what f_N is to f, $s((\breve{f})_N)$ is a better approximation to $s(\breve{f})$, compared to what $s(f_N)$ is to $s(f)$.

Since $s(\breve{f})$ agrees with $s(f)$ except near the boundaries, we can thus conclude that $s((\breve{f})_N)$ is a better approximation to $s(f)$ than what $s(f_N)$ is.

We have seen that the restriction of s to $V_{M,T}$ is equivalent to an $N \times N$ digital filter S, where $N = 2M + 1$. Let $\boldsymbol{x}$ be the samples of f, $\breve{\boldsymbol{x}}$ the samples of $\breve{f}$. Turning around the fact that $(\breve{f})_N$ is a better approximation to $\breve{f}$, compared to what f_N is to f, the following is clear.

Observation 4.18. *Using symmetric extensions for approximations.*
The samples $\breve{\boldsymbol{x}}$ are a better approximation to the samples of $(\breve{f})_N$, than the samples $\boldsymbol{x}$ are to the samples of f_N.

Now, let $\boldsymbol{z} = S\boldsymbol{x}$, and $\breve{\boldsymbol{z}} = S\breve{\boldsymbol{x}}$. The following is also clear from the preceding observation, due to continuity of the digital filter S.

Observation 4.19. *Using symmetric extensions for approximations.*
$\breve{\boldsymbol{z}}$ is a better approximation to $S(\text{samples of } (\breve{f})_N) = \text{samples of } s((\breve{f})_N)$, than $\boldsymbol{z}$ is to $S(\text{samples of } f_N) = \text{samples of } s(f_N)$.

Since by Observation 4.17 $s((\breve{f})_N)$ is a better approximation to the output $s(f)$, we conclude that $\breve{\boldsymbol{z}}$ is a better approximation than $\boldsymbol{z}$ to the samples of the output of the filter.

Observation 4.20. *Using symmetric extensions for approximations.*
$S\breve{\boldsymbol{x}}$ is a better approximation to the samples of $s(f)$ than $S\boldsymbol{x}$ is ($\boldsymbol{x}$ are the samples of f).

Now, let us also bring in the assumption that s is symmetric. Then the corresponding digital filter S is also symmetric, and we know then that we can view its restriction to symmetric extensions in $\mathbb{R}^{2N}$ in terms of the mapping $S_r : \mathbb{R}^N \to \mathbb{R}^N$. We can thus specialize Figure 3.5 to symmetric filters by adding the step of creating the symmetric extension, and replacing S with S_r. We have summarized these remarks in Figure 4.3. The DCT here appears, since we have used Theorem 4.16 to interpolate with the DCT basis, instead of the Fourier

basis. Note that this also requires that the sampling is performed as required in that theorem, i.e. the samples are the midpoints on all intervals. This new sampling procedure is not indicated in Figure 4.3.

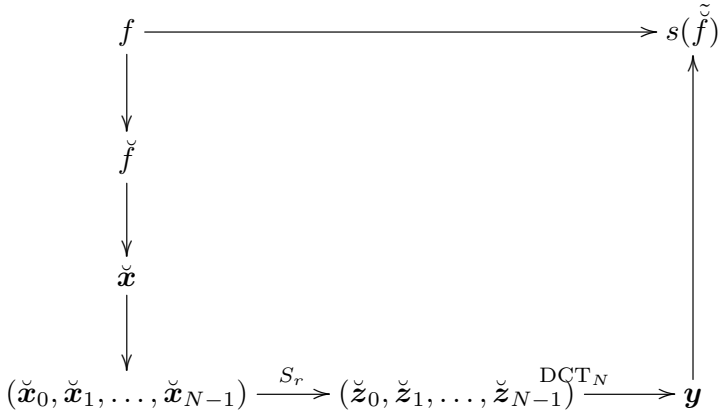

Figure 4.3: The connections between the new mapping S_r, sampling, and interpolation. The right vertical arrow represents interpolation with the DCT, i.e. that we compute $\sum_{n=0}^{N-1} y_n d_{n,N} \cos(2\pi(n/2)t/T)$ for values of t.

Figure 4.3 can be further simplified to that shown in Figure 4.4.

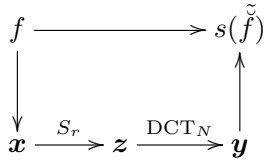

Figure 4.4: Simplification of Figure 4.3. The left vertical arrow represents sampling as dictated by the DCT.

Note that the assumption that s is symmetric only helped us to implement the approximation $s(\breve{\breve{f}})$ in a nore efficient way, since S_r has N points and S has $2N$ points. $s(\breve{\breve{f}})$ can in any way be used as an approximation, even if s is not symmetric. But this approximation is actually even better when s is symmetric: Since s is symmetric, $s(\breve{f})$ is a symmetric function (since $\breve{f}$ is a symmetric function). The N'th order Fourier series of $s(\breve{f})$ is $s((\breve{f})_N) = (s(\breve{f}))_N$, and this is a better approximation to $s(\breve{f})$ since $s(\breve{f})$ is a symmetric function. Since the procedure above obtained an approximation to (the samples of) $(s(\breve{f}))_N$, it follows that the approximations are better when s is symmetric.

As mentioned in Section 3.2, interpolation of a function from its samples can be seen as a special case. This can thus be illustrated as in Figure 4.5.

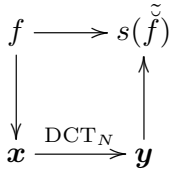

Figure 4.5: How we can approximate a function from its samples with the DCT.

Note that the approximation lies in $V_{2M,2T}$ (i.e. it is in a higher order Fourier space), but the point is that the same number of samples is used.

4.2.1 Implementations of symmetric filters

Symmetric filters are also important for applications since they can be implemented efficiently. To see this, we can write

$$\begin{aligned}
(S\boldsymbol{x})_n &= \sum_{k=0}^{N-1} s_k x_{(n-k) \bmod N} \\
&= s_0 x_n + \sum_{k=1}^{(N-1)/2} s_k x_{(n-k) \bmod N} + \sum_{k=(N+1)/2}^{N-1} s_k x_{(n-k) \bmod N} \\
&= s_0 x_n + \sum_{k=1}^{(N-1)/2} s_k x_{(n-k) \bmod N} + \sum_{k=1}^{(N-1)/2} s_k x_{(n-(N-k)) \bmod N} \\
&= s_0 x_n + \sum_{k=1}^{(N-1)/2} s_k (x_{(n-k) \bmod N} + x_{(n+k) \bmod N}). \qquad (4.7)
\end{aligned}$$

If we compare the first and last expressions here, we need the same number of summations, but the number of multiplications needed in the latter expression has been halved.

Observation 4.21. *Reducing arithmetic operations for symmetric filters.*

Assume that a symmetric filter has $2s+1$ filter coefficients. The filter applied to a vector of length N can then be implemented using $(s+1)N$ multiplications and $2sN$ additions. This gives a reduced number of arithmetic operations when compared to a filter with the same number of coefficients which is not symmetric, where a direct implementations requires $(2s+1)N$ multiplications and $2sN$ additions.

Similarly to as in Section 3.6.2, a symmetric filter can be factored into a product of symmetric filters. To see how, note first that a real polynomial is symmetric if and only if $1/a$ is a root whenever a is. If we pair together the

factors for the roots $a, 1/a$ when a is real we get a component in the frequency response of degree 2. If we pair the factors for the roots $a, 1/a, \overline{a}, \overline{1/a}$ when a is complex, we get a component in the frequency response of degree 4. We thus get the following idea:

Idea 4.22. *Factorizing symmetric filters.*

Let S be a symmetric filter with real coefficients. There exist constants K, $a_1, \ldots, a_m$, $b_1, c_1, \ldots, b_n, c_n$ so that

$$\begin{aligned}\lambda_S(\omega) =& K(a_1 e^{i\omega} + 1 + a_1 e^{-i\omega}) \ldots (a_m e^{i\omega} + 1 + a_m e^{-i\omega}) \\ & \times (b_1 e^{2i\omega} + c_1 e^{i\omega} + 1 + c_1 e^{-i\omega} + b_1 e^{-2i\omega}) \ldots \\ & \times (b_n e^{2i\omega} + c_n e^{i\omega} + 1 + c_n e^{-i\omega} + b_n e^{-2i\omega}).\end{aligned}$$

We can write $S = KA_1 \ldots A_m B_1 \ldots B_n$, where $A_i = \{a_i, \underline{1}, a_i\}$ and $B_i = \{b_i, c_i, \underline{1}, c_i, b_i\}$.

In any case we see that the component filters have 3 and 5 filter coefficients.

Exercise 4.6: Component expressions for a symmetric filter

Assume that $S = t_{-L}, \ldots, \underline{t_0}, \ldots, t_L$ is a symmetric filter. Use Equation (4.7) in the compendium to show that $z_n = (S\boldsymbol{x})_n$ in this case can be split into the following different formulas, depending on n:

a) $0 \leq n < L$:

$$z_n = t_0 x_n + \sum_{k=1}^{n} t_k (x_{n+k} + x_{n-k}) + \sum_{k=n+1}^{L} t_k (x_{n+k} + x_{n-k+N}). \tag{4.8}$$

b) $L \leq n < N - L$:

$$z_n = t_0 x_n + \sum_{k=1}^{L} t_k (x_{n+k} + x_{n-k}). \tag{4.9}$$

c) $N - L \leq n < N$:

$$z_n = t_0 x_n + \sum_{k=1}^{N-1-n} t_k (x_{n+k} + x_{n-k}) + \sum_{k=N-1-n+1}^{L} t_k (x_{n+k-N} + x_{n-k}). \tag{4.10}$$

The `convolve` function may not pick up this reduction in the number of multiplications, since it does not assume that the filter is symmetric. We will still use the `convolve` function in implementations, however, due to its heavy optimization.

4.3 Efficient implementations of the DCT

When we defined the DCT in the preceding section, we considered symmetric vectors of twice the length, and viewed these in the frequency domain. In order to have a fast algorithm for the DCT, which are comparable to the FFT algorithms we developed in Section 2.4, we need to address the fact that vectors of twice the length seem to be involved. The following theorem addresses this. This result is much used in practical implementations of DCT, and can also be used for practical implementation of the DFT as we will see in Exercise 4.8. Note that the result, and the following results in this section, are stated in terms of the cosine matrix C_N (where the entries are $(C_N)_{n,k} = \cos\left(2\pi \frac{n}{2N}\left(k+\frac{1}{2}\right)\right)$, rather than the DCT_N matrix (which uses the additional scaling factor $d_{n,N}$ for the rows). The reason is that C_N appears to me most practical for stating algorithms. When computing the DCT, we simply need to scale with the $d_{n,N}$ at the end, after using the statements below.

Theorem 4.23. *DCT algorithm.*
Let $\boldsymbol{y} = C_N \boldsymbol{x}$. Then we have that

$$y_n = \left(\cos\left(\pi \frac{n}{2N}\right) \Re((\mathrm{DFT}_N \boldsymbol{x}^{(1)})_n) + \sin\left(\pi \frac{n}{2N}\right) \Im((\mathrm{DFT}_N \boldsymbol{x}^{(1)})_n)\right), \quad (4.11)$$

where $\boldsymbol{x}^{(1)} \in \mathbb{R}^N$ is defined by

$$(\boldsymbol{x}^{(1)})_k = x_{2k} \text{ for } 0 \le k \le N/2-1$$
$$(\boldsymbol{x}^{(1)})_{N-k-1} = x_{2k+1} \text{ for } 0 \le k \le N/2-1,$$

Proof. Using the definition of C_N, ans splitting the computation of $\boldsymbol{y} = C_N \boldsymbol{x}$ into two sums, corresponding to the even and odd indices as follows:

$$\begin{aligned} y_n &= \sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{2N}\left(k+\frac{1}{2}\right)\right) \\ &= \sum_{k=0}^{N/2-1} x_{2k} \cos\left(2\pi \frac{n}{2N}\left(2k+\frac{1}{2}\right)\right) + \sum_{k=0}^{N/2-1} x_{2k+1} \cos\left(2\pi \frac{n}{2N}\left(2k+1+\frac{1}{2}\right)\right). \end{aligned}$$

If we reverse the indices in the second sum, this sum becomes

$$\sum_{k=0}^{N/2-1} x_{N-2k-1} \cos\left(2\pi \frac{n}{2N}\left(N-2k-1+\frac{1}{2}\right)\right).$$

If we then also shift the indices with $N/2$ in this sum, we get

$$\sum_{k=N/2}^{N-1} x_{2N-2k-1} \cos\left(2\pi \frac{n}{2N}\left(2N-2k-1+\frac{1}{2}\right)\right)$$
$$= \sum_{k=N/2}^{N-1} x_{2N-2k-1} \cos\left(2\pi \frac{n}{2N}\left(2k+\frac{1}{2}\right)\right),$$

where we used that cos is symmetric and periodic with period 2π. We see that we now have the same cos-terms in the two sums. If we thus define the vector $\boldsymbol{x}^{(1)}$ as in the text of the theorem, we see that we can write

$$\begin{aligned}
y_n &= \sum_{k=0}^{N-1} (\boldsymbol{x}^{(1)})_k \cos\left(2\pi \frac{n}{2N}\left(2k+\frac{1}{2}\right)\right) \\
&= \Re\left(\sum_{k=0}^{N-1} (\boldsymbol{x}^{(1)})_k e^{-2\pi i n\left(2k+\frac{1}{2}\right)/(2N)}\right) \\
&= \Re\left(e^{-\pi i n/(2N)} \sum_{k=0}^{N-1} (\boldsymbol{x}^{(1)})_k e^{-2\pi i n k/N}\right) \\
&= \Re\left(e^{-\pi i n/(2N)} (\mathrm{DFT}_N \boldsymbol{x}^{(1)})_n\right) \\
&= \left(\cos\left(\pi \frac{n}{2N}\right) \Re((\mathrm{DFT}_N \boldsymbol{x}^{(1)})_n) + \sin\left(\pi \frac{n}{2N}\right) \Im((\mathrm{DFT}_N \boldsymbol{x}^{(1)})_n)\right),
\end{aligned}$$

where we have recognized the N-point DFT. This completes the proof. □

With the result above we have avoided computing a DFT of double size. If we in the proof above define the $N \times N$-diagonal matrix Q_N by $Q_{n,n} = e^{-\pi i n/(2N)}$, the result can also be written on the more compact form

$$\boldsymbol{y} = C_N \boldsymbol{x} = \Re\left(Q_N \mathrm{DFT}_N \boldsymbol{x}^{(1)}\right).$$

We will, however, not use this form, since there is complex arithmetic involved, contrary to Equation(4.11). Code which uses Equation (4.11) to compute the DCT, using the function `FFTImpl` from Section 2.4, can look as follows:

```
def DCTImpl(x):
    """
    Compute the DCT of the vector x

    x: a vector
    """
    N = len(x)
    if N > 1:
        x1 = concatenate([x[0::2], x[-1:0:-2]]).astype(complex)
        FFTImpl(x1, FFTKernelStandard)
        cosvec = cos(pi*arange(float(N))/(2*N))
        sinvec = sin(pi*arange(float(N))/(2*N))
```

```
        if ndim(x) == 1:
            x[:] = cosvec*real(x1) + sinvec*imag(x1)
        else:
            for s2 in xrange(shape(x)[1]):
                x[:, s2] = cosvec*real(x1[:, s2]) \
                + sinvec*imag(x1[:, s2])
        x[0] *= sqrt(1/float(N))
        x[1:] *= sqrt(2/float(N))
```

In the code, the vector $\boldsymbol{x}^{(1)}$ is created first by rearranging the components, and it is sent as input to `FFTImpl`. After this we take real parts and imaginary parts, and multiply with the cos- and sin-terms in Equation (4.11).

4.3.1 Efficient implementations of the IDCT

As with the FFT, it is straightforward to modify the DCT implementation so that it returns the IDCT. To see how we can do this, write from Theorem 4.23, for $n \geq 1$

$$\begin{aligned} y_n &= \left(\cos\left(\pi\frac{n}{2N}\right)\Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) + \sin\left(\pi\frac{n}{2N}\right)\Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n)\right) \\ y_{N-n} &= \left(\cos\left(\pi\frac{N-n}{2N}\right)\Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_{N-n}) + \sin\left(\pi\frac{N-n}{2N}\right)\Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_{N-n})\right) \\ &= \left(\sin\left(\pi\frac{n}{2N}\right)\Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) - \cos\left(\pi\frac{n}{2N}\right)\Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n)\right), \end{aligned} \tag{4.12}$$

where we have used the symmetry of DFT_N for real signals. These two equations enable us to determine $\Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n)$ and $\Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n)$ from y_n and y_{N-n}. We get

$$\begin{aligned} \cos\left(\pi\frac{n}{2N}\right)y_n + \sin\left(\pi\frac{n}{2N}\right)y_{N-n} &= \Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) \\ \sin\left(\pi\frac{n}{2N}\right)y_n - \cos\left(\pi\frac{n}{2N}\right)y_{N-n} &= \Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n). \end{aligned}$$

Adding we get

$$\begin{aligned} (\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n =& \cos\left(\pi\frac{n}{2N}\right)y_n + \sin\left(\pi\frac{n}{2N}\right)y_{N-n} + i(\sin\left(\pi\frac{n}{2N}\right)y_n - \cos\left(\pi\frac{n}{2N}\right)y_{N-n}) \\ =& (\cos\left(\pi\frac{n}{2N}\right) + i\sin\left(\pi\frac{n}{2N}\right))(y_n - iy_{N-n}) = e^{\pi in/(2N)}(y_n - iy_{N-n}). \end{aligned}$$

This means that $(\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n = e^{\pi in/(2N)}(y_n + iy_{N-n}) = (y_n + iy_{N-n})/Q_{n,n}$ for $n \geq 1$. Since $\Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_0) = 0$ we have that $(\mathrm{DFT}_N\boldsymbol{x}^{(1)})_0 = \frac{1}{d_{0,N}}y_0 = y_0/Q_{0,0}$. This means that $\boldsymbol{x}^{(1)}$ can be recovered by taking the IDFT of the vector with component 0 being $y_0/Q_{0,0}$, and the remaining components being $(y_n - iy_{N-n})/Q_{n,n}$:

Theorem 4.24. *IDCT algorithm.*

Let $\boldsymbol{x} = (C_N)^{-1}\boldsymbol{y}$. and let $\boldsymbol{z}$ be the vector with component 0 being $y_0/Q_{0,0}$, and the remaining components being $(y_n - iy_{N-n})/Q_{n,n}$. Then we have that

$$\boldsymbol{x}^{(1)} = \mathrm{IDFT}_N \boldsymbol{z},$$

where $\boldsymbol{x}^{(1)}$ is defined as in Theorem 4.23.

The implementation of IDCT can thus go as follows:

```
def IDCTImpl(y):
    """
    Compute the IDCT of the vector y

    y: a vector
    """
    N = len(y)
    if N > 1:
        y[0] /= sqrt(1/float(N))
        y[1:] /= sqrt(2/float(N))
        Q = exp(-pi*1j*arange(float(N))/(2*N))
        y1 = zeros_like(y).astype(complex)
        y1[0] = y[0]/Q[0]
        if ndim(y) == 1:
            y1[1:] = (y[1:] - 1j*y[-1:0:-1])/Q[1:]
        else:
            for s2 in xrange(shape(y)[1]):
                y1[1:, s2] = (y[1:, s2] - 1j*y[-1:0:-1, s2])/Q[1:]
        FFTImpl(y1, FFTKernelStandard, 0)
        y[0::2] = real(y1[0:(N/2)])
        y[1::2] = real(y1[-1:(N/2-1):-1])
```

4.3.2 Reduction in the number of multiplications with the DCT

Let us also state a result which confirms that the DCT and IDCT implementations we have described give the same type of reductions in the number multiplications as the FFT and IFFT:

Theorem 4.25. *Number of multiplications required by the DCT and IDCT algorithms.*

The DCT and the IDCT can be implemented so that they use any FFT and IFFT algorithms. Their operation counts then have the same order as these. In particular, when the standard FFT algorithms of Section 2.4 are used, i.e. their operation counts are $O(5N\log_2/2)$. In comparison, the operation count for a direct implementation of the N-point DCT/IDCT is $2N^2$.

Note that we divide the previous operation counts by 2 since the DCT applies an FFT to real input only, and the operation count for the FFT can be halved when we adapt to real data, see Exercise 2.27.

Proof. By Theorem 2.36, the number of multiplications required by the standard FFT algorithm from Section 2.4 adapted to real data is $O(N\log_2 N)$, while

the number of additions is $O(3N \log_2 N/2)$. By Theorem 4.23, two additional multiplications and one addition are required for each index (so that we have $2N$ extra real multiplications and N extra real additions in total), but this does not affect the operation count, since $O(N \log_2 N + 2N) = O(N \log_2 N)$. Since the operation counts for the IFFT is the same as for the FFT, we only need to count the additional multiplications needed in forming the vector $\boldsymbol{z} = (y_n - iy_{N-n})/Q_{n,n}$. Clearly, this also does not affect the order of the algorithm. □

Since the DCT and IDCT can be implemented using the FFT and IFFT, it has the same advantages as the FFT when it comes to parallel computing. Much literature is devoted to reducing the number of multiplications in the DFT and the DCT even further than what we have done (see [18] for one of the most recent developments). Another note on computational complexity is in order: we have not counted the operations sin and cos in the DCT. The reason is that these values can be precomputed, since we take the sine and cosine of a specific set of values for each DCT or DFT of a given size. This is contrary to to multiplication and addition, since these include the input values, which are only known at runtime. We have, however, not written down that we use precomputed arrays for sine and cosine in our algorithms: This is an issue to include in more optimized algorithms.

Exercise 4.7: Trick for reducing the number of multiplications with the DCT

In this exercise we will take a look at a small trick which reduces the number of additional multiplications we need for DCT algorithm from Theorem 4.23. This exercise does not reduce the order of the DCT algorithms, but we will see in Exercise 4.8 how the result can be used to achieve this.

a) Assume that $\boldsymbol{x}$ is a real signal. Equation (4.12) in the compendium, which said that

$$\begin{aligned} y_n &= \cos\left(\pi\frac{n}{2N}\right)\Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) + \sin\left(\pi\frac{n}{2N}\right)\Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) \\ y_{N-n} &= \sin\left(\pi\frac{n}{2N}\right)\Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) - \cos\left(\pi\frac{n}{2N}\right)\Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) \end{aligned}$$

for the n'th and $N-n$'th coefficient of the DCT. This can also be rewritten as

$$\begin{aligned} y_n &= \left(\Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) + \Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n)\right)\cos\left(\pi\frac{n}{2N}\right) \\ &\quad - \Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n)(\cos\left(\pi\frac{n}{2N}\right) - \sin\left(\pi\frac{n}{2N}\right)) \\ y_{N-n} &= -\left(\Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n) + \Im((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n)\right)\cos\left(\pi\frac{n}{2N}\right) \\ &\quad + \Re((\mathrm{DFT}_N\boldsymbol{x}^{(1)})_n)(\sin\left(\pi\frac{n}{2N}\right) + \cos\left(\pi\frac{n}{2N}\right)). \end{aligned}$$

Explain that the first two equations require 4 multiplications to compute y_n and y_{N-n}, and that the last two equations require 3 multiplications to compute y_n and y_{N-n}.

b) Explain why the trick in a) reduces the number of additional multiplications in a DCT, from $2N$ to $3N/2$.

c) Explain why the trick in a) can be used to reduce the number of additional multiplications in an IDCT with the same number.

Hint. match the expression $e^{\pi i n/(2N)}(y_n - iy_{N-n})$ you encountered in the IDCT with the rewriting you did in b.

d) Show that the penalty of the trick we here have used to reduce the number of multiplications, is an increase in the number of additional additions from N to $3N/2$. Why can this trick still be useful?

Exercise 4.8: An efficient joint implementation of the DCT and the FFT

In this exercise we will explain another joint implementation of the DFT and the DCT, which has the benefit of a low multiplication count, at the expense of a higher addition count. It also has the benefit that it is specialized to real vectors, with a very structured implementation (this is not always the case for the quickest FFT implementations. Not surprisingly, one often sacrifices clarity of code when one pursues higher computational speed). a) of this exercise can be skipped, as it is difficult and quite technical. For further details of the algorithm the reader is refered to [38].

a) Let $\boldsymbol{y} = \mathrm{DFT}_N \boldsymbol{x}$ be the N-point DFT of the real vector $\boldsymbol{x}$. Show that

$$\Re(y_n) = \begin{cases} \Re((\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) + (C_{N/4}\boldsymbol{z})_n & 0 \le n \le N/4-1 \\ \Re((\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) & n = N/4 \\ \Re((\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) - (C_{N/4}\boldsymbol{z})_{N/2-n} & N/4+1 \le n \le N/2-1 \end{cases} \tag{4.13}$$

$$\Im(y_n) = \begin{cases} \Im((\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) & n = 0 \\ \Im((\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) + (C_{N/4}\boldsymbol{w})_{N/4-n} & 1 \le n \le N/4-1 \\ \Im((\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) + (C_{N/4}\boldsymbol{w})_{n-N/4} & N/4 \le n \le N/2-1 \end{cases} \tag{4.14}$$

where $\boldsymbol{x}^{(e)}$ is as defined in Theorem 2.31, where $\boldsymbol{z}, \boldsymbol{w} \in \mathbb{R}^{N/4}$ defined by

$$\begin{aligned} z_k &= x_{2k+1} + x_{N-2k-1} && 0 \le k \le N/4-1, \\ w_k &= (-1)^k(x_{N-2k-1} - x_{2k+1}) && 0 \le k \le N/4-1, \end{aligned}$$

Explain from this how you can make an algorithm which reduces an FFT of length N to an FFT of length $N/2$ (on $\boldsymbol{x}^{(e)}$), and two DCT's of length $N/4$ (on $\boldsymbol{z}$ and $\boldsymbol{w}$). We will call this algorithm the revised FFT algorithm.

a) says nothing about the coefficients y_n for $n > \frac{N}{2}$. These are obtained in the same way as before through symmetry. a. also says nothing about $y_{N/2}$. This can be obtained with the same formula as in Theorem 2.31.

Let us now compute the number of arithmetic operations our revised algorithm needs. Denote by the number of real multiplications needed by the revised N-point FFT algorithm

b) Explain from the algorithm in a) that

$$M_N = 2(M_{N/4} + 3N/8) + M_{N/2} \quad A_N = 2(A_{N/4} + 3N/8) + A_{N/2} + 3N/2 \tag{4.15}$$

Hint. $3N/8$ should come from the extra additions/multiplications (see Exercise 4.7) you need to compute when you run the algorithm from Theorem 4.23 for $C_{N/4}$. Note also that the equations in a) require no extra multiplications, but that there are xix equations involved, each needing $N/4$ additions, so that we need $6N/4 = 3N/2$ extra additions.

c) Explain why $x_r = M_{2^r}$ is the solution to the difference equation

$$x_{r+2} - x_{r+1} - 2x_r = 3 \times 2^r,$$

and that $x_r = A_{2^r}$ is the solution to

$$x_{r+2} - x_{r+1} - 2x_r = 9 \times 2^r.$$

and show that the general solution to these are $x_r = \frac{1}{2}r2^r + C2^r + D(-1)^r$ for multiplications, and $x_r = \frac{3}{2}r2^r + C2^r + D(-1)^r$ for additions.

d) Explain why, regardless of initial conditions to the difference equations, $M_N = O\left(\frac{1}{2}N\log_2 N\right)$ and $A_N = O\left(\frac{3}{2}N\log_2 N\right)$ both for the revised FFT and the revised DCT. The total number of operations is thus $O(2N\log_2 N)$, i.e. half the operation count of the split-radix algorithm. The orders of these algorithms are thus the same, since we here have adapted to read data.

e) Explain that, if you had not employed the trick from Exercise 4.7, we would instead have obtained $M_N = O\left(\frac{2}{3}\log_2 N\right)$, and $A_N = O\left(\frac{4}{3}\log_2 N\right)$, which equal the orders for the number of multiplications/additions for the split-radix algorithm. In particular, the order of the operation count remains the same, but the trick from Exercise 4.7 turned a bigger percentage of the arithmetic operations into additions.

The algorithm we here have developed thus is constructed from the beginning to apply for real data only. Another advantage of the new algorithm is that it can be used to compute both the DCT and the DFT.

Exercise 4.9: Implementation of the IFFT/IDCT

We did not write down corresponding algorithms for the revised IFFT and IDCT algorithms. We will consider this in this exercise.

a) Using equations (2.13) in the compendium-(4.14) in the compendium, show that

$$\begin{aligned}\Re(y_n) - \Re(y_{N/2-n}) &= 2(C_{N/4}\boldsymbol{z})_n \\ \Im(y_n) + \Im(y_{N/2-n}) &= 2(C_{N/4}\boldsymbol{w})_{N/4-n}\end{aligned}$$

for $1 \leq n \leq N/4 - 1$. Explain how one can compute $\boldsymbol{z}$ and $\boldsymbol{w}$ from this using two IDCT's of length $N/4$.

b) Using equations (2.13) in the compendium-(4.14) in the compendium, show that

$$\begin{aligned}\Re(y_n) + \Re(y_{N/2-n}) &= \Re((\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) \\ \Im(y_n) - \Im(y_{N/2-n}) &= \Im((\mathrm{DFT}_{N/2}\boldsymbol{x}^{(e)})_n),\end{aligned}$$

and explain how one can compute $\boldsymbol{x}^{(e)}$ from this using an IFFT of length $N/2$.

4.4 Summary

We started this chapter by extending a previous result which had to do with that the Fourier series of a symmetric function converged quicker. To build on this we first needed to define symmetric extensions of vectors and symmetric vectors, before we classified symmetric extensions in the frequency domain. From this we could find a nice, orthonormal basis for the symmetric extensions, which lead us to the definition of the DCT. We also saw a connection with symmetric filters: These are exactly the filters which preserve symmetric extensions, and we could characterize symmetric filters restricted to symmetric extension as an N-dimensional mapping. We also showed that it is smart to replace the DFT with the DCT when we work with filters which are known to be symmetric. Among other things, this lead to a better way of approximating analog filters, and better interpolation of functions.

We also showed how to obtain an efficient implementation of the DCT, which could reuse the FFT implementation. The DCT has an important role in the MP3 standard. As we have explained, the MP3 standard applies several filters to the sound, in order to split it into bands concentrating on different frequency ranges. In Section 8.3 we will look closer at how these filters can be implemented and constructed. The implementation can use transforms similar to the MDCT, as explained in Exercise 4.5. The MDCT is also used in the more advanced version of the MP3 standard (layer III). Here it is applied to the filtered data to obtain a higher spectral resolution of the sound. The MDCT is applied to groups

of 576 (in special circumstances 192) samples. The MP3 standard document [16] does not dig into the theory for this, only representing what is needed in order to make an implementation. It is somewhat difficult to read this document, since it is written in quite a different language, familiar mainly to those working with international standards.

The different type of cosine-matrices can all be associated with some extension strategy for the signal. [25] contains a review of these.

The DCT is particularly popular for processing sound data before they are compressed with lossless techniques such as Huffman coding or arithmetic coding. The reason is, as mentioned, that the DCT provides a better approximation from a low-dimensional space than the DFT does, and that it has a very efficient implementation. Libraries exist which goes into lengths to provide efficient implementation of the FFT and the DCT. FFTW, short for *Fastest Fourier Transform in the West* [14], is perhaps the best known of these.

Signal processing literature often does not motivate digital filters in explaining where they come from, and where the input to the filters come from. Using analog filters to motivate this, and to argue for improvements in using the DCT and symmeric extensions, is not that common. Much literature simply says that the property of linear phase is good, without elaborating on this further.

Chapter 5

Motivation for wavelets and some simple examples

In the first part of the book our focus was to approximate functions or vectors with trigonometric functions. We saw that the Discrete Fourier transform could be used to obtain a representation of a vector in terms of such functions, and that computations could be done efficiently with the FFT algorithm. This was useful for analyzing, filtering, and compressing sound and other discrete data. The approach with trigonometric functions has some limitations, however. One of these is that, in a representation with trigonometric functions, the frequency content is fixed over time. This is in contrast with most sound data, where the characteristics are completely different in different parts. We have also seen that, even if a sound has a simple representation in terms of trigonometric functions on two different parts, the representation of the entire sound may not be simple. In particular, if the function is nonzero only on a very small interval, a representation of it in terms of trigonometric functions is not so simple.

In this chapter we are going to introduce the basic properties of an alternative to Fourier analysis for representing functions. This alternative is called wavelets. Similar to Fourier analysis, wavelets are also based on the idea of expressing a function in some basis. But in contrast to Fourier analysis, where the basis is fixed, wavelets provide a general framework with many different types of bases. In this chapter we first give a motivation for wavelets, before we continue by introducing some very simple wavelets. The first wavelet we look at can be interpreted as an approximation scheme based on piecewise constant functions. The next wavelet we look at is similar, but with piecewise linear functions used instead. Following these examples we will establish a more general framework, based on experiences from the simple wavelets. In the following chapters we will interpret this framework in terms of filters, and use this connection to construct even more interesting wavelets.

The examples in this and the next chapters can be run from the notebook `applinalgnbchap5.ipynb`. Core functions are collected in a module called `dwt`.

5.1 Why wavelets?

The left image in Figure 5.1 shows a view of the entire Earth.

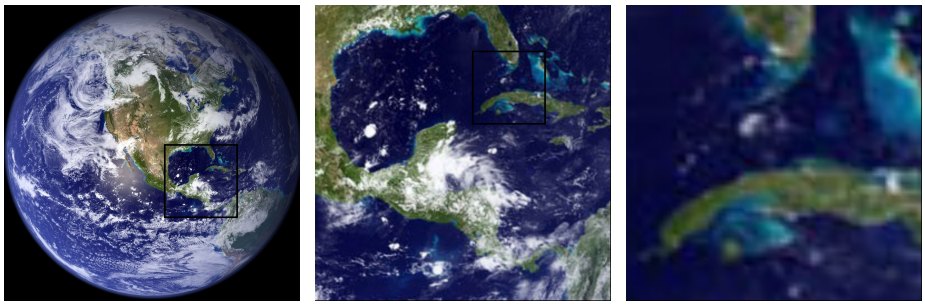

Figure 5.1: A view of Earth from space, together with versions of the image where we have zoomed in.

The startup image in Google Earth^{TM}, a program for viewing satellite images, maps and other geographic information, is very similar to this. In the middle image we have zoomed in on the Mexican Gulff, as marked with a rectangle in the left image. Similarly, in the right image we have further zoomed in on Cuba and a small portion of Florida, as marked with a rectangle in the middle image. There is clearly an amazing amount of information available behind a program like Google Earth^{TM}, since we there can zoom further in, and obtain enough detail to differentiate between buildings and even trees or cars all over the Earth. So, when the Earth is spinning in the opening screen of Google Earth^{TM}, all the Earth's buildings appear to be spinning with it! If this was the case the Earth would not be spinning on the screen, since there would just be so much information to process that a laptop would not be able to display a rotating Earth.

There is a simple reason that the globe can be shown spinning in spite of the huge amounts of information that need to be handled. We are going to see later that a digital image is just a rectangular array of numbers that represent the color at a dense set of points. As an example, the images in Figure 5.1 are made up of a grid of 1064×1064 points, which gives a total of 1 132 096 points. The color at a point is represented by three eight-bit integers, which means that the image files contain a total of 3 396 288 bytes each. So regardless of how close to the surface of the Earth our viewpoint is, the resulting image always contains the same number of points. This means that when we are far away from the Earth we can use a very coarse model of the geographic information that is being displayed, but as we zoom in, we need to display more details and therefore need a more accurate model.

Observation 5.1. *Images models.*

When discrete information is displayed in an image, there is no need to use a mathematical model that contains more detail than what is visible in the image.

A consequence of Observation 5.1 is that for applications like Google Earth^{TM} we should use a mathematical model that makes it easy to switch between different levels of detail, or different resolutions. Such models are called *multiresolution models*, and wavelets are prominent examples of this kind of models. We will see that multiresolution models also provide us with means of approximating functions, just as Taylor series and Fourier series. Our new approximation scheme differs from these in one important respect, however: When we approximate with Taylor series and Fourier series, the error must be computed at the same data points as well, so that the error contains just as much information as the approximating function, and the function to be approximated. Multiresolution models on the other hand will be defined in such a way that the error and the "approximating function" each contain half of the information from the function we approximate, i.e. their amount of data is reduced. This property makes multiresolution models attractive for the problems at hand, when compared to approaches such as Taylor series and Fourier series.

When we zoom in with Google Earth^{TM}, it seems that this is done continuously. The truth is probably that the program only has representations at some given resolutions (since each representation requires memory), and that one interpolates between these to give the impression of a continuous zoom. In the coming chapters we will first look at how we can represent the information at different resolutions, so that only new information at each level is included.

We will now turn to how wavelets are defined more formally, and construct the simplest wavelet we have. Its construction goes in the following steps: First we introduce what we call resolution spaces, and the corresponding scaling function. Then we introduce the detail spaces, and the corresponding mother wavelet. These two functions will give rise to certain bases for these spaces, and we will define the Discrete Wavelet Transform as a change of coordinates between these bases.

5.2 A wavelet based on piecewise constant functions

Our starting point will be the space of piecewise constant functions on an interval $[0, N)$. This will be called a resolution space.

Definition 5.2. *The resolution space V_0.*

Let N be a natural number. The resolution space V_0 is defined as the space of functions defined on the interval $[0, N)$ that are constant on each subinterval $[n, n+1)$ for $n = 0, \ldots, N-1$.

Note that this also corresponds to piecewise constant functions which are periodic with period N. We will, just as we did in Fourier analysis, identify a function defined on $[0, N)$ with its (period N) periodic extension. An example of a function in V_0 for $N = 10$ is shown in Figure 5.2. It is easy to check that V_0 is a linear space, and for computations it is useful to know the dimension of the space and have a basis.

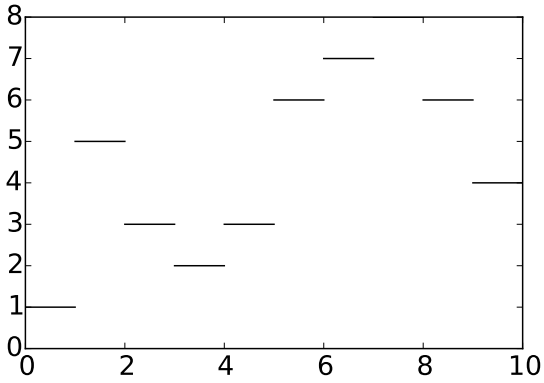

Figure 5.2: A piecewise constant function.

Lemma 5.3. *The function ϕ.*
Define the function $\phi(t)$ by

$$\phi(t) = \begin{cases} 1, & \text{if } 0 \le t < 1; \\ 0, & \text{otherwise;} \end{cases} \tag{5.1}$$

and set $\phi_n(t) = \phi(t-n)$ for any integer n. The space V_0 has dimension N, and the N functions $\{\phi_n\}_{n=0}^{N-1}$ form an orthonormal basis for V_0 with respect to the standard inner product

$$\langle f, g \rangle = \int_0^N f(t)g(t)\,dt. \tag{5.2}$$

In particular, any $f \in V_0$ can be represented as

$$f(t) = \sum_{n=0}^{N-1} c_n \phi_n(t) \tag{5.3}$$

for suitable coefficients $(c_n)_{n=0}^{N-1}$. The function ϕ_n is referred to as the *characteristic* function of the interval $[n, n+1)$

Note the small difference between the inner product we define here from the inner product we used for functions previously: Here there is no scaling $1/T$ involved. Also, for wavelets we will only consider real functions, and the inner product will therefore not be defined for complex functions. Two examples of the basis functions defined in Lemma 5.5 are shown in Figure 5.3.

Proof. Two functions ϕ_{n_1} and ϕ_{n_2} with $n_1 \neq n_2$ clearly satisfy $\int \phi_{n_1}(t)\phi_{n_2}(t)dt = 0$ since $\phi_{n_1}(t)\phi_{n_2}(t) = 0$ for all values of x. It is also easy to check that $\|\phi_n\| = 1$ for all n. Finally, any function in V_0 can be written as a linear combination the functions ϕ_0, ϕ_1, …, ϕ_{N-1}, so the conclusion of the lemma follows. □

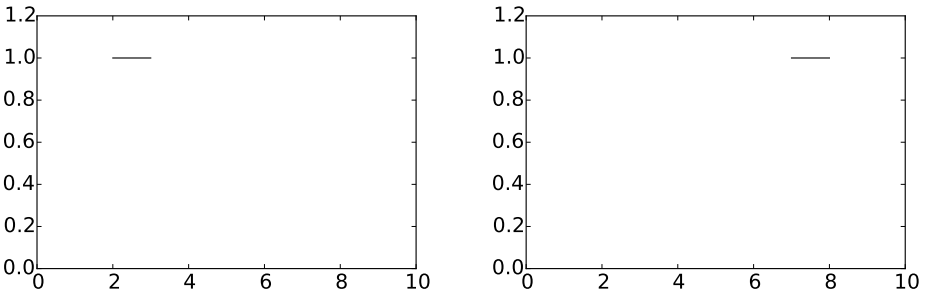

Figure 5.3: The basis functions ϕ_2 and ϕ_7 from $\boldsymbol{\phi}_0$.

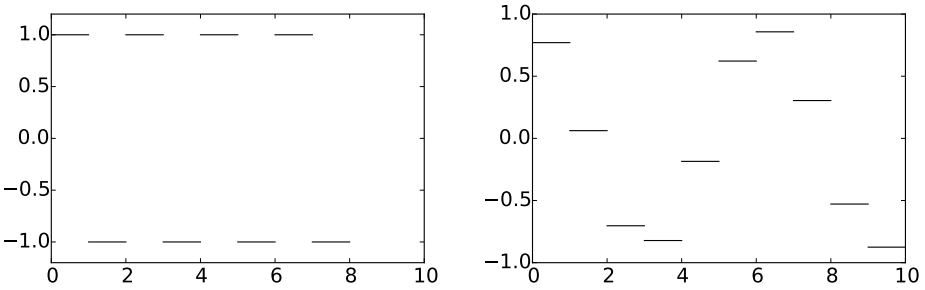

Figure 5.4: Examples of functions from V_0. The square wave in V_0 (left), and an approximation to $\cos t$ from V_0 (right).

In our discussion of Fourier analysis, the starting point was the function $\sin(2\pi t)$ that has frequency 1. We can think of the space V_0 as being analogous to this function: The function $\sum_{n=0}^{N-1}(-1)^n\phi_n(t)$ is (part of the) square wave that we discussed in Chapter 1, and which also oscillates regularly like the sine function, see the left plot in Figure 5.4. The difference is that we have more flexibility since we have a whole space at our disposal instead of just one function — the right plot in Figure 5.4 shows another function in V_0.

In Fourier analysis we obtained a linear space of possible approximations by including sines of frequency 1, 2, 3, ..., up to some maximum. We use a similar approach for constructing wavelets, but we double the frequency each time and label the spaces as V_0, V_1, V_2, ...

Definition 5.4. *Refined resolution spaces.*

The space V_m for the interval $[0, N)$ is the space of piecewise linear functions defined on $[0, N)$ that are constant on each subinterval $[n/2^m, (n+1)/2^m)$ for $n = 0, 1, \ldots, 2^m N - 1$.

Some examples of functions in the spaces V_1, V_2 and V_3 for the interval $[0, 10]$ are shown in Figure 5.5. As m increases, we can represent smaller details. In particular, the function in the rightmost is a piecewise constant function that oscillates like $\sin(2\pi 2^2 t)$ on the interval $[0, 10]$.

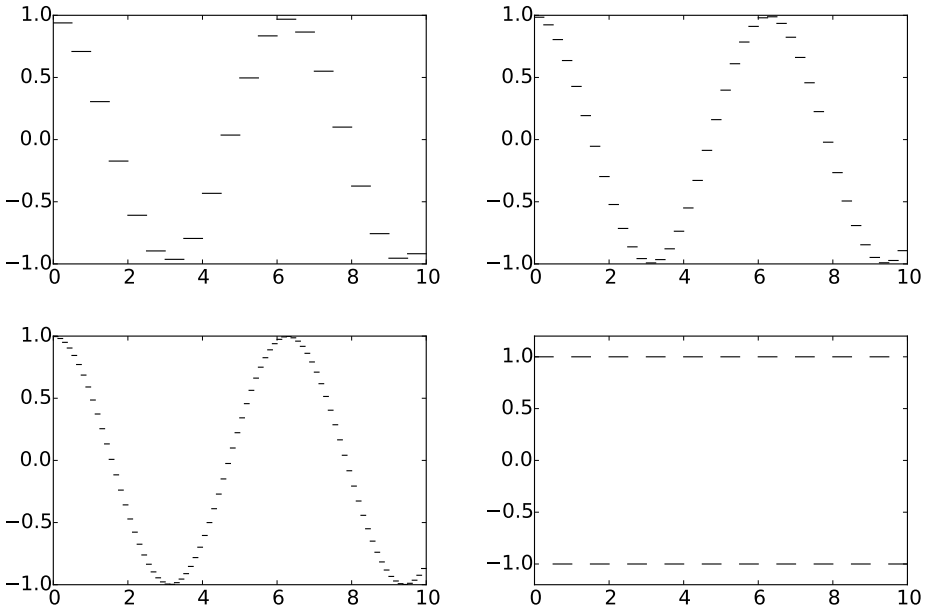

Figure 5.5: Piecewise constant approximations to $\cos t$ on the interval $[0, 10]$ in the spaces V_1, V_2, and V_3. The lower right plot shows the square wave in V_2.

It is easy to find a basis for V_m, we just use the characteristic functions of each subinterval.

Lemma 5.5. *Basis for V_m.*

Let $[0, N)$ be a given interval with N some positive integer. Then the dimension of V_m is $2^m N$. The functions

$$\phi_{m,n}(t) = 2^{m/2}\phi(2^m t - n), \quad \text{for } n = 0,\, 1,\, \ldots,\, 2^m N - 1. \tag{5.4}$$

$\{\phi_{m,n}\}_{n=0}^{2^m N-1}$ form an orthonormal basis for V_m, which we will denote by $\boldsymbol{\phi}_m$. Any function $f \in V_m$ can thus be represented uniquely as

$$f(t) = \sum_{n=0}^{2^m N-1} c_{m,n}\phi_{m,n}(t).$$

Proof. The functions given by Equation (5.4) are nonzero on the subintervals $[n/2^m, (n+1)/2^m)$ which we referred to in Definition 5.4, so that $\phi_{m,n_1}\phi_{m,n_2} = 0$ when $n_1 \neq n_2$, since these intervals are disjoint. The only mysterious thing may be the normalisation factor $2^{m/2}$. This comes from the fact that

$$\int_0^N \phi(2^m t - n)^2\, dt = \int_{n/2^m}^{(n+1)/2^m} \phi(2^m t - n)^2\, dt = 2^{-m}\int_0^1 \phi(u)^2\, du = 2^{-m}.$$

The normalisation therefore thus ensures that $\|\phi_{m,n}\| = 1$ for all m. □

In the following we will always denote the coordinates in the basis $\boldsymbol{\phi}_m$ by $c_{m,n}$. Note that our definition restricts the dimensions of the spaces we study to be on the form $N2^m$. In Chapter 6 we will explain how this restriction can be dropped, but until then the dimensions will be assumed to be on this form. In the theory of wavelets, the function ϕ is also called a *scaling function*. The origin behind this name is that the scaled (and translated) functions $\phi_{m,n}$ of ϕ are used as basis functions for the refined resolution spaces. Later on we will see that other scaling functions ϕ can be chosen, where the scaled versions $\phi_{m,n}$ will be used to define similar resolution spaces, with slightly different properties.

5.2.1 Function approximation property

Each time m is increased by 1, the dimension of V_m doubles, and the subinterval on which the functions in V_m are constant are halved in size. It therefore seems reasonable that, for most functions, we can find good approximations in V_m provided m is big enough.

Theorem 5.6. *Resolution spaces and approximation.*

Let f be a given function that is continuous on the interval $[0, N]$. Given $\epsilon > 0$, there exists an integer $m \geq 0$ and a function $g \in V_m$ such that

$$\big|f(t) - g(t)\big| \leq \epsilon$$

for all t in $[0, N]$.

Proof. Since f is (uniformly) continuous on $[0, N]$, we can find an integer m so that $\big|f(t_1) - f(t_2)\big| \leq \epsilon$ for any two numbers t_1 and t_2 in $[0, N]$ with $|t_1 - t_2| \leq 2^{-m}$. Define the approximation g by

$$g(t) = \sum_{n=0}^{2^m N-1} f\big(t_{m,n+1/2}\big)\phi_{m,n}(t),$$

where $t_{m,n+1/2}$ is the midpoint of the subinterval $\big[n2^{-m}, (n+1)2^{-m}\big)$,

$$t_{m,n+1/2} = (n + 1/2)2^{-m}.$$

For t in this subinterval we then obviously have $|f(t) - g(t)| \leq \epsilon$, and since these intervals cover $[0, N]$, the conclusion holds for all $t \in [0, N]$. □

Theorem 5.6 does not tell us how to find the approximation g although the proof makes use of an approximation that interpolates f at the midpoint of each subinterval. Note that if we measure the error in the L^2-norm, we have

$$\|f - g\|^2 = \int_0^N \big|f(t) - g(t)\big|^2 \, dt \leq N\epsilon^2,$$

so $\|f - g\| \leq \epsilon\sqrt{N}$. We therefore have the following corollary.

Corollary 5.7. *Resolution spaces and approximation.*

Let f be a given continuous function on the interval $[0, N]$. Then

$$\lim_{m\to\infty} \|f - \text{proj}_{V_m}(f)\| = 0.$$

Figure 5.6 illustrates how some of the approximations of the function $f(x) = x^2$ from the resolution spaces for the interval $[0, 1]$ improve with increasing m.

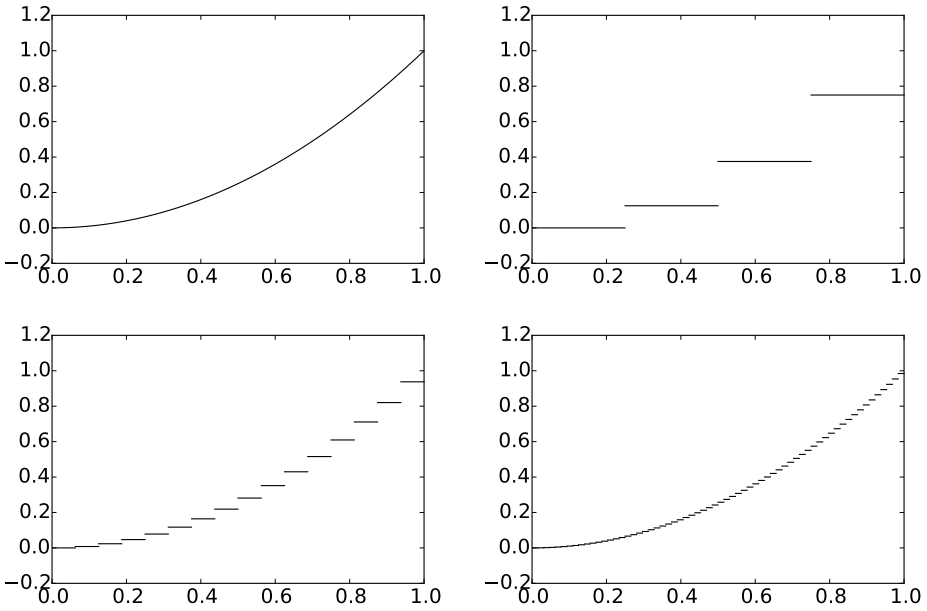

Figure 5.6: Comparison of the function defined by $f(t) = t^2$ on $[0, 1]$ with the projection onto V_2, V_4, and V_6, respectively.

5.2.2 Detail spaces and wavelets

So far we have described a family of function spaces that allow us to determine arbitrarily good approximations to a continuous function. The next step is to introduce the so-called detail spaces and the wavelet functions. We start by observing that since

$$[n, n+1) = [2n/2, (2n+1)/2) \cup [(2n+1)/2, (2n+2)/2),$$

we have

$$\phi_{0,n} = \frac{1}{\sqrt{2}}\phi_{1,2n} + \frac{1}{\sqrt{2}}\phi_{1,2n+1}.$$

This provides a formal proof of the intuitive observation that $V_0 \subset V_1$, for if $g \in V_0$, we can write

$$g(t) = \sum_{n=0}^{N-1} c_{0,n}\phi_{0,n}(t) = \sum_{n=0}^{N-1} c_{0,n}\big(\phi_{1,2n} + \phi_{1,2n+1}\big)/\sqrt{2},$$

and the right-hand side clearly lies in V_1. Since also

$$\begin{aligned}\phi_{m-1,n}(t) &= 2^{(m-1)/2}\phi(2^{m-1}t - n) = 2^{(m-1)/2}\phi_{0,n}(2^{m-1}t)\\ &= 2^{(m-1)/2}\frac{1}{\sqrt{2}}(\phi_{1,2n}(2^{m-1}t) + \phi_{1,2n+1}(2^{m-1}t))\\ &= 2^{(m-1)/2}(\phi(2^m t - 2n) + \phi(2^m t - (2n+1))) = \frac{1}{\sqrt{2}}(\phi_{m,2n}(t) + \phi_{m,2n+1}(t)),\end{aligned}$$

we also have that

$$\phi_{m-1,n} = \frac{1}{\sqrt{2}}\phi_{m,2n} + \frac{1}{\sqrt{2}}\phi_{m,2n+1}, \tag{5.5}$$

so that also $V_k \subset V_{k+1}$ for any integer $k \geq 0$.

Lemma 5.8. *Resolution spaces are nested.*
The spaces V_0, V_1, ..., V_m, ... are nested,

$$V_0 \subset V_1 \subset V_2 \subset \cdots \subset V_m \cdots .$$

This means that it is meaningful to project V_{k+1} onto V_k. The next step is to characterize the projection from V_1 onto V_0, and onto the orthogonal complement of V_0 in V_1. Before we do this, let us make the following definitions.

Definition 5.9. *Detail spaces.*
The orthogonal complement of V_{m-1} in V_m is denoted W_{m-1}. All the spaces W_k are also called detail spaces, or error spaces.

The name detail space is used since the projection from V_m onto V_{m-1} in considered as a (low-resolution) approximation, and the error, which lies in W_{m-1}, is the detail which is left out when we replace with this approximation. We will also write $g_m = g_{m-1} + e_{m-1}$ when we split $g_m \in V_m$ into a sum of a low-resolution approximation and a detail component. In the context of our Google Earth$^{\text{TM}}$example, in Figure 5.1 you should interpret g_0 as the left image, the middle image as an excerpt of g_1, and e_0 as the additional details which are needed to reproduce the middle image from the left image.

Since V_0 and W_0 are mutually orthogonal spaces they are also linearly independent spaces. When U and V are two such linearly independent spaces, we will write $U \oplus V$ for the vector space consisting of all vectors of the form $\boldsymbol{u} + \boldsymbol{v}$, with $\boldsymbol{u} \in U$, $\boldsymbol{v} \in V$. $U \oplus V$ is also called the *direct sum* of U and V. This also makes sense if we have more than two vector spaces (such as $U \oplus V \oplus W$),

and the direct sum clearly obeys the associate law $U \oplus (V \oplus W) = (U \oplus V) \oplus W$. Using the direct sum notation, we can first write

$$V_m = V_{m-1} \oplus W_{m-1}. \tag{5.6}$$

Since V_m has dimension $2^m N$, it follows that also W_m has dimension $2^m N$. We can continue the direct sum decomposition by also writing V_{m-1} as a direct sum, then V_{m-2} as a direct sum, and so on, and end up with

$$V_m = V_0 \oplus W_0 \oplus W_1 \oplus \cdots \oplus W_{m-1}, \tag{5.7}$$

where the spaces on the right hand side have dimension $N, N, 2N, \ldots, 2^{m-1}N$. This decomposition wil be important for our purposes. It says that the resolution space V_m acan be written as the sum of a lower order resolution space V_0, and m detail spaces $W_0, \ldots, W_{m-1}$. We will later interpret this splitting into a low-resolution component and m detail components.

It turns out that the following function will play the same role for the detail space W_k as the function ϕ plays for the resolution space V_k.

Definition 5.10. *The function ψ.*

We define

$$\psi(t) = \big(\phi_{1,0}(t) - \phi_{1,1}(t)\big)/\sqrt{2} = \phi(2t) - \phi(2t-1), \tag{5.8}$$

and

$$\psi_{m,n}(t) = 2^{m/2}\psi(2^m t - n), \quad \text{for } n = 0,\, 1,\, \ldots,\, 2^m N - 1. \tag{5.9}$$

The functions ϕ and ψ are shown in Figure 5.7.

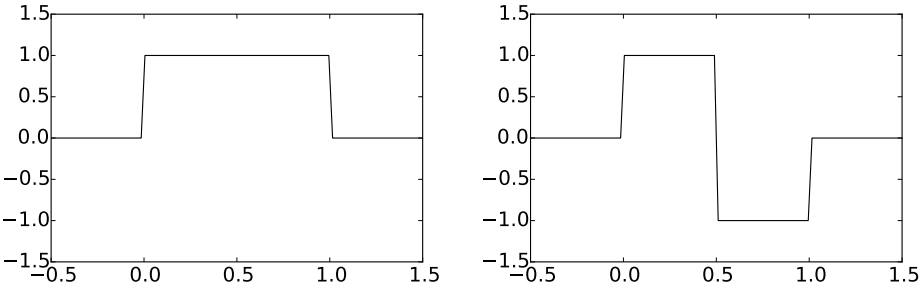

Figure 5.7: The functions ϕ and ψ we used to analyse the space of piecewise constant functions.

As in the proof for Equation (5.5), it follows that

$$\psi_{m-1,n} = \frac{1}{\sqrt{2}}\phi_{m,2n} - \frac{1}{\sqrt{2}}\phi_{m,2n+1}, \tag{5.10}$$

Clearly ψ is supported on $[0, 1)$, and $\|\psi\| = 1$. From this it follows as for $\boldsymbol{\phi}_0$ that the $\{\psi_{0,n}\}_{n=0}^{N-1}$ are orthonormal. In the same way as for $\boldsymbol{\phi}_m$, it follows

also that the $\{\psi_{m,n}\}_{n=0}^{2^m N-1}$ is orthonormal for any m. We will write $\boldsymbol{\psi}_m$ for the orthonormal basis $\{\psi_{m,n}\}_{n=0}^{2^m N-1}$, and we will always denote the coordinates in the basis $\boldsymbol{\psi}_m$ by $w_{m,n}$. The next result motivates the definition of ψ, and states how we can project from V_1 onto V_0 and W_0, i.e. find the low-resolution approximation and the detail component of $g_1 \in V_1$.

Lemma 5.11. *Orthonormal bases.*

For $0 \leq n < N$ we have that

$$\operatorname{proj}_{V_0}(\phi_{1,n}) = \begin{cases} \phi_{0,n/2}/\sqrt{2}, & \text{if } n \text{ is even;} \\ \phi_{0,(n-1)/2}/\sqrt{2}, & \text{if } n \text{ is odd.} \end{cases} \tag{5.11}$$

$$\operatorname{proj}_{W_0}(\phi_{1,n}) = \begin{cases} \psi_{0,n/2}/\sqrt{2}, & \text{if } n \text{ is even;} \\ -\psi_{0,(n-1)/2}/\sqrt{2}, & \text{if } n \text{ is odd.} \end{cases} \tag{5.12}$$

In particular, $\boldsymbol{\psi}_0$ is an orthonormal basis for W_0. More generally, if $g_1 = \sum_{n=0}^{2N-1} c_{1,n}\phi_{1,n} \in V_1$, then

$$\operatorname{proj}_{V_0}(g_1) = \sum_{n=0}^{N-1} c_{0,n}\phi_{0,n}, \text{ where } c_{0,n} = \frac{c_{1,2n} + c_{1,2n+1}}{\sqrt{2}} \tag{5.13}$$

$$\operatorname{proj}_{W_0}(g_1) = \sum_{n=0}^{N-1} w_{0,n}\psi_{0,n}, \text{ where } w_{0,n} = \frac{c_{1,2n} - c_{1,2n+1}}{\sqrt{2}}. \tag{5.14}$$

Proof. We first observe that $\phi_{1,n}(t) \neq 0$ if and only if $n/2 \leq t < (n+1)/2$. Suppose that n is even. Then the intersection

$$\left[\frac{n}{2}, \frac{n+1}{2}\right) \cap [n_1, n_1 + 1) \tag{5.15}$$

is nonempty only if $n_1 = \frac{n}{2}$. Using the orthogonal decomposition formula we get

$$\begin{aligned} \operatorname{proj}_{V_0}(\phi_{1,n}) &= \sum_{k=0}^{N-1} \langle \phi_{1,n}, \phi_{0,k} \rangle \phi_{0,k} = \langle \phi_{1,n}, \phi_{0,n_1} \rangle \phi_{0,n_1} \\ &= \int_{n/2}^{(n+1)/2} \sqrt{2}\, dt\, \phi_{0,n/2} = \frac{1}{\sqrt{2}}\phi_{0,n/2}. \end{aligned}$$

Using this we also get

$$\begin{aligned} \operatorname{proj}_{W_0}(\phi_{1,n}) &= \phi_{1,n} - \frac{1}{\sqrt{2}}\phi_{0,n/2} = \phi_{1,n} - \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\phi_{1,n} + \frac{1}{\sqrt{2}}\phi_{1,n+1}\right) \\ &= \frac{1}{2}\phi_{1,n} - \frac{1}{2}\phi_{1,n+1} = \psi_{0,n/2}/\sqrt{2}. \end{aligned}$$

This proves the expressions for both projections when n is even. When n is odd, the intersection (5.15) is nonempty only if $n_1 = (n-1)/2$, which gives the expressions for both projections when n is odd in the same way. In particular we get

$$\begin{aligned}\mathrm{proj}_{W_0}(\phi_{1,n}) &= \phi_{1,n} - \frac{\phi_{0,(n-1)/2}}{\sqrt{2}} = \phi_{1,n} - \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\phi_{1,n-1} + \frac{1}{\sqrt{2}}\phi_{1,n}\right) \\ &= \frac{1}{2}\phi_{1,n} - \frac{1}{2}\phi_{1,n-1} = -\psi_{0,(n-1)/2}/\sqrt{2}.\end{aligned}$$

$\boldsymbol{\psi}_0$ must be an orthonormal basis for W_0 since $\boldsymbol{\psi}_0$ is contained in W_0, and both have dimension N.

We project the function g_1 in V_1 using the formulas in (5.11). We first split the sum into even and odd values of n,

$$g_1 = \sum_{n=0}^{2N-1} c_{1,n}\phi_{1,n} = \sum_{n=0}^{N-1} c_{1,2n}\phi_{1,2n} + \sum_{n=0}^{N-1} c_{1,2n+1}\phi_{1,2n+1}. \tag{5.16}$$

We can now apply the two formulas in (5.11),

$$\begin{aligned}\mathrm{proj}_{V_0}(g_1) &= \mathrm{proj}_{V_0}\left(\sum_{n=0}^{N-1} c_{1,2n}\phi_{1,2n} + \sum_{n=0}^{N-1} c_{1,2n+1}\phi_{1,2n+1}\right) \\ &= \sum_{n=0}^{N-1} c_{1,2n}\,\mathrm{proj}_{V_0}(\phi_{1,2n}) + \sum_{n=0}^{N-1} c_{1,2n+1}\,\mathrm{proj}_{V_0}(\phi_{1,2n+1}) \\ &= \sum_{n=0}^{N-1} c_{1,2n}\phi_{0,n}/\sqrt{2} + \sum_{n=0}^{N-1} c_{1,2n+1}\phi_{0,n}/\sqrt{2} \\ &= \sum_{n=0}^{N-1} \frac{c_{1,2n} + c_{1,2n+1}}{\sqrt{2}}\phi_{0,n}\end{aligned}$$

which proves Equation (5.13). Equation (5.14) is proved similarly. □

In Figure 5.8 we have used Lemma 5.11 to plot the projections of $\phi_{1,0} \in V_1$ onto V_0 and W_0. It is an interesting exercise to see from the plots why exactly these functions should be least-squares approximations of $\phi_{1,n}$. It is also an interesting exercise to prove the following from Lemma 5.11:

Proposition 5.12. *Projections.*

Let $f(t) \in V_1$, and let $f_{n,1}$ be the value f attains on $[n, n+1/2)$, and $f_{n,2}$ the value f attains on $[n+1/2, n+1)$. Then $\mathrm{proj}_{V_0}(f)$ is the function in V_0 which equals $(f_{n,1} + f_{n,2})/2$ on the interval $[n, n+1)$. Moreover, $\mathrm{proj}_{W_0}(f)$ is the function in W_0 which is $(f_{n,1} - f_{n,2})/2$ on $[n, n+1/2)$, and $-(f_{n,1} - f_{n,2})/2$ on $[n+1/2, n+1)$.

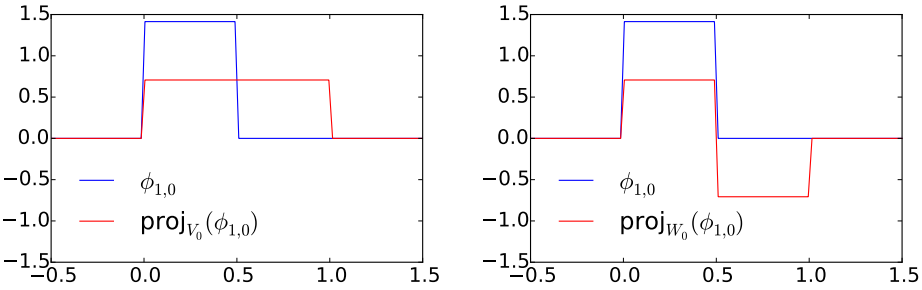

Figure 5.8: The projection of $\phi_{1,0} \in V_1$ onto V_0 and W_0.

In other words, the projection on V_0 is constructed by averaging on two subintervals, while the projection on W_0 is constructed by taking the difference from the mean. This sounds like a reasonable candidate for the least-squares approximations. In the exercise we generalize these observations.

In the same way as in Lemma 5.11, it is possible to show that

$$\text{proj}_{W_{m-1}}(\phi_{m,n}) = \begin{cases} \psi_{m-1,n/2}/\sqrt{2}, & \text{if } n \text{ is even;} \\ -\psi_{m-1,(n-1)/2}/\sqrt{2}, & \text{if } n \text{ is odd.} \end{cases} \tag{5.17}$$

From this it follows as before that $\boldsymbol{\psi}_m$ is an orthonormal basis for W_m. If $\{\mathcal{B}_i\}_{i=1}^n$ are mutually independent bases, we will in the following write $(\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n)$ for the basis where the basis vectors from $\mathcal{B}_i$ are included before $\mathcal{B}_j$ when $i < j$. With this notation, the decomposition in Equation (5.7) can be restated as follows

Theorem 5.13. *Bases for V_m.*
$\boldsymbol{\phi}_m$ and $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \boldsymbol{\psi}_1, \cdots, \boldsymbol{\psi}_{m-1})$ are both bases for V_m.

The function ψ thus has the property that its dilations and translations together span the detail components. Later we will encounter other functions, which also will be denoted by ψ, and have similar properties. In the theory of wavelets, such ψ are called *mother wavelets*. There is one important property of ψ, which we will return to:

Observation 5.14. *Vanishing moment.*
We have that $\int_0^N \psi(t)dt = 0$.

This can be seen directly from the plot in Figure 5.7, since the parts of the graph above and below the x-axis cancel. In general we say that ψ has k vanishing moments if the integrals $\int t^l \psi(t)dt = 0$ for all $0 \leq l \leq k-1$. Due to Observation 5.14, ψ has one vanishing moment. In Chapter 7 we will show that mother wavelets with many vanishing moments are very desirable when it comes to approximation of functions.

We now have all the tools needed to define the Discrete Wavelet Transform.

Definition 5.15. *Discrete Wavelet Transform.*

The DWT (Discrete Wavelet Transform) is defined as the change of coordinates from $\boldsymbol{\phi}_1$ to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$. More generally, the m-level DWT is defined as the change of coordinates from $\boldsymbol{\phi}_m$ to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \boldsymbol{\psi}_1, \cdots, \boldsymbol{\psi}_{m-1})$. In an m-level DWT, the change of coordinates from

$$(\boldsymbol{\phi}_{m-k+1}, \boldsymbol{\psi}_{m-k+1}, \boldsymbol{\psi}_{m-k+2}, \cdots, \boldsymbol{\psi}_{m-1}) \text{ to } (\boldsymbol{\phi}_{m-k}, \boldsymbol{\psi}_{m-k}, \boldsymbol{\psi}_{m-k+1}, \cdots, \boldsymbol{\psi}_{m-1}) \tag{5.18}$$

is also called the k'th stage. The (m-level) IDWT (Inverse Discrete Wavelet Transform) is defined as the change of coordinates the opposite way.

The DWT corresponds to replacing as many ϕ-functions as we can with ψ-functions, i.e. replacing the original function with a sum of as much detail at different resolutions as possible. We now can state the following result.

Theorem 5.16. *Expression for the DWT.*

If $g_m = g_{m-1} + e_{m-1}$ with

$$g_m = \sum_{n=0}^{2^m N-1} c_{m,n}\phi_{m,n} \in V_m,$$

$$g_{m-1} = \sum_{n=0}^{2^{m-1}N-1} c_{m-1,n}\phi_{m-1,n} \in V_{m-1} \quad e_{m-1} = \sum_{n=0}^{2^{m-1}N-1} w_{m-1,n}\psi_{m-1,n} \in W_{m-1},$$

then the change of coordinates from $\boldsymbol{\phi}_m$ to $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$ (i.e. first stage in a DWT) is given by

$$\begin{pmatrix} c_{m-1,n} \\ w_{m-1,n} \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} c_{m,2n} \\ c_{m,2n+1} \end{pmatrix} \tag{5.19}$$

Conversely, the change of coordinates from $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$ to $\boldsymbol{\phi}_m$ (i.e. the last stage in an IDWT) is given by

$$\begin{pmatrix} c_{m,2n} \\ c_{m,2n+1} \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} c_{m-1,n} \\ w_{m-1,n} \end{pmatrix} \tag{5.20}$$

Proof. Equations (5.5) and (5.10) say that

$$\phi_{m-1,n} = \phi_{m,2n}/\sqrt{2} + \phi_{m,2n+1}/\sqrt{2} \quad \psi_{m-1,n} = \phi_{m,2n}/\sqrt{2} - \phi_{m,2n+1}/\sqrt{2}.$$

The change of coordinate matrix from the basis $\{\phi_{m-1,n}, \psi_{m-1,n}\}$ to $\{\phi_{m,2n}, \phi_{m,2n+1}\}$ is thus $\begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$. This proves Equation (5.20). Equation (5.19) follows immediately since this matrix equals its inverse. □

Above we assumed that N is even. In Exercise 5.8 we will see how we can handle the case when N is odd.

From Theorem 5.16, we see that, if we had defined

$$\mathcal{C}_m = \{\phi_{m-1,0}, \psi_{m-1,0}, \phi_{m-1,1}, \psi_{m-1,1}, \cdots, \phi_{m-1,2^{m-1}N-1}, \psi_{m-1,2^{m-1}N-1}\}. \tag{5.21}$$

i.e. we have reordered the basis vectors in $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$ (the subscript m is used since $\mathcal{C}_m$ is a basis for V_m), it is apparent from Equation (5.20) that $G = P_{\boldsymbol{\phi}_m \leftarrow \mathcal{C}_m}$ is the matrix where

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

is repeated along the main diagonal $2^{m-1}N$ times. Also, from Equation (5.19) it is apparent that $H = P_{\mathcal{C}_m \leftarrow \boldsymbol{\phi}_m}$ is the same matrix. Such matrices are called *block diagonal matrices*. This particular block diagonal matrix is clearly orthogonal. Let us make the following definition

Definition 5.17. *DWT and IDWT kernel transformations.*

The matrices $H = P_{\mathcal{C}_m \leftarrow \boldsymbol{\phi}_m}$ and $G = P_{\boldsymbol{\phi}_m \leftarrow \mathcal{C}_m}$ are called the *DWT and IDWT kernel transformations*. The DWT and the IDWT can be expressed in terms of these kernel transformations by

$$\text{DWT} = P_{(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1}) \leftarrow \mathcal{C}_m} H \text{ and IDWT} = G P_{\mathcal{C}_m \leftarrow (\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})},$$

respectively, where

- $P_{(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1}) \leftarrow \mathcal{C}_m}$ is a permutation matrix which groups the even elements first, then the odd elements,
- $P_{\mathcal{C}_m \leftarrow (\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})}$ is a permutation matrix which places the first half at the even indices, the last half at the odd indices.

Clearly, the kernel transformations H and G also invert each other. The point of using the kernel transformation is that they compute the output sequentially, similarly to how a filter does. Clearly also the kernel transformations are very similar to a filter, and we will return to this in the next chapter.

At each level in a DWT, V_k is split into one low-resolution component from V_{k-1}, and one detail component from W_{k-1}. We have illustrated this in figure 5.9, where the arrows represent changes of coordinates.

The detail component from W_{k-1} is not subject to further transformation. This is seen in the figure since $\boldsymbol{\psi}_{k-1}$ is a leaf node, i.e. there are no arrows going out from $\boldsymbol{\psi}_{m-1}$. In a similar illustration for the IDWT, the arrows would go the opposite way.

The Discrete Wavelet Transform is the analogue in a wavelet setting to the Discrete Fourier transform. When applying the DFT to a vector of length N,

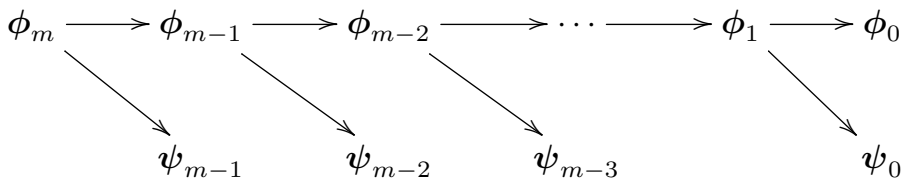

Figure 5.9: Illustration of a wavelet transform.

one starts by viewing this vector as coordinates relative to the standard basis. When applying the DWT to a vector of length N, one instead views the vector as coordinates relative to the basis $\boldsymbol{\phi}_m$. This makes sense in light of Exercise 5.1.

What you should have learned in this section.

- Definition of resolution spaces (V_m), detail spaces (W_m), scaling function (ϕ), and mother wavelet (ψ) for the wavelet based on piecewise constant functions.
- The nesting of resolution spaces, and how one can project from one resolution space onto a lower order resolution space, and onto its orthogonal complement.
- The definition of the Discrete Wavelet Transform as a change of coordinates, and how this can be written down from relations between basis functions.

Exercise 5.1: Samples are the coordinate vector

Show that the coordinate vector for $f \in V_0$ in the basis $\{\phi_{0,0}, \phi_{0,1}, \ldots, \phi_{0,N-1}\}$ is $(f(0), f(1), \ldots . f(N-1))$.

Exercise 5.2: Proposition 5.12

Prove Proposition 5.12.

Exercise 5.3: Computing projections

In this exercise we will consider the two projections from V_1 onto V_0 and W_0.

a) Consider the projection proj_{V_0} of V_1 onto V_0. Use Lemma 5.11 to write down the matrix for proj_{V_0} relative to the bases $\boldsymbol{\phi}_1$ and $\boldsymbol{\phi}_0$.

b) Similarly, use Lemma 5.11 to write down the matrix for $\text{proj}_{W_0} : V_1 \to W_0$ relative to the bases $\boldsymbol{\phi}_1$ and $\boldsymbol{\psi}_0$.

Exercise 5.4: Computing projections 2

Consider again the projection proj_{V_0} of V_1 onto V_0.

a) Explain why $\text{proj}_{V_0}(\phi) = \phi$ and $\text{proj}_{V_0}(\psi) = 0$.

b) Show that the matrix of proj_{V_0} relative to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$ is given by the diagonal matrix where the first half of the entries on the diagonal are 1, the second half 0.

c) Show in a similar way that the projection of V_1 onto W_0 has a matrix relative to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$ given by the diagonal matrix where the first half of the entries on the diagonal are 0, the second half 1.

Exercise 5.5: Computing projections

Show that

$$\text{proj}_{V_0}(f) = \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt \right) \phi_{0,n}(t) \tag{5.22}$$

for any f. Show also that the first part of Proposition 5.12 follows from this.

Exercise 5.6: Finding the least squares error

Show that

$$\| \sum_n \left(\int_n^{n+1} f(t)dt \right) \phi_{0,n}(t) - f \|^2 = \langle f, f \rangle - \sum_n \left(\int_n^{n+1} f(t)dt \right)^2 .$$

This, together with the previous exercise, gives us an expression for the least-squares error for f from V_0 (at least after taking square roots). 2DO: Generalize to m

Exercise 5.7: Projecting on W_0

Show that

$$\text{proj}_{W_0}(f) = \sum_{n=0}^{N-1} \left(\int_n^{n+1/2} f(t)dt - \int_{n+1/2}^{n+1} f(t)dt \right) \psi_{0,n}(t) \tag{5.23}$$

for any f. Show also that the second part of Proposition 5.12 follows from this.

Exercise 5.8: When N is odd

When N is odd, the (first stage in a) DWT is defined as the change of coordinates from $(\phi_{1,0}, \phi_{1,1}, \ldots, \phi_{1,N-1})$ to

$$(\phi_{0,0}, \psi_{0,0}, \phi_{0,1}, \psi_{0,1}, \ldots, \phi_{0,(N-1)/2}, \psi_{(N-1)/2}, \phi_{0,(N+1)/2}).$$

Since all functions are assumed to have period N, we have that

$$\phi_{0,(N+1)/2} = \frac{1}{\sqrt{2}}(\phi_{1,N-1} + \phi_{1,N}) = \frac{1}{\sqrt{2}}(\phi_{1,0} + \phi_{1,N-1}).$$

From this relation one can find the last column in the change of coordinate matrix from $\boldsymbol{\phi}_0$ to $(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1)$, i.e. the IDWT matrix. In particular, when N is odd, we see that the last column in the IDWT matrix circulates to the upper right corner. In terms of coordinates, we thus have that

$$c_{1,0} = \frac{1}{\sqrt{2}}(c_{0,0} + w_{0,0} + c_{0,(N+1)/2}) \qquad c_{1,N-1} = \frac{1}{\sqrt{2}} c_{0,(N+1)/2}. \tag{5.24}$$

a) If $N = 3$, the DWT matrix equals $\frac{1}{\sqrt{2}} \begin{pmatrix} 1\&1 & 1 \\ 1 & -1\&0 \\ 0\&0 & 1 \end{pmatrix}$, and the inverse of this is $\frac{1}{\sqrt{2}} \begin{pmatrix} 1\&1 & -1 & \\ 1 & -1 & -1 \\ 0\&0 & 2 & \end{pmatrix}$. Explain from this that, when N is odd, the DWT matrix can be constructed by adding a column on the form $\frac{1}{\sqrt{2}}(-1, -1, 0, \ldots, 0, 2)$ to the DWT matrices we had for N even (in the last row zeros are also added). In terms of the coordinates, we thus have the additional formulas

$$c_{0,0} = \frac{1}{\sqrt{2}}(c_{1,0+c_{1,1}-c_{1,N-1}}) \quad w_{0,0} = \frac{1}{\sqrt{2}}(c_{1,0-c_{1,1}-c_{1,N-1}}) \quad c_{0,(N+1)/2} = \frac{1}{\sqrt{2}} 2c_{1,N-1}. \tag{5.25}$$

b) Explain that the DWT matrix is orthogonal if and only if N is even. Also explain that it is only the last column which spoils the orthogonality.

5.3 Implementation of the DWT and examples

The DWT is straightforward to implement: One simply needs to iterate Equation (5.19) in the compendium for $m, m-1, \ldots, 1$. We will use a DWT kernel function which takes as input the coordinates $(c_{m,0}, c_{m,1}, \ldots)$, and returns the coordinates $(c_{m-1,0}, w_{m-1,0}, c_{m-1,1}, w_{m-1,1}, \ldots)$, i.e. computes one stage of the DWT. This is a different order for the coordinates than that given by the basis $(\boldsymbol{\phi}_m, \boldsymbol{\psi}_m)$. The reason is that it is easier with this new order to compute the DWT in-place. As

an example, the kernel transformation for the Haar wavelet can be implemented as follows. For simplicity this first version of the code assumes that N is even:

```
def DWTKernelHaar(x, symm, dual):
    x /= sqrt(2)
    for k in range(2,len(x) - 1,2):
        a, b = x[k] + x[k+1], x[k] - x[k+1]
        x[k], x[k+1] = a, b
```

Note that the code above accepts two-dimensional data, just as our function `FFTImpl` did. Thus, the function may be applied simultaneously to all channels in a sound. The mysterious parameters `symm` and `dual` will be explained in Chapter 6. For now they have no role in the code, but will still appear several places in the code in this section. When N is even, `IDWTKernelHaar` can be implemented with the exact same code. When N is odd, we can use the results from Exercise 5.8 (see also Exercise 5.24). The reason for using a general kernel function will be apparent later, when we change to different types of wavelets.

Since the code above does not give the coordinates in the same order as $(\boldsymbol{\phi}_m, \boldsymbol{\psi}_m)$, an implementation of the DWT needs to organize the DWT coefficients in the right order, in addition to calling the kernel function for each stage, and applying the kernel to the right coordinates. Clearly, the coordinates from $\boldsymbol{\phi}_m$ end up at indices $k2^m$, where m represents the current stage, and k runs through the indices. The following function, called `DWTImpl`, follows this procedure. It takes as input the number of levels, `nres`, as well as the input vector $\boldsymbol{x}$, runs the DWT on $\boldsymbol{x}$ with the given number of resolutions, and returns the result:

```
def DWTImpl(x, nres, f, symm=True, dual=False):
    for res in range(nres):
        f(x[0::2**res], symm, dual)
    reorganize_coefficients(x, nres, True)
```

Again note that the code is applied to all columns if the data is two-dimensional. Note also that here the kernel function `f` is first invoked, one time for each resolution. Finally, the coefficients are reorganized so that the $\boldsymbol{\phi}_m$ coordinates come first, followed by the coordinates from the different levels. We have provided a function `reorganize_coefficients` which does this reorganization, and you will be spared the details in this implementation. In Exercise 5.25 we go through some aspects of this implementation. Note that, although the DWT requires this reorganization, this reorganization may not be required in practice, as further processing is needed, for which the coefficients can be accessed where they have been placed after the in-place operations. Note also the two last arguments, `symm` and `dual`, which we have not commented on. We will return to these in Chapter 6. This implementation is not recursive, as the for-loop runs through the different stages. Inside the loop we perform the change of coordinates from $\boldsymbol{\phi}_k$ to $(\boldsymbol{\phi}_{k-1}, \boldsymbol{\psi}_{k-1})$ by applying Equation (5.19). This works on the first coordinates, since the coordinates from $\boldsymbol{\phi}_k$ are stored first in

$$(\boldsymbol{\phi}_k, \boldsymbol{\psi}_k, \boldsymbol{\psi}_{k+1}, \cdots, \boldsymbol{\psi}_{m-2}, \boldsymbol{\psi}_{m-1}).$$

Finally, the $\boldsymbol{c}$-coordinates are stored before the $\boldsymbol{w}$-coordinates. In this implementation, note that the first levels require the most multiplications, since the latter levels leave an increasing part of the coordinates unchanged. Note also that the change of coordinates matrix is a very sparse matrix: At each level a coordinate can be computed from only two of the other coordinates, so that this matrix has only two nonzero elements in each row/column. The algorithm clearly shows that there is no need to perform a full matrix multiplication to perform the change of coordinates.

The corresponding function for the IDWT, called IDWTImpl, goes as follows:

```
def IDWTImpl(x, nres, f, symm=True, dual=False):
    reorganize_coefficients(x, nres, False)
    for res in range(nres - 1, -1, -1):
        f(x[0::2**res], symm, dual)
```

Here the steps are simply performed in the reverse order, and by iterating Equation (5.20). You may be puzzled by the names `DWTKernelHaar` and `IDWTKernelHaar`. In the next sections we will consider other cases, where the underlying function ϕ may be a different function, not necessarily piecewise constant. It will turn out that much of the analysis we have done makes sense for other functions ϕ as well, giving rise to other structures which we also will refer to as wavelets. The wavelet resulting from piecewise constant functions is thus simply one example out of many, and it is commonly referred to as the *Haar wavelet*.

Let us round off this section with some important examples.

Example 5.18. *Computing the DWT by hand.*

In some cases, the DWT can be computed by hand, keeping in mind its definition as a change of coordinates. As an example, consider the simple vector $\boldsymbol{x}$ of length $2^{10} = 1024$ defined by

$$x_n = \begin{cases} 1 & \text{for } n < 512 \\ 0 & \text{for } n \geq 512, \end{cases}$$

and let us compute the 10-level DWT of this vector by first visualizing the function with these coordinates. Since $m = 10$ here, we should view $\boldsymbol{x}$ as coordinates in the basis $\boldsymbol{\phi}_{10}$ of a function $f(t) \in V_{10}$. This is $f(t) = \sum_{n=0}^{511} \phi_{10,n}$, and since $\phi_{10,n}$ is supported on $[2^{-10}n, 2^{-10}(n+1))$, the support of f has width $512 \times 2^{-10} = 1/2$ (512 translates, each with width 2^{-10}). Moreover, since $\phi_{10,n}$ is $2^{10/2} = 2^5 = 32$ on $[2^{-10}n, 2^{-10}(n+1))$ and 0 elsewhere, it is clear that

$$f(t) = \begin{cases} 32 & \text{for } 0 \leq t < 1/2 \\ 0 & \text{for } 0t \geq 1/2. \end{cases}$$

This is by definition a function in V_1: f must in fact be a multiplum of $\phi_{1,0}$, since this also is supported on $[0, 1/2)$. We can thus write $f(t) = c\phi_{1,0}(t)$ for some c. We can find c by setting $t = 0$. This gives that $32 = 2^{1/2}c$ (since $f(0) = 32$,

$\phi_{1,0}(0) = 2^{1/2}$), so that $c = 32/\sqrt{2}$. This means that $f(t) = \frac{32}{\sqrt{2}}\phi_{1,0}(t)$, f is in V_1, and with coordinates $(32/\sqrt{2}, 0, \ldots, 0)$ in $\boldsymbol{\phi}_1$.

When we run a 10-level DWT we make a change of coordinates from $\boldsymbol{\phi}_{10}$ to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \cdots, \boldsymbol{\psi}_9)$. The first 9 levels give us the coordinates in $(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1, \boldsymbol{\psi}_2, \ldots, \boldsymbol{\psi}_9)$, and these are $(32/\sqrt{2}, 0, \ldots, 0)$ from what we showed. It remains thus only to perform the last level in the DWT, i.e. perform the change of coordinates from $\boldsymbol{\phi}_1$ to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$. Since $\phi_{1,0} = \frac{1}{\sqrt{2}}(\phi_{0,0} + \psi_{0,0})$, so that we get

$$f(t) = \frac{32}{\sqrt{2}}\phi_{1,0}(t) = \frac{32}{\sqrt{2}}\frac{1}{\sqrt{2}}(\phi_{0,0} + \psi_{0,0}) = 16\phi_{0,0} + 16\psi_{0,0}.$$

From this we see that the coordinate vector of f in $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \cdots, \boldsymbol{\psi}_9)$, i.e. the 10-level DWT of $\boldsymbol{x}$, is $(16, 16, 0, 0, \ldots, 0)$. Note that here V_0 and W_0 are both 1-dimensional, since V_{10} was assumed to be of dimension 2^{10} (in particular, $N = 1$).

It is straightforward to verify what we found using the algorithm above:

```
x = hstack([ones(512), zeros(512)])
DWTImpl(x, 10, DWTKernelHaar)
print x
```

The reason why the method from this example worked was that the vector we started with had a simple representation in the wavelet basis, actually it equaled the coordinates of a basis function in $\boldsymbol{\phi}_1$. Usually this is not the case, and our only possibility then is to run the DWT on a computer.

Example 5.19. *DWT and sound.*

When you run a DWT you may be led to believe that coefficients from the lower order resolution spaces may correspond to lower frequencies. This sounds reasonable, since the functions $\phi(2^m t - n) \in V_m$ change more quickly than $\phi(t - n) \in V_0$. However, the functions $\phi_{m,n}$ do not correspond to pure tones in the setting of wavelets. But we can still listen to sound from the different resolution spaces. In Exercise 5.19 you will be asked to implement a function which runs an m-level DWT on the first samples of the sound file `castanets.wav`, extracts the coefficients from the lower order resolution spaces or the detail spaces, transforms the values back to sound samples with the IDWT, and plays the result. When you listen to the result the sound is clearly recognizable for lower values of m, but is degraded for higher values of m. The explanation is that too much of the detail is omitted when you use a higher m. To be more precise, when listening to the sound by throwing away everything from the detail spaces $W_0, W_1, \ldots, W_{m-1}$, we are left with a 2^{-m} share of the data. Note that this procedure is mathematically not the same as setting some DFT coefficients to zero, since the DWT does not operate on pure tones.

It is of interest to plot the samples of our test audio file `castanets.wav`, and compare it with the first order DWT coefficients of the same samples. This is shown in Figure 5.10. The first half part of the plot represents the low-resolution approximation of the sound, the second half part represents the detail/error.

We see that the detail is quite significant in this case. This means that the first order wavelet approximation does not give a very good approximation to the sound. In the exercises we will experiment more on this.

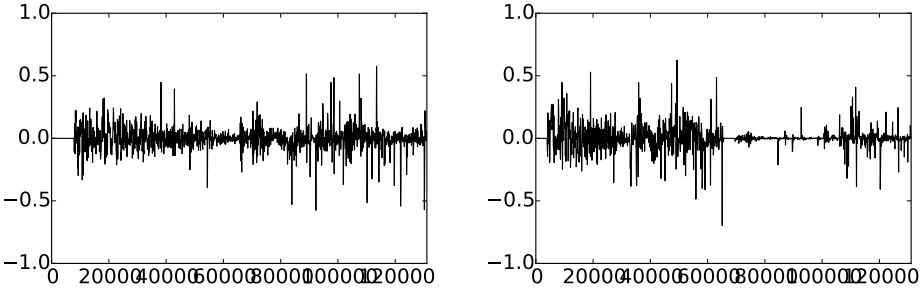

Figure 5.10: The 2^17 first sound samples (left) and the DWT coefficients (right) of the sound `castanets.wav`.

It is also interesting to plot only the detail/error in the sound, for different resolutions. For this, we must perform a DWT so that we get a representation in the basis $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_{m-1})$, set the coefficients from V_0 to zero, and transform back with the IDWT. In figure 5.11 the error is shown for the test audio file `castanets.wav` for $m = 1$, $m = 2$. This clearly shows that the error is larger when two levels of the DWT are performed, as one would suspect. It is also seen that the error is larger in the part of the file where there are bigger variations. This also sounds reasonable.

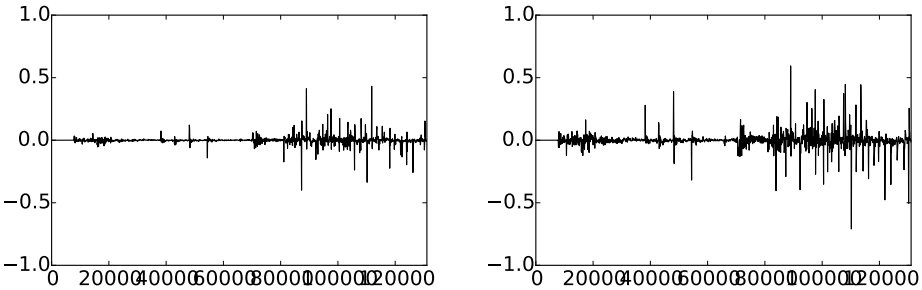

Figure 5.11: The error (i.e. the contribution from $W_0 \oplus W_1 \oplus \cdots \oplus W_{m-1}$) in the sound file `castanets.wav`, for $m = 1$ and $m = 2$, respectively.

The previous example illustrates that wavelets as well may be used to perform operations on sound. As we will see later, however, our main application for wavelets will be images, where they have found a more important role than for sound. Images typically display variations which are less abrupt than the ones found in sound. Just as the functions above had smaller errors in the corresponding resolution spaces than the sound had, images are thus more suited for for use with wavelets. The main idea behind why wavelets are so useful

comes from the fact that the detail, i.e., wavelet coefficients corresponding to the spaces W_k, are often very small. After a DWT one is therefore often left with a couple of significant coefficients, while most of the coefficients are small. The approximation from V_0 can be viewed as a good approximation, even though it contains much less information. This gives another reason why wavelets are popular for images: Detailed images can be very large, but when they are downloaded to a web browser, the browser can very early show a low-resolution of the image, while waiting for the rest of the details in the image to be downloaded. When we later look at how wavelets are applied to images, we will need to handle one final hurdle, namely that images are two-dimensional.

Example 5.20. *DWT on the samples of a mathematical function.*

Above we plotted the DWT coefficients of a sound, as well as the detail/error. We can also experiment with samples generated from a mathematical function. Figure 5.12 plots the error for different functions, with $N = 1024$.

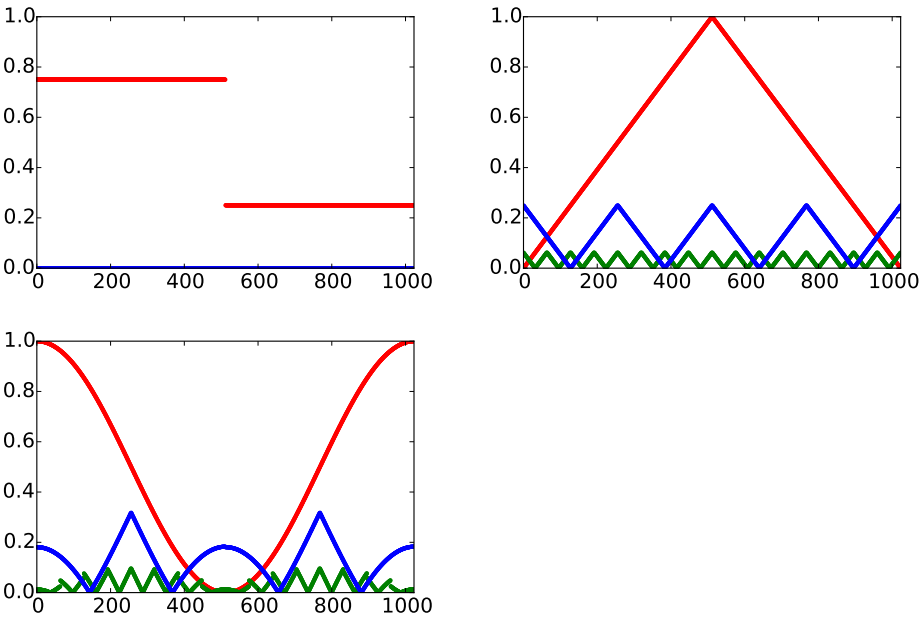

Figure 5.12: The error (i.e. the contribution from $W_0 \oplus W_1 \oplus \cdots \oplus W_{m-1}$) for $N = 1024$ when f is a square wave, the linear function $f(t) = 1 - 2|1/2 - t/N|$, and the trigonometric function $f(t) = 1/2 + \cos(2\pi t/N)/2$, respectively. The detail is indicated for $m = 6$ and $m = 8$.

In these cases, we see that we require large m before the detail/error becomes significant. We see also that there is no error for the square wave. The reason is that the square wave is a piecewise constant function, so that it can be represented exactly by the ϕ-functions. For the other functions, however, this is not the case, so we here get an error.

Above we used the functions `DWTImpl`, `IDWTImpl` to plot the error. For the functions we plotted in the previous example it is also possible to compute the wavelet coefficients, which we previously have denoted by $w_{m,n}$, exactly. You will be asked to do this in exercises 5.21 and 5.22. The following example shows the general procedure which can be used for this:

Example 5.21. *Computing the wavelet coefficients.*

Let us consider the function $f(t) = 1 - t/N$. This function decreases linearly from 1 to 0 on $[0, N]$, so that it is not piecewise constant, and does not lie in any of the spaces V_m. We can instead consider $\text{proj}_{V_m} f \in V_m$, and apply the DWT to this. Let us compute the $\boldsymbol{\psi}_m$-coordinates $w_{m,n}$ of $\text{proj}_{V_m} f$ in the orthonormal basis $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_{m-1})$. The orthogonal decomposition theorem says that

$$w_{m,n} = \langle f, \psi_{m,n} \rangle = \int_0^N f(t)\psi_{m,n}(t)dt = \int_0^N (1 - t/N)\psi_{m,n}(t)dt.$$

Using the definition of $\psi_{m,n}$ we see that this can also be written as

$$2^{m/2} \int_0^N (1 - t/N)\psi(2^m t - n)dt = 2^{m/2} \left(\int_0^N \psi(2^m t - n)dt - \int_0^N \frac{t}{N}\psi(2^m t - n)dt \right).$$

Using Observation 5.14 we get that $\int_0^N \psi(2^m t - n)dt = 0$, so that the first term above vanishes. Moreover, $\psi_{m,n}$ is nonzero only on $[2^{-m}n, 2^{-m}(n+1))$, and is 1 on $[2^{-m}n, 2^{-m}(n+1/2))$, and -1 on $[2^{-m}(n+1/2), 2^{-m}(n+1))$. We therefore get

$$\begin{aligned}
w_{m,n} &= -2^{m/2} \left(\int_{2^{-m}n}^{2^{-m}(n+1/2)} \frac{t}{N}dt - \int_{2^{-m}(n+1/2)}^{2^{-m}(n+1)} \frac{t}{N}dt \right) \\
&= -2^{m/2} \left(\left[\frac{t^2}{2N} \right]_{2^{-m}n}^{2^{-m}(n+1/2)} - \left[\frac{t^2}{2N} \right]_{2^{-m}(n+1/2)}^{2^{-m}(n+1)} \right) \\
&= -2^{m/2} \left(\left(\frac{2^{-2m}(n+1/2)^2}{2N} - \frac{2^{-2m}n^2}{2N} \right) - \left(\frac{2^{-2m}(n+1)^2}{2N} - \frac{2^{-2m}(n+1/2)^2}{2N} \right) \right) \\
&= -2^{m/2} \left(-\frac{2^{-2m}n^2}{2N} + \frac{2^{-2m}(n+1/2)^2}{N} - \frac{2^{-2m}(n+1)^2}{2N} \right) \\
&= -\frac{2^{-3m/2}}{2N} \left(-n^2 + 2(n+1/2)^2 - (n+1)^2 \right) = \frac{1}{N2^{2+3m/2}}.
\end{aligned}$$

We see in particular that $w_{m,n} \to 0$ when $m \to \infty$. Also, all coordinates were equal, i.e. $w_{m,0} = w_{m,1} = w_{m,2} = \cdots$. It is not too hard to convince oneself that this equality has to do with the fact that f is linear. We see also that there were a lot of computations even in this very simple example. For most functions we therefore usually do not compute $w_{m,n}$ symbolically, but instead run implementations like `DWTImpl`, `IDWTImpl` on a computer.

What you should have learned in this section.

- Definition of the m-level Discrete Wavelet Transform.
- Implementation of the Haar wavelet transform and its inverse.
- Experimentation with wavelets on sound.

Exercise 5.9: Implement IDWT for The Haar wavelet

Write a function `IDWTKernelHaar` which uses the formulas (5.24) in the compendium to implement the IDWT, similarly to how the function `DWTKernelHaar` implemented the DWT using the formulas (5.25) in the compendium.

Exercise 5.10: Computing projections

Generalize Exercise 5.4 to the projections from V_{m+1} onto V_m and W_m.

Exercise 5.11: Scaling a function

Show that $f(t) \in V_m$ if and only if $g(t) = f(2t) \in V_{m+1}$.

Exercise 5.12: Direct sums

Let $C_1, C_2 \ldots, C_n$ be independent vector spaces, and let $T_i : C_i \to C_i$ be linear transformations. The direct sum of T_1, T_2,...,T_n, written as $T_1 \oplus T_2 \oplus \ldots \oplus T_n$, denotes the linear transformation from $C_1 \oplus C_2 \oplus \cdots \oplus C_n$ to itself defined by

$$T_1 \oplus T_2 \oplus \ldots \oplus T_n(\boldsymbol{c}_1 + \boldsymbol{c}_2 + \cdots + \boldsymbol{c}_n) = T_1(\boldsymbol{c}_1) + T_2(\boldsymbol{c}_2) + \cdots + T_n(\boldsymbol{c}_n)$$

when $\boldsymbol{c}_1 \in C_1$, $\boldsymbol{c}_2 \in C_2$, ..., $\boldsymbol{c}_n \in C_n$. Similarly, when $A_1, A_2, \ldots, A_n$ are square matrices, $A_1 \oplus A_2 \oplus \cdots \oplus A_n$ is defined as the block matrix where the blocks along the diagonal are $A_1, A_2, \ldots, A_n$, and where all other blocks are 0. Show that, if $\mathcal{B}_i$ is a basis for C_i then

$$[T_1 \oplus T_2 \oplus \ldots \oplus T_n]_{(\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n)} = [T_1]_{\mathcal{B}_1} \oplus [T_2]_{\mathcal{B}_2} \oplus \cdots \oplus [T_n]_{\mathcal{B}_n},$$

Here two new concepts are used: a direct sum of matrices, and a direct sum of linear transformations.

Exercise 5.13: Eigenvectors of direct sums

Assume that T_1 and T_2 are matrices, and that the eigenvalues of T_1 are equal to those of T_2. What are the eigenvalues of $T_1 \oplus T_2$? Can you express the eigenvectors of $T_1 \oplus T_2$ in terms of those of T_1 and T_2?

Exercise 5.14: Invertibility of direct sums

Assume that A and B are square matrices which are invertible. Show that $A \oplus B$ is invertible, and that $(A \oplus B)^{-1} = A^{-1} \oplus B^{-1}$.

Exercise 5.15: Multiplying direct sums

Let A, B, C, D be square matrices of the same dimensions. Show that $(A \oplus B)(C \oplus D) = (AC) \oplus (BD)$.

Exercise 5.16: Finding N

Assume that you run an m-level DWT on a vector of length r. What value of N does this correspond to? Note that an m-level DWT performs a change of coordinates from $\boldsymbol{\phi}_m$ to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_{m-2}, \boldsymbol{\psi}_{m-1})$.

Exercise 5.17: Different DWTs for similar vectors

In Figure 5.13 we have plotted the DWT's of two vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. In both vectors we have 16 ones followed by 16 zeros, and this pattern repeats cyclically so that the length of both vectors is 256. The only difference is that the second vector is obtained by delaying the first vector with one element.

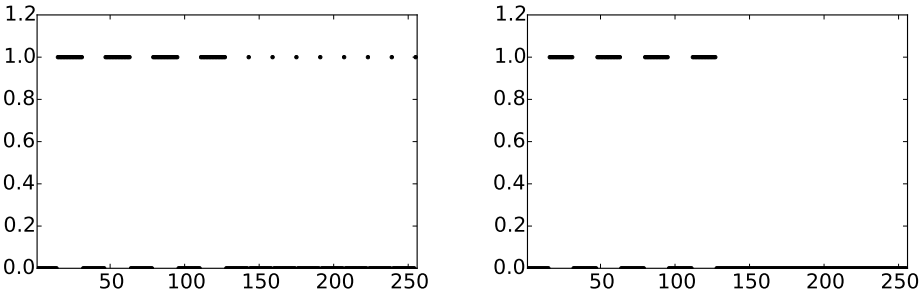

Figure 5.13: 2 vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ which seem equal, but where the DWT's are very different.

You see that the two DWT's are very different: For the first vector we see that there is much detail present (the second part of the plot), while for the second vector there is no detail present. Attempt to explain why this is the case. Based on your answer, also attempt to explain what can happen if you change the point of discontinuity for the piecewise constant function in Figure 5.20(a) to something else.

Exercise 5.18: Plotting the DWT on a sound

Run a 2-level DWT on the first 2^{17} sound samples of the audio file `castanets.wav`, and plot the values of the resulting DWT-coefficients. Compare the values of the coefficients from V_0 with those from W_0 and W_1.

Exercise 5.19: Zeroing out DWT coefficients

In this exercise we will experiment with applying an m-level DWT to a sound file.

a) Write a function `playDWT` which takes m, a DWT kernel `f`, an IDWT kernel `invf`, and a variable `lowres` as input, and

- reads the audio file `castanets.wav`,
- performs an m-level DWT to the first 2^{17} sound samples of $\boldsymbol{x}$ using the function `DWTImpl` with DWT kernel `f`,
- sets all wavelet coefficients representing detail to zero if `lowres` is true (i.e. keep only the coordinates from $\boldsymbol{\phi}_0$ in the basis $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_{m-2}, \boldsymbol{\psi}_{m-1})$),
- sets all low-resolution coefficients to zero if `lowres` is false (i.e. zero out the coordinates from $\boldsymbol{\phi}_0$ and keep the others),
- performs an IDWT on the resulting coefficients using the function `IDWTImpl` with IDWT kernel `invf`,
- plays the resulting sound.

b) Do the sound samples returned by `playDWT` lie in $[-1, 1]$?

c) Run the function `playDWT` with `DWTKernelHaar` and `IDWTKernelHaar` as inputs, and for different values of m, with 'lowres' set to true (i.e. with the low-resolution approximation chosen). For which m can you hear that the sound gets degraded? How does it get degraded? Compare with what you heard through the function `playDFT` in Example 2.27, where you performed a DFT on the sound sample instead, and set some of the DFT coefficients to zero.

d) Repeat the listening experiment from c., but this time with `lowres` set to false (i.e. keep only the detail from W_0, $W_1, \ldots$. What kind of sound do you hear? Can you recognize the original sound in what you hear?

Exercise 5.20: Construct a sound

Attempt to construct a (nonzero) sound where the function `playDWT` form the previous exercise does not change the sound for $m = 1, 2$.

Exercise 5.21: Exact computation of wavelet coefficients 1

Compute the wavelet detail coefficients analytically for the functions in Example 5.20, i.e. compute the quantities $w_{m,n} = \int_0^N f(t)\psi_{m,n}(t)dt$ similarly to how this was done in Example 5.21.

Exercise 5.22: Exact compution of wavelet coefficients 2

Compute the wavelet detail coefficients analytically for the functions $f(t) = \left(\frac{t}{N}\right)^k$, i.e. compute the quantities $w_{m,n} = \int_0^N \left(\frac{t}{N}\right)^k \psi_{m,n}(t)dt$ similarly to how this was done in Example 5.21. How do these compare with the coefficients from the Exercise 5.21?

Exercise 5.23: Computing the DWT of a simple vector

Suppose that we have the vector $\boldsymbol{x}$ with length $2^{10} = 1024$, defined by $x_n = 1$ for n even, $x_n = -1$ for n odd. What will be the result if you run a 10-level DWT on $\boldsymbol{x}$? Use the function `DWTImpl` to verify what you have found.

Hint. We defined ψ by $\psi(t) = (\phi_{1,0}(t) - \phi_{1,1}(t))/\sqrt{2}$. From this connection it follows that $\psi_{9,n} = (\phi_{10,2n} - \phi_{10,2n+1})/\sqrt{2}$, and thus $\phi_{10,2n} - \phi_{10,2n+1} = \sqrt{2}\psi_{9,n}$. Try to couple this identity with the alternating sign you see in $\boldsymbol{x}$.

Exercise 5.24: The Haar wavelet when N is odd

Use the results from Exercise 5.8 to rewrite the implementations `DWTKernelHaar` and `IDWTKernelHaar` so that they also work in the case when N is odd.

Exercise 5.25: in-place DWT

Show that the coordinates in $\boldsymbol{\phi}_m$ after an in-place m-level DWT end up at indices $k2^m$, $k = 0, 1, 2, \ldots$. Show similarly that the coordinates in $\boldsymbol{\psi}_m$ after an in-place m-level DWT end up at indices $2^{m-1} + k2^m$, $k = 0, 1, 2, \ldots$. Find these indices in the code for the function `reorganize_coefficients`.

5.4 A wavelet based on piecewise linear functions

Unfortutately, piecewise constant functions are too simple to provide good approximations. In this section we are going to extend the construction of wavelets to piecewise linear functions. The advantage is that piecewise linear functions are better for approximating smooth functions and data than piecewise constants, which should translate into smaller components (errors) in the detail spaces in many practical situations. As an example, this would be useful if we are interested in compression. In this new setting it turns out that we loose the

orthonormality we had for the Haar wavelet. On the other hand, we will see that the new scaling functions and mother wavelets are symmetric functions. We will later see that this implies that the corresponding DWT and IDWT have simple implementations with higher precision. Our experience from deriving Haar wavelets will guide us in the construction of piecewise linear wavelets. The first task is to define the new resolution spaces.

Definition 5.22. *Resolution spaces of piecewise linear functions.*

The space V_m is the subspace of continuous functions on $\mathbb{R}$ which are periodic with period N, and linear on each subinterval of the form $[n2^{-m}, (n+1)2^{-m})$.

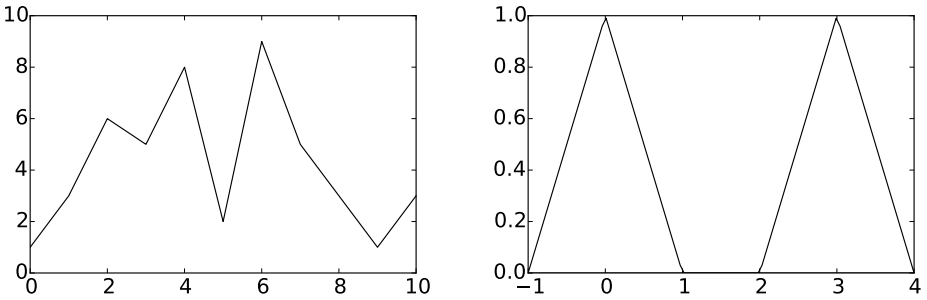

Figure 5.14: A piecewise linear function and the two functions $\phi(t)$ and $\phi(t-3)$.

Any $f \in V_m$ is uniquely determined by its values in the points $\{2_{-m}n\}_{n=0}^{2^m N-1}$. The linear mapping which sends f to these samples is thus an isomorphism from V_m onto R^{N2^m}, so that the dimension of V_m is $\mathbb{N}2^m$. The lft plot in Figure 5.14 shows an example of a piecewise linear function in V_0 on the interval $[0, 10]$. We note that a piecewise linear function in V_0 is completely determined by its value at the integers, so the functions that are 1 at one integer and 0 at all others are particularly simple and therefore interesting, see the right plot in Figure 5.14. These simple functions are all translates of each other and can therefore be built from one scaling function, as is required for a multiresolution analysis.

Lemma 5.23. *The function ϕ.*

Let the function ϕ be defined by

$$\phi(t) = \begin{cases} 1 - |t|, & \text{if } -1 \le t \le 1; \\ 0, & \text{otherwise;} \end{cases} \tag{5.26}$$

and for any $m \ge 0$ set

$$\phi_{m,n}(t) = 2^{m/2}\phi(2^m t - n) \quad \text{for } n = 0,\, 1,\, \ldots,\, 2^m N - 1,$$

and $\boldsymbol{\phi}_m = \{\phi_{m,n}\}_{n=0}^{2^m N-1}$. $\boldsymbol{\phi}_m$ is a basis for V_m, and $\phi_{0,n}(t)$ is the function in V_0 with smallest support that is nonzero at $t = n$.

Proof. It is clear that $\phi_{m,n} \in V_m$, and

$$\phi_{m,n'}(n2^{-m}) = 2^{m/2}\phi(2^m(2^{-m}n) - n') = 2^{m/2}\phi(n - n').$$

Since ϕ is zero at all nonzero integers, and $\phi(0) = 1$, we see that $\phi_{m,n'}(n2^{-m}) = 2^{m/2}$ when $n' = n$, and 0 if $n' \neq n$. Let $L_m : V_m \to R^{N2^m}$ be the isomorphism mentioned above which sends $f \in V_m$ to the samples in the points $\{2_{-m}n\}_{n=0}^{2^m N-1}$. Our calculation shows that $L_m(\phi_{m,n}) = 2^{m/2}e_n$. Since L_m is an isomorphism it follows that $\boldsymbol{\phi}_m = \{\phi_{m,n}\}_{n=0}^{2^m N-1}$ is a basis for V_m.

Suppose that the function $g \in V_0$ has smaller support than $\phi_{0,n}$, but is nonzero at $t = n$. We must have that $L_0(g) = c\boldsymbol{e}_n$ for some c, since g is zero on the integers different from n. But then g is a multiple of $\phi_{0,n}$, so that it is the function in V_0 with smallest support that is nonzero at $t = n$. □

The function ϕ and its translates and dilates are often referred to as hat functions for obvious reasons. Note that the new function ϕ is nonzero for small negative x-values, contrary to the ϕ we defined in Chapter 5. If we plotted the function on $[0, N)$, we would see the nonzero parts at the beginning and end of this interval, due to the period N, but we will mostly plot on an interval around zero, since such an interval captures the entire support of the function. Also for the piecewise linear wavelet the coordinates of a basis function is given by the samples:

Lemma 5.24. *Writing in terms of the samples.*
A function $f \in V_m$ may be written as

$$f(t) = \sum_{n=0}^{2^m N-1} f(n/2^m)2^{-m/2}\phi_{m,n}(t). \tag{5.27}$$

An essential property also here is that the spaces are nested.

Lemma 5.25. *Resolution spaces are nested.*
The piecewise linear resolution spaces are nested,

$$V_0 \subset V_1 \subset \cdots \subset V_m \subset \cdots .$$

Proof. We only need to prove that $V_0 \subset V_1$ since the other inclusions are similar. But this is immediate since any function in V_0 is continuous, and linear on any subinterval in the form $[n/2, (n + 1)/2)$. □

In the piecewise constant case, we saw in Lemma 5.5 that the scaling functions were automatically orthogonal since their supports did not overlap. This is not the case in the linear case, but we could orthogonalise the basis $\boldsymbol{\phi}_m$ with the Gram-Schmidt process from linear algebra. The disadvantage is that we lose the nice local behaviour of the scaling functions and end up with basis functions that are nonzero over all of $[0, N]$. And for most applications, orthogonality is not essential; we just need a basis. The next step in the derivation of wavelets is to find formulas that let us express a function given in the basis $\boldsymbol{\phi}_0$ for V_0 in terms of the basis $\boldsymbol{\phi}_1$ for V_1.

Lemma 5.26. *The two-scale equation.*
The functions $\phi_{0,n}$ satisfy the relation

$$\phi_{0,n} = \frac{1}{\sqrt{2}}\left(\frac{1}{2}\phi_{1,2n-1} + \phi_{1,2n} + \frac{1}{2}\phi_{1,2n+1}\right). \tag{5.28}$$

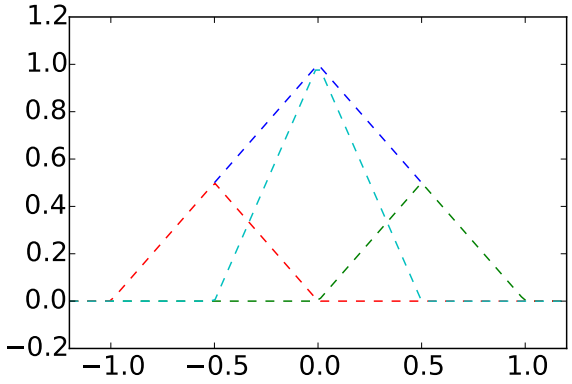

Figure 5.15: How $\phi(t)$ can be decomposed as a linear combination of $\phi_{1,-1}$, $\phi_{1,0}$, and $\phi_{1,1}$.

Proof. Since $\phi_{0,n}$ is in V_0 it may be expressed in the basis $\boldsymbol{\phi}_1$ with formula (5.27),

$$\phi_{0,n}(t) = 2^{-1/2}\sum_{k=0}^{2N-1}\phi_{0,n}(k/2)\phi_{1,k}(t).$$

The relation (5.28) now follows since

$$\phi_{0,n}\big((2n-1)/2\big) = \phi_{0,n}\big((2n+1)/2\big) = 1/2, \quad \phi_{0,n}(2n/2) = 1,$$

and $\phi_{0,n}(k/2) = 0$ for all other values of k. □

The relationship given by Equation (5.28) is shown in Figure 5.15.

5.4.1 Detail spaces and wavelets

The next step in our derivation of wavelets for piecewise linear functions is the definition of the detail spaces. We need to determine a space W_0 that is linearly independent from V_0, and so that $V_1 = V_0 \oplus W_0$. In the case of piecewise constant functions we started with a function g_1 in V_1, computed the least squares approximation g_0 in V_0, and then defined the error function $e_0 = g_1 - g_0$, with $e_0 \in W_0$ and W_0 as the orthogonal complement of V_0 in V_1.

It turns out that this strategy is less appealing in the case of piecewise linear functions. The reason is that the functions $\phi_{0,n}$ are not orthogonal anymore (see Exercise 5.26). Due to this we have no simple, orthogonal basis for the set of piecewise linear functions, so that the orthogonal decomposition theorem fails to give us the projection onto V_0 in a simple way. It is therefore no reason to use the orthogonal complement of V_0 in V_1 as our error space, since it is hard to write a piecewise linear function as a sum of two other piecewise linear functions which are orthogonal. Instead of using projections to find low-resolution approximations, and orthogonal complements to find error functions, we will attempt the following simple approximation method:

Definition 5.27. *Alternative projection.*
Let g_1 be a function in V_1 given by

$$g_1 = \sum_{n=0}^{2N-1} c_{1,n}\phi_{1,n}. \tag{5.29}$$

The approximation $g_0 = P(g_1)$ in V_0 is defined as the unique function in V_0 which has the same values as g_1 at the integers, i.e.

$$g_0(n) = g_1(n), \quad n = 0, 1, \ldots, N-1. \tag{5.30}$$

It is easy to show that $P(g_1)$ actually is different from the projection of g_1 onto V_0: If $g_1 = \phi_{1,1}$, then g_1 is zero at the integers, and then clearly $P(g_1) = 0$. But in Exercise 5.27 you will be asked to compute the projection onto V_0 using different means than the orthogonal decomposition theorem, and the result will be seen to be nonzero. It is also very easy to see that the coordinates of g_0 in $\boldsymbol{\phi}_0$ can be obtained by dropping every second coordinate of g_0 in $\boldsymbol{\phi}_1$. To be more precise, the following holds:

Lemma 5.28. *Expression for the alternative projection.*
We have that

$$P(\phi_{1,n}) = \begin{cases} \sqrt{2}\phi_{0,n/2}, & \text{if } n \text{ is an even integer;} \\ 0, & \text{otherwise.} \end{cases}$$

Once this approximation method is determined, it is straightforward to determine the detail space as the space of error functions.

Lemma 5.29. *Resolution spaces.*
Define

$$W_0 = \{f \in V_1 \mid f(n) = 0, \quad \text{for } n = 0, 1, \ldots, N-1\},$$

and

$$\psi(t) = \frac{1}{\sqrt{2}}\phi_{1,1}(t) \qquad \psi_{m,n}(t) = 2^{m/2}\psi(2^m t - n). \tag{5.31}$$

Suppose that $g_1 \in V_1$ and that $g_0 = P(g_1)$. Then

- the error $e_0 = g_1 - g_0$ lies in W_0,
- $\boldsymbol{\psi}_0 = \{\psi_{0,n}\}_{n=0}^{N-1}$ is a basis for W_0.
- V_0 and W_0 are linearly independent, and $V_1 = V_0 \oplus W_0$.

Proof. Since $g_0(n) = g_1(n)$ for all integers n, $e_0(n) = (g_1 - g_0)(n) = 0$, so that $e_0 \in W_0$. This proves the first statement.

For the second statement, note first that

$$\psi_{0,n}(t) = \psi(t-n) = \frac{1}{\sqrt{2}}\phi_{1,1}(t-n) = \phi(2(t-n)-1) = \phi(2t-(2n+1)) = \frac{1}{\sqrt{2}}\phi_{1,2n+1}(t). \tag{5.32}$$

$\boldsymbol{\psi}_0$ is thus a linearly independent set of dimension N, since it corresponds to a subset of $\boldsymbol{\phi}_1$. Since $\phi_{1,2n+1}$ is nonzero only on $(n, n+1)$, it follows that all of $\boldsymbol{\psi}_0$ lies in W_0. Clearly then $\boldsymbol{\psi}_0$ is also a basis for W_0, since W_0 also has dimension N (its image under L_1 consists of points where every second component is zero).

Consider finally a linear combination from $\boldsymbol{\phi}_0$ and $\boldsymbol{\psi}_0$ which gives zero:

$$\sum_{n=0}^{N-1} a_n \phi_{0,n} + \sum_{n=0}^{N-1} b_n \psi_{0,n} = 0.$$

If we evaluate this at $t = k$, we see that $\psi_{0,n}(k) = 0$, $\phi_{0,n}(k) = 0$ when $n \neq k$, and $\phi_{0,k}(k) = 1$. When we evaluate at k we thus get a_k, which must be zero. If we then evaluate at $t = k + 1/2$ we get in a similar way that all $b_n = 0$, and it follows that V_0 and W_0 are linearly independent. That $V_1 = V_0 \oplus W_0$ follows from the fact that V_1 has dimension $2N$, and V_0 and W_0 both have dimension N. □

We can define W_m in a similar way for $m > 0$, and generalize the lemma to W_m. We can thus state the following analog to Theorem 5.16 for writing $g_m \in V_m$ as a sum of a low-resolution approximation $g_{m-1} \in V_{m-1}$, and a detail/error component $e_{m-1} \in W_{m-1}$.

Theorem 5.30. *Decomposing V_m.*

The space V_m can be decomposed as the direct sum $V_m = V_{m-1} \oplus W_{m-1}$ where

$$W_{m-1} = \{f \in V_m \mid f(n/2^{m-1}) = 0, \quad \text{for } n = 0, 1, \ldots, 2^{m-1}N - 1\}.$$

W_m has the base $\boldsymbol{\psi}_m = \{\psi_{m,n}\}_{n=0}^{2^m N-1}$, and V_m has the two bases

$$\boldsymbol{\phi}_m = \{\phi_{m,n}\}_{n=0}^{2^m N-1}, \text{ and } (\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1}) = \big(\{\phi_{m-1,n}\}_{n=0}^{2^{m-1}N-1}, \{\psi_{m-1,n}\}_{n=0}^{2^{m-1}N-1}\big).$$

With this result we can define the DWT and the IDWT with their stages as before, but the matrices thesemselves are now different. For the IDWT (i.e. $P_{\boldsymbol{\phi}_1 \leftarrow (\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)}$), the columns in the matrix can be found from equations (5.28) and (5.32), i.e.

$$\begin{aligned} \phi_{0,n} &= \frac{1}{\sqrt{2}} \left(\frac{1}{2}\phi_{1,2n-1} + \phi_{1,2n} + \frac{1}{2}\phi_{1,2n+1} \right) \\ \psi_{0,n} &= \frac{1}{\sqrt{2}}\phi_{1,2n+1}. \end{aligned} \tag{5.33}$$

For the DWT we can find the columns in the matrix by rewriting these equations to

$$\begin{aligned} \frac{1}{\sqrt{2}}\phi_{1,2n} &= \phi_{0,n} - \frac{1}{2\sqrt{2}}\phi_{1,2n-1} - \frac{1}{2\sqrt{2}}\phi_{1,2n+1} \\ \frac{1}{\sqrt{2}}\phi_{1,2n+1} &= \psi_{0,n}, \end{aligned}$$

so that

$$\phi_{1,2n} = \sqrt{2}\phi_{0,n} - \frac{1}{2}\phi_{1,2n-1} - \frac{1}{2}\phi_{1,2n+1} = -\frac{\sqrt{2}}{2}\psi_{0,n-1} + \sqrt{2}\phi_{0,n} - \frac{\sqrt{2}}{2}\psi_{0,n} \tag{5.34}$$

$$\phi_{1,2n+1} = \sqrt{2}\psi_{0,n}. \tag{5.35}$$

Example 5.31. *DWT on sound.*

Later we will write algorithms which performs the DWT/IDWT for the piecewise linear wavelet, similarly to how we implemented the Haar wavelet transformation in the previous chapter. This gives us new kernel transformations, which we will call `DWTKernelpwl0`, `IDWTKernelpwl0` (The 0 stands for 0 vanishing moments. We defined vanishing moments after Observation 5.14. We will have more to say about vanishing moments later). Using these new kernels, let us plot the detail/error in the test audio file `castanets.wav` for different resolutions, as we did in Example 5.19. The result is shown in Figure 5.16. When comparing with Figure 5.11 we see much of the same, but it seems here that the error is bigger than before. In the next section we will try to explain why this is the case, and construct another wavelet based on piecewise linear functions which remedies this.

Example 5.32. *DWT on the samples of a mathematical function.*

Let us also repeat Exercise 5.20, where we plotted the detail/error at different resolutions, for the samples of a mathematical function. Figure 5.17 shows the new plot.

With the square wave we see now that there is an error. The reason is that a piecewise constant function can not be represented exactly by piecewise linear

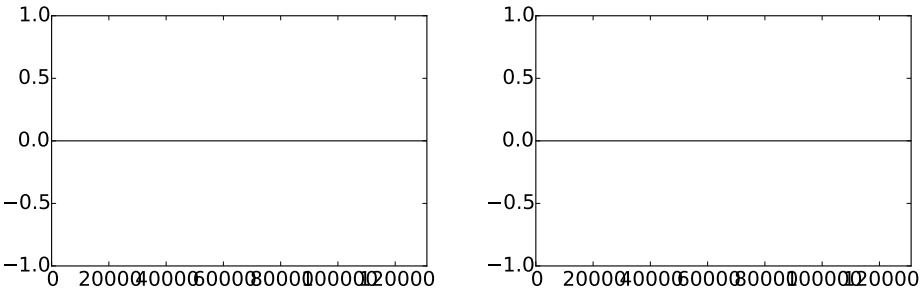

Figure 5.16: The error (i.e. the contribution from $W_0 \oplus W_1 \oplus \cdots \oplus W_{m-1}$) in the sound file `castanets.wav`, for $m = 1$ and $m = 2$, respectively.

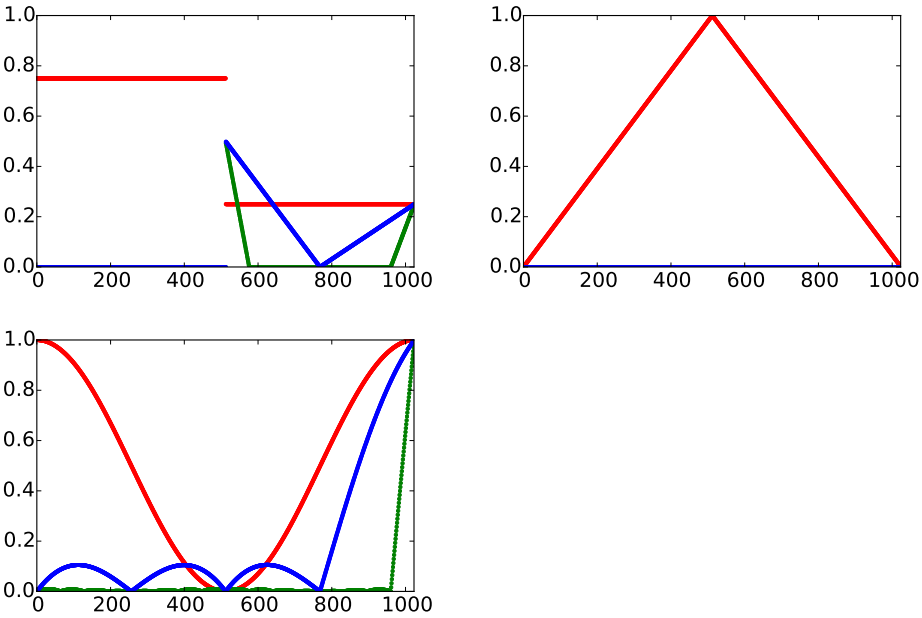

Figure 5.17: The error (i.e. the contribution from $W_0 \oplus W_1 \oplus \cdots \oplus W_{m-1}$) for $N = 1025$ when f is a square wave, the linear function $f(t) = 1 - 2|1/2 - t/N|$, and the trigonometric function $f(t) = 1/2 + \cos(2\pi t/N)/2$, respectivey. The detail is indicated for $m = 6$ and $m = 8$.

functions, due to discontinuity. For the second function we see that there is no error. The reason is that this function is piecewise linear, so there is no error when we represent the function from the space V_0. With the third function, hoewever, we see an error.

What you should have learned in this section.

- Definition of scaling function, mother wavelet, resolution spaces, and detail spaces for the wavelet of piecewise linear functions.

Exercise 5.26: The sample values are coordinates

Show that, for $f \in V_0$ we have that $[f]_{\boldsymbol{\phi}_0} = (f(0), f(1), \ldots, f(N-1))$. This generalizes the result for piecewise constant functions.

Exercise 5.27: Computing projections

In this exercise we will show how the projection of $\phi_{1,1}$ onto V_0 can be computed. We will see from this that it is nonzero, and that its support is the entire $[0, N]$. Let $f = \text{proj}_{V_0}\phi_{1,1}$, and let $x_n = f(n)$ for $0 \leq n < N$. This means that, on $(n, n+1)$, $f(t) = x_n + (x_{n+1} - x_n)(t-n)$.

a) Show that $\int_n^{n+1} f(t)^2 dt = (x_n^2 + x_n x_{n+1} + x_{n+1}^2)/3$.

b) Show that

$$\int_0^{1/2} (x_0 + (x_1 - x_0)t)\phi_{1,1}(t)dt = 2\sqrt{2}\left(\frac{1}{12}x_0 + \frac{1}{24}x_1\right)$$
$$\int_{1/2}^{1} (x_0 + (x_1 - x_0)t)\phi_{1,1}(t)dt = 2\sqrt{2}\left(\frac{1}{24}x_0 + \frac{1}{12}x_1\right).$$

c) Use the fact that

$$\int_0^N (\phi_{1,1}(t) - \sum_{n=0}^{N-1} x_n \phi_{0,n}(t))^2 dt$$
$$= \int_0^1 \phi_{1,1}(t)^2 dt - 2\int_0^{1/2} (x_0 + (x_1 - x_0)t)\phi_{1,1}(t)dt - 2\int_{1/2}^1 (x_0 + (x_1 - x_0)t)\phi_{1,1}(t)dt$$
$$+ \sum_{n=0}^{N-1} \int_n^{n+1} (x_n + (x_{n-1} - x_n)t)^2 dt$$

and a) and b) to find an expression for $\|\phi_{1,1}(t) - \sum_{n=0}^{N-1} x_n \phi_{0,n}(t)\|^2$.

d) To find the minimum least squares error, we can set the gradient of the expression in c. to zero, and thus find the expression for the projection of $\phi_{1,1}$ onto V_0. Show that the values $\{x_n\}_{n=0}^{N-1}$ can be found by solving the equation $S\boldsymbol{x} = \boldsymbol{b}$, where $S = \frac{1}{3}\{1, \underline{4}, 1\}$ is an $N \times N$ symmetric filter, and $\boldsymbol{b}$ is the vector with components $b_0 = b_1 = \sqrt{2}/2$, and $b_k = 0$ for $k \geq 2$.

e) Solve the system in d. for some values of N to verify that the projection of $\phi_{1,1}$ onto V_0 is nonzero, and that its support covers the entire $[0, N]$.

Exercise 5.28: Non-orthogonality for the piecewise linear wavelet

Show that

$$\langle \phi_{0,n}, \phi_{0,n} \rangle = \frac{2}{3} \qquad \langle \phi_{0,n}, \phi_{0,n\pm 1} \rangle = \frac{1}{6} \qquad \langle \phi_{0,n}, \phi_{0,n\pm k} \rangle = 0 \text{ for } k > 1.$$

As a consequence, the $\{\phi_{0,n}\}_n$ are neither orthogonal, nor have norm 1.

Exercise 5.29: Wavelets based on polynomials

The convolution of two functions defined on $(-\infty, \infty)$ is defined by

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t)g(x-t)dt.$$

Show that we can obtain the piecewise linear ϕ we have defined as $\phi = \chi_{[-1/2,1/2)} * \chi_{[-1/2,1/2)}$ (recall that $\chi_{[-1/2,1/2)}$ is the function which is 1 on $[-1/2, 1/2)$ and 0 elsewhere). This gives us a nice connection between the piecewise constant scaling function (which is similar to $\chi_{[-1/2,1/2)}$) and the piecewise linear scaling function in terms of convolution.

5.5 Alternative wavelet based on piecewise linear functions

For the scaling function used for piecewise linear functions, $\{\phi(t-n)\}_{0 \le n < N}$ were not orthogonal anymore, contrary to the case for piecewise constant functions. We were still able to construct what we could call resolution spaces and detail spaces. We also mentioned that having many vanishing moments is desirable for a mother wavelet, and that the scaling function used for piecewise constant functions had one vanishing moment. It is easily checked, however, that the mother wavelet we now introduced for piecewise linear functions (i.e. $\psi(t) = \frac{1}{\sqrt{2}}\phi_{1,1}(t)$) has no vanishing moments. Therefore, this is not a very good choice of mother wavelet. We will attempt the following adjustment strategy to construct an alternative mother wavelet $\hat{\psi}$ which has two vanishing moments, i.e. one more than the Haar wavelet.

Idea 5.33. *Adjusting the wavelet construction.*

Adjust the wavelet construction in Theorem 5.30 to

$$\hat{\psi} = \psi - \alpha\phi_{0,0} - \beta\phi_{0,1} \tag{5.36}$$

and choose α, β so that

$$\int_0^N \hat{\psi}(t)\,dt = \int_0^N t\hat{\psi}(t)\,dt = 0, \tag{5.37}$$

and define $\boldsymbol{\psi}_m = \{\hat{\psi}_{m,n}\}_{n=0}^{N2^m-1}$, and W_m as the space spanned by $\boldsymbol{\psi}_m$.

We thus have two free variables α, β in Equation (5.36), to enforce the two conditions in Equation (5.37). In Exercise 5.30 you are taken through the details of solving this as two linear equations in the two unknowns α and β, and this gives the following result:

Lemma 5.34. *The new function ψ.*
The function

$$\hat{\psi}(t) = \psi(t) - \frac{1}{4}\big(\phi_{0,0}(t) + \phi_{0,1}(t)\big) \tag{5.38}$$

satisfies the conditions (5.37).

Using Equation (5.28), which stated that

$$\phi_{0,n} = \frac{1}{\sqrt{2}}\left(\frac{1}{2}\phi_{1,2n-1} + \phi_{1,2n} + \frac{1}{2}\phi_{1,2n+1}\right), \tag{5.39}$$

we get

$$\begin{aligned}
\hat{\psi}_{0,n} &= \psi_{0,n} - \frac{1}{4}\big(\phi_{0,n} + \phi_{0,n+1}\big) \\
&= \frac{1}{\sqrt{2}}\phi_{1,2n+1} - \frac{1}{4}\frac{1}{\sqrt{2}}\left(\frac{1}{2}\phi_{1,2n-1} + \phi_{1,2n} + \frac{1}{2}\phi_{1,2n+1}\right) \\
&- \frac{1}{4}\frac{1}{\sqrt{2}}\left(\frac{1}{2}\phi_{1,2n+1} + \phi_{1,2n+2} + \frac{1}{2}\phi_{1,2n+3}\right) \\
&= \frac{1}{\sqrt{2}}\left(-\frac{1}{8}\phi_{1,2n-1} - \frac{1}{4}\phi_{1,2n} + \frac{3}{4}\phi_{1,2n+1} - \frac{1}{4}\phi_{1,2n+2} - \frac{1}{8}\phi_{1,2n+3}\right)
\end{aligned} \tag{5.40}$$

Note that what we did here is equivalent to finding the coordinates of $\hat{\psi}$ in the basis $\boldsymbol{\phi}_1$: Equation (5.38) says that

$$[\hat{\psi}]_{(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)} = (-1/4, -1/4, 0, \ldots, 0) \oplus (1, 0, \ldots, 0). \tag{5.41}$$

Since the IDWT is the change of coordinates from $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$ to $\boldsymbol{\phi}_1$, we could also have computed $[\hat{\psi}]_{\boldsymbol{\phi}_1}$ by taking the IDWT of $(-1/4, -1/4, 0, \ldots, 0) \oplus (1, 0, \ldots, 0)$. In the next section we will consider more general implementations of the DWT and the IDWT, which we thus can use instead of performing the computation above.

In summary we have

$$\begin{aligned}
\phi_{0,n} &= \frac{1}{\sqrt{2}}(\frac{1}{2}\phi_{1,2n-1} + \phi_{1,2n} + \frac{1}{2}\phi_{1,2n+1}) \\
\hat{\psi}_{0,n} &= \frac{1}{\sqrt{2}}\left(-\frac{1}{8}\phi_{1,2n-1} - \frac{1}{4}\phi_{1,2n} + \frac{3}{4}\phi_{1,2n+1} - \frac{1}{4}\phi_{1,2n+2} - \frac{1}{8}\phi_{1,2n+3}\right),
\end{aligned} \tag{5.42}$$

which gives us the change of coordinate matrix $P_{\boldsymbol{\phi}_1 \leftarrow (\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)}$. The new function $\hat{\psi}$ is plotted in Figure 5.18.

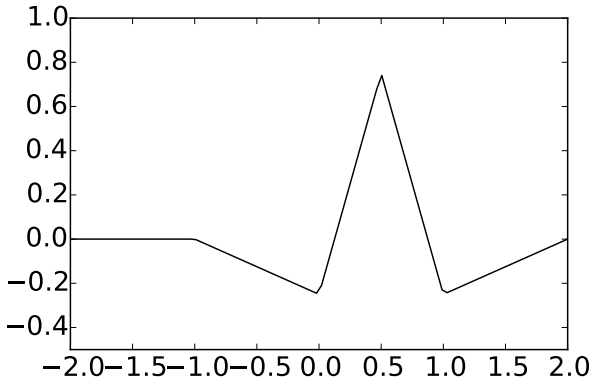

Figure 5.18: The function $\hat{\psi}$ we constructed as an alternative wavelet for piecewise linear functions.

We see that $\hat{\psi}$ has support $(-1, 2)$, and consist of four linear segments glued together. This is in contrast with the old ψ, which was simpler in that it had the shorther support $(0, 1)$, and consisted of only two linear segments glued together. It may therefore seem surprising that $\hat{\psi}$ is better suited for approximating functions than ψ. This is indeed a more complex fact, which may not be deduced by simply looking at plots of the functions.

Example 5.35. *DWT on sound.*

Also in this case we will see later how to write kernel transformations for the alternative piecewise wavelet. We will call these `DWTKernelpwl2` and `IDWTKernelpwl2` (2 stands for 2 vanishing moments). Using these we can plot the detail/error in the test audio file `castanets.wav` for different resolutions for our alternative wavelet, as we did in Example 5.19. The result is shown in Figure 5.19. Again, when comparing with Figure 5.11 we see much of the same. It is difficult to see an improvement from this figure. However, this figure also clearly shows a smaller error than the wavelet of the preceding section. A partial explanation is that the wavelet we now have constructed has two vanishing moments, while the previous one had not.

Example 5.36. *DWT on the samples of a mathematical function.*

Let us also repeat Exercise 5.20 for our alternative wavelet, where we plotted the detail/error at different resolutions, for the samples of a mathematical function. Figure 5.20 shows the new plot.

Again for the square wave there is an error, which seems to be slightly lower than for the previous wavelet. For the second function we see that there is no error, as before. The reason is the same as before, since the function is piecewise

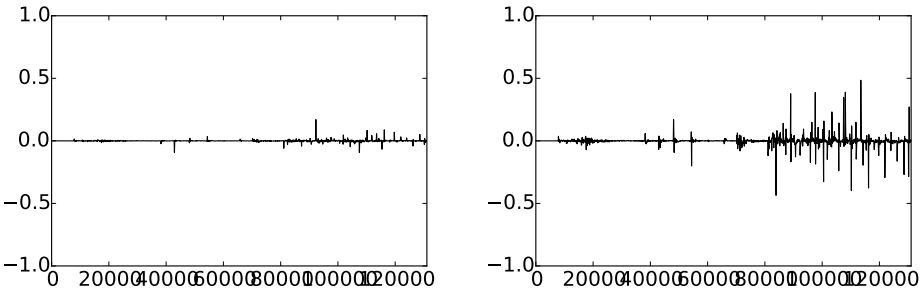

Figure 5.19: The error (i.e. the contribution from $W_0 \oplus W_1 \oplus \cdots \oplus W_{m-1}$) in the sound file `castanets.wav`, for $m = 1$ and $m = 2$, respectively.

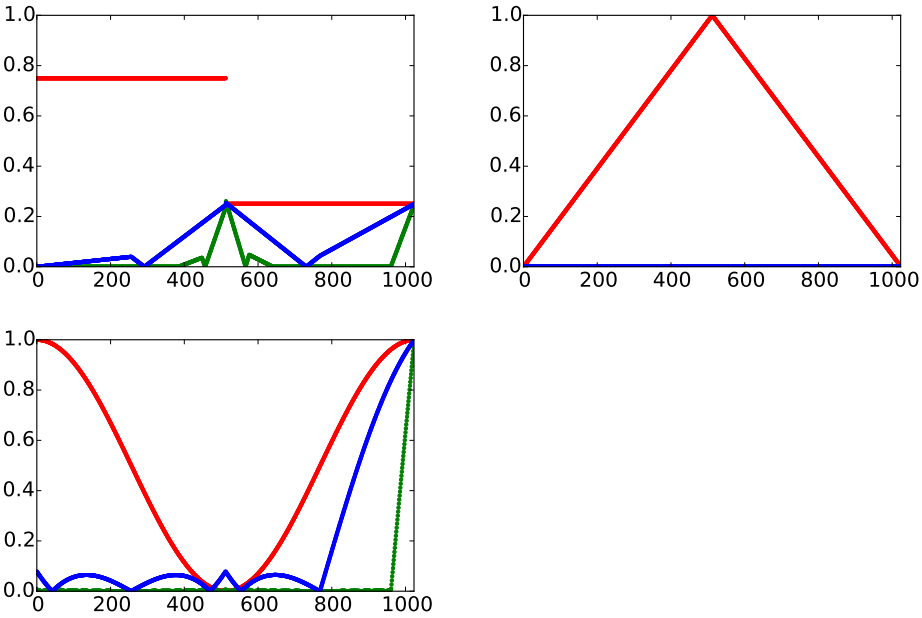

Figure 5.20: The error (i.e. the contribution from $W_0 \oplus W_1 \oplus \cdots \oplus W_{m-1}$) for $N = 1025$ when f is a square wave, the linear function $f(t) = 1 - 2|1/2 - t/N|$, and the trigonometric function $f(t) = 1/2 + \cos(2\pi t/N)/2$, respectivey. The detail is indicated for $m = 6$ and $m = 8$.

linear. With the third function there is an error. The error seems to be slightly lower than for the previous wavelet, which fits well with tha fact that this new wavelet has a bigger number of vanishing moments.

Example 5.37. *Playing sound.*

In Exercise 5.19 we implemented a function `playDWT` which could play the low resolution part in a sound, and we tested this for the Haar wavelet. Let us now also test this for the two piecewise linear wavelets we have constructed,

and the new wavelet kernels we have implemented. Code which plays the low resolution part for all three wavelet kernels can look as follows:

```
playDWT(m, DWTKernelHaar, IDWTKernelHaar, True)
playDWT(m, DWTKernelpwl0, IDWTKernelpwl0, True)
playDWT(m, DWTKernelpwl2, IDWTKernelpwl2, True)
```

The first call to `playDWT` plays the low-resolution part using the Haar wavelet. The code then moves on to the two piecewise linear wavelets. We clearly hear different sounds when we run this code for different m, so that the three wavelets act differently on the sound (if you want, you can here write a `for`-loop around the code, running through different m). Perhaps the alternative piecewise wavelet gives a bit better quality.

What you should have learned in this section.

- How one alters the mother wavelet for piecewise linear functions, in order to add a vanishing moment.

Exercise 5.30: Two vanishing moments

In this exercise we will show that there is a unique function on the form fiven by Equation (5.36) in the compendium which has two vanishing moments.

a) Show that, when $\hat{\psi}$ is defined by Equation (5.36) in the compendium, we have that

$$\hat{\psi}(t) = \begin{cases} -\alpha t - \alpha & \text{for } -1 \leq t < 0 \\ (2+\alpha-\beta)t - \alpha & \text{for } 0 \leq t < 1/2 \\ (\alpha-\beta-2)t - \alpha + 2 & \text{for } 1/2 \leq t < 1 \\ \beta t - 2\beta & \text{for } 1 \leq t < 2 \\ 0 & \text{for all other } t \end{cases}$$

b) Show that

$$\int_0^N \hat{\psi}(t)dt = \frac{1}{2} - \alpha - \beta, \qquad \int_0^N t\hat{\psi}(t)dt = \frac{1}{4} - \beta.$$

c) Explain why there is a unique function on the form given by Equation (5.36) in the compendium which has two vanishing moments, and that this function is given by Equation (5.38) in the compendium.

Exercise 5.31: Implement finding ψ with vanishing moments

In the previous exercise we ended up with a lot of calculations to find α, β in Equation (5.36) in the compendium. Let us try to make a program which does this for us, and which also makes us able to generalize the result.

a) Define

$$a_k = \int_{-1}^{1} t^k(1-|t|)dt, \quad b_k = \int_0^2 t^k(1-|t-1|)dt, \quad e_k = \int_0^1 t^k(1-2|t-1/2|)dt,$$

for $k \geq 0$. Explain why finding α, β so that we have two vanishing moments in Equation (5.36) in the compendium is equivalent to solving the following equation:

$$\begin{pmatrix} a_0 & b_0 \\ a_1 & b_1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \end{pmatrix}$$

Write a program which sets up and solves this system of equations, and use this program to verify the values for α, β we previously have found.

Hint. you can integrate functions in Python with the function `quad` in the package `scipy.integrate`. As an example, the function $\phi(t)$, which is nonzero only on $[-1, 1]$, can be integrated as follows:

```
res, err = quad(lambda t: t**k*(1-abs(t)), -1, 1)
```

b) The procedure where we set up a matrix equation in a) allows for generalization to more vanishing moments. Define

$$\hat{\psi} = \psi_{0,0} - \alpha\phi_{0,0} - \beta\phi_{0,1} - \gamma\phi_{0,-1} - \delta\phi_{0,2}. \tag{5.43}$$

We would like to choose $\alpha, \beta, \gamma, \delta$ so that we have 4 vanishing moments. Define also

$$g_k = \int_{-2}^{0} t^k(1-|t+1|)dt, \qquad d_k = \int_1^3 t^k(1-|t-2|)dt$$

for $k \geq 0$. Show that $\alpha, \beta, \gamma, \delta$ must solve the equation

$$\begin{pmatrix} a_0 & b_0 & g_0 & d_0 \\ a_1 & b_1 & g_1 & d_1 \\ a_2 & b_2 & g_2 & d_2 \\ a_3 & b_3 & g_3 & d_3 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix},$$

and solve this with your computer.

c) Plot the function defined by (5.43) in the compendium, which you found in b.

Hint. If `t` is the vector of t-values, and you write

```
(t >= 0)*(t <= 1)*(1-2*abs(t-0.5))
```

you get the points $\phi_{1,1}(t)$.

d) Explain why the coordinate vector of $\hat{\psi}$ in the basis $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$ is

$$[\hat{\psi}]_{(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)} = (-\alpha, -\beta, -\delta, 0, \ldots, 0 - \gamma) \oplus (1, 0, \ldots, 0).$$

Hint. You can also compare with Equation (5.41) in the compendium here. The placement of $-\gamma$ may seem a bit strange here, and has to with that $\phi_{0,-1}$ is not one of the basis functions $\{\phi_{0,n}\}_{n=0}^{N-1}$. However, we have that $\phi_{0,-1} = \phi_{0,N-1}$, i.e. $\phi(t+1) = \phi(t-N+1)$, since we always assume that the functions we work with have period N.

e) Sketch a more general procedure than the one you found in b., which can be used to find wavelet bases where we have even more vanishing moments.

Exercise 5.32: ψ for the Haar wavelet with two vanishing moments

Let $\phi(t)$ be the function we used when we defined the Haar-wavelet.

a) Compute $\text{proj}_{V_0}(f(t))$, where $f(t) = t^2$, and where f is defined on $[0, N)$.

b) Find constants α, β so that $\hat{\psi}(t) = \psi(t) - \alpha\phi_{0,0}(t) - \beta\phi_{0,1}(t)$ has two vanishing moments, i.e. so that $\langle\hat{\psi}, 1\rangle = 0$, and $\langle\hat{\psi}, t\rangle = 0$. Plot also the function $\hat{\psi}$.

Hint. Start with computing the integrals $\int \psi(t)dt$, $\int t\psi(t)dt$, $\int \phi_{0,0}(t)dt$, $\int \phi_{0,1}(t)dt$, and $\int t\phi_{0,0}(t)dt$, $\int t\phi_{0,1}(t)dt$.

c) Express ϕ and $\hat{\psi}$ with the help of functions from $\boldsymbol{\phi}_1$, and use this to write down the change of coordinate matrix from $(\boldsymbol{\phi}_0, \hat{\boldsymbol{\psi}}_0)$ to $\boldsymbol{\phi}_1$.

Exercise 5.33: More vanishing moments for the Haar wavelet

It is also possible to add more vanishing moments to the Haar wavelet. Define

$$\hat{\psi} = \psi_{0,0} - a_0\phi_{0,0} - \cdots - a_{k-1}\phi_{0,k-1}.$$

Define also $c_{r,l} = \int_l^{l+1} t^r dt$, and $e_r = \int_0^1 t^r \psi(t)dt$.

a) Show that $\hat{\psi}$ has k vanishing moments if and only if $a_0, \ldots, a_{k-1}$ solves the equation

$$\begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,k-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,k-1} \\ \vdots & \vdots & \vdots & \vdots \\ c_{k-1,0} & c_{k-1,1} & \cdots & c_{k-1,k-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{k-1} \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{k-1} \end{pmatrix} \tag{5.44}$$

b) Write a function `vanishingmomshaar` which takes k as input, solves Equation (5.44) in the compendium, and returns the vector $\boldsymbol{a} = (a_0, a_1, \ldots, a_{k-1})$.

Exercise 5.34: Listening experiments

Run the function `playDWT` for different m for the Haar wavelet, the piecewise linear wavelet, and the alternative piecewise linear wavelet, but listen to the detail components $W_0 \oplus W_1 \oplus \cdots \oplus W_{m-1}$ instead. Describe the sounds you hear for different m, and try to explain why the sound seems to get louder when you increase m.

5.6 Multiresolution analysis: A generalization

Let us summarize the properties of the spaces V_m. In both our examples we showed that they were nested, i.e.

$$V_0 \subset V_1 \subset V_2 \subset \cdots \subset V_m \subset \cdots .$$

We also showed that continuous functions could be approximated arbitrarily well from V_m, as long as m was chosen large enough. Moreover, the space V_0 is closed under all translates, at least if we view the functions in V_0 as periodic with period N. In the following we will always identify a function with this periodic extension, just as we did in Fourier analysis. When performing this identification, we also saw that $f(t) \in V_m$ if and only if $g(t) = f(2t) \in V_{m+1}$. We have therefore shown that the scaling functions we have considered fit into the following general framework.

Definition 5.38. *Multiresolution analysis.*

A Multiresolution analysis, or MRA, is a nested sequence of function spaces

$$V_0 \subset V_1 \subset V_2 \subset \cdots \subset V_m \subset \cdots , \tag{5.45}$$

called resolution spaces, so that

Any function can be approximated arbitrarily well from V_m, as long as m is large enough,

$f(t) \in V_0$ if and only if $f(2^m t) \in V_m$,

$f(t) \in V_0$ if and only if $f(t - n) \in V_0$ for all n.

There is a function ϕ, called a scaling function, so that $\boldsymbol{\phi} = \{\phi(t-n)\}_{0\leq n<N}$ is a basis for V_0.

When $\boldsymbol{\phi}$ is an orthonormal basis we say that the MRA is *orthonormal*.

The wavelet of piecewise constant functions was an orthonormal MRA, while the wavelets for piecewise linear functions were not. Although the definition above states that any function can be approximated with MRA's, in practice one needs to restrict to certain functions: Certain pathological functions may be difficult to approximate. In the literature one typically requires that the function is in $L^2(\mathbb{R})$, and also that the scaling function and the spaces V_m are in $L^2(\mathbb{R})$. MRA's are much used, and one can find a wide variety of functions ϕ, not only piecewise constant functions, which give rise to MRA's.

In the examples we have considered we also chose a mother wavelet. The term wavelet is used in very general terms. However, the term mother wavelet is quite concrete, and is what gives rise to the theory of wavelets. This was necessary in order to efficiently decompose the $g_m \in V_m$ into a low resolution approximation $g_{m-1} \in V_{m-1}$, and a detail/error e_{m-1} in a detail space we called W_{m-1}. We have freedom in how we define these detail spaces, as well as how we define a mother wavelet whose translates span the detail space (in general we choose a mother wavelet which simplifies the computation of the decomposition $g_m = g_{m-1} + e_{m-1}$, but we will see later that it also is desirable to choose a ψ with other properties). Once we agree on the detail spaces and the mother wavelet, we can perform a change of coordinates to find detail and low resolution approximations. We thus have the following general recipe.

Idea 5.39. *Recipe for constructing wavelets.*

In order to construct MRA's which are useful for practical purposes, we need to do the following:

- Find a function ϕ which can serve as the scaling function for an MRA,
- Find a function ψ so that $\boldsymbol{\psi} = \{\psi(t-n)\}_{0\leq n<N}$ and $\boldsymbol{\phi} = \{\phi(t-n)\}_{0\leq n<N}$ together form an orthonormal basis for V_1. The function ψ is also called a mother wavelet.

With V_0 the space spanned by $\boldsymbol{\phi} = \{\phi(t-n)\}_{0\leq n<N}$, and W_0 the space spanned by $\boldsymbol{\psi} = \{\psi(t-n)\}_{0\leq n<N}$, ϕ and ψ should be chosen so that we easily can compute the decomposition of $g_1 \in V_1$ into $g_0 + e_0$, where $g_0 \in V_0$ and $e_0 \in W_0$. If we can achieve this, the Discrete Wavelet Transform is defined as the change of coordinates from $\boldsymbol{\phi}_1$ to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$.

More generally, if

$$f(t) = \sum_n c_{m,n}\phi_{m,n} = \sum_n c_{0,n}\phi_{0,n} + \sum_{m'<m,n} w_{m',n}\psi_{m',n},$$

then the m-level DWT is defined by $\mathrm{DWT}(\boldsymbol{c}_m) = (\boldsymbol{c}_0, \boldsymbol{w}_0, \ldots, \boldsymbol{w}_{m-1})$. It is useful to interpret m as frequency, n as time, and $w_{m,n}$ as the contribution

at frequency m and time n. In this sense, wavelets provide a *time-frequency representation* of signals. This is what can make them more useful than Fourier analysis, which only provides frequency representations.

While there are in general many possible choices of detail spaces, in the case of an orthonormal wavelet we saw that it was natural to choose the detail space W_{m-1} as the orthogonal complement of V_{m-1} in V_m, and obtain the mother wavelet by projecting the scaling function onto the detail space. Thus, for orthonormal MRA's, the low-resolution approximation and the detail can be obtained by computing projections, and the least squares approximation of f from V_m can be computed as

$$\text{proj}_{V_m}(f) = \sum_n \langle f, \phi_{m,n}\rangle \phi_{m,n}(t).$$

5.6.1 Working with the samples of f instead of f

In the MRA-setting it helps to think about the continuous-time function $f(t)$ as the model for an image, which is the object under study. f itself may not be in any V_m, however (this corresponds to that detail is present in the image for infinitely many m), and increasing m corresponds to that we also include the detail we see when we zoom in on the image. But how can we obtain useful approximations to f from V_m? In case of an orthonormal MRA we can compute the least squares approximation as above, but we then need to compute the integrals $\langle f, \phi_{m,n}\rangle$, so that all function values are needed. However, as before we have only access to some samples $f(2^{-m}n)$, $0 \le n < 2^m N$. These are called pixel values in the context of images, so that we can only hope to obtain a good approximation to $f^{(m)}$ (and thus f) from the pixel values. The following result explains how we can obtain this.

Theorem 5.40. *Using the samples.*

If f is continuous, and ϕ has compact support, we have that, for all t,

$$f(t) = \lim_{m\to\infty} \sum_{n=0}^{2^m N-1} \frac{2^{-m}}{\int_0^N \phi_{m,0}(t)dt} f(n/2^m)\phi_{m,n}(t).$$

Proof. We have that

$$2^{-m} \sum_{n=0}^{2^m N-1} \phi_{m,n} = \sum_{n=0}^{2^m N-1} 2^{-m}\phi_{m,0}(t - 2^{-m}n).$$

We recognize this as a Riemann sum for the integral $\int_0^N \phi_{m,0}(t)dt$. Therefore,

$$\lim_{m\to\infty} \sum_{n=0}^{2^m N-1} 2^{-m}\phi_{m,n} = \int_0^N \phi_{m,0}(t)dt.$$

Also, finitely many n contribute in this sum since ϕ has compact support. We now get that

$$\begin{aligned}\sum_{n=0}^{2^m N-1} 2^{-m} f(n/2^m)\phi_{m,n}(t) &= \sum_{n \text{ so that } 2^{-m}n \approx t} 2^{-m} f(n/2^m)\phi_{m,n}(t) \\ &\approx \sum_{n \text{ so that } 2^{-m}n \approx t} 2^{-m} f(t)\phi_{m,n}(t) \\ &= f(t) \sum_{n \text{ so that } 2^{-m}n \approx t} 2^{-m}\phi_{m,n}(t) \approx f(t)\int_0^N \phi_{m,0}(t)dt.\end{aligned}$$

where we have used the continuity of f and that

$$\lim_{m\to\infty} \sum_{n=0}^{2^m N-1} 2^{-m}\phi_{m,n} = \int_0^N \phi_{m,0}(t)dt.$$

The result follows. Note that here we have not used the requirement that $\{\phi(t-n)\}_n$ are orthogonal. □

The coordinate vector $\boldsymbol{x} = \left(\frac{2^{-m}}{\int_0^N \phi_{m,0}(t)dt} f(n/2^m)\right)_{n=0}^{2^m N-1}$ in $\boldsymbol{\phi}_m$ is therefore a candidate to an approximation of both f and $f^{(m)}$ from V_m, using only the pixel values. Normally one drops the leading constant $\frac{2^{-m}}{\int_0^N \phi_{m,0}(t)dt}$, so that one simply considers the sample values $f(n/2^m)$ as a coordinate vector in $\boldsymbol{\phi}_m$. This is used as the input to the DWT.

5.6.2 Increasing the precision of the DWT

Even though the samples of f give a good approximation to f as above, the approximation and f are still different, so that we obtain different output from the DWT. In Section 7.1 we will argue that the output from the DWT is equivalent to sampling the output from certain analog filters. We would like the difference in the output from these analog filters to be as small as possible. If the functions ϕ, ψ are symmetric around 0, we will also see that the analog filters are symmetric (a filter is symmetric if and only if the convolution kernel is symmetric around 0), in which case we know that such a high precision implementation is possible using the simple technique of symmetric extension. Let us summarize this as the following idea.

Idea 5.41. *Symmetric wavelets.*
If the functions ϕ, ψ in a wavelet are symmetric around 0, then we can obtain an implementation of the DWT with higher precision when we consider symmetric extensions of the input.

Unfortunately, the piecewise constant scaling function we encountered was not symmetric. However, the piecewise linear scaling function was, and so are also many other interesting scaling functions we will encounter later. For a

symmetric function, denote as before the symmetric extension of the input f with $\breve{f}$. If the input $\boldsymbol{x}$ to the DWT are the samples $(f(n/2^m))_{n=0}^{2^mN-1}$, we create a vector $\breve{\boldsymbol{x}}$ representing the samples of $\breve{f}$. It is clear that this vector should be

$$\breve{\boldsymbol{x}} = \left((f(n/2^m))_{n=0}^{2^mN-1}, \lim_{t \to N_-} f(t), (f((2^mN-n)/2^m))_{n=1}^{2^mN-1} \right).$$

In this vector there is symmetry around entry 2^mN, so that the vector is determined from the 2^mN+1 first elements. Also the boundary is not duplicated, contrary to the previous symmetric extension given by Definition 4.1. We are thus lead to define a symmetric extension in the following way instead:

Definition 5.42. *Symmetric extension of a vector.*

By the *symmetric extension* of $\boldsymbol{x} \in \mathbb{R}^N$, we mean $\breve{\boldsymbol{x}} \in \mathbb{R}^{2N-2}$ defined by

$$\breve{\boldsymbol{x}}_k = \begin{cases} x_k & 0 \leq k < N \\ x_{2N-2-k} & N \leq k < 2N-3 \end{cases} \tag{5.46}$$

With this notation, $N-1$ is the symmetry point in all symmetric extensions. This is illustrated in Figure 5.21

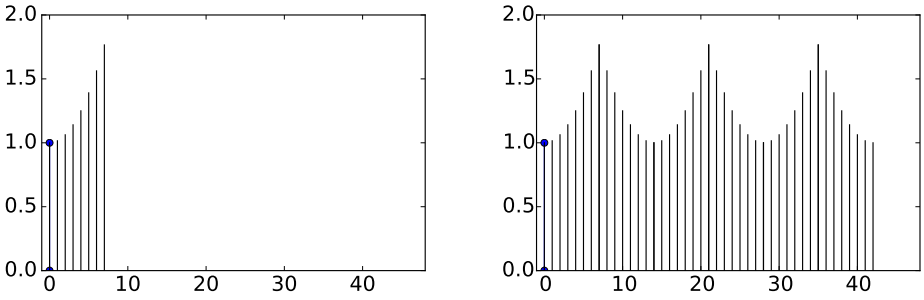

Figure 5.21: A vector and its symmetric extension. Note that the period of the vector is now $2N-2$, while it was $2N$ for the vector shown in Figure 4.1.

From Chapter 4 it follows that symmetric filters preserve the symmetry around $N-1$ when applied to such vectors. We can now define the symmetric restriction S_r as before, with the definition of symmetric extension replaced with the above. We now have the following analog to Theorem 4.9. The proof of this is left as an exercise.

Theorem 5.43. *Expression for S_r.*

With $S = \begin{pmatrix} S_1 \& S_2 \\ S_3 \& S_4 \end{pmatrix} \in \mathbb{R}^{2N-2} \times \mathbb{R}^{2N-2}$ a symmetric filter, with $S_1 \in \mathbb{R}^N \times \mathbb{R}^N$, $S_2 \in \mathbb{R}^N \times \mathbb{R}^{N-2}$, we have that $S_r = S_1 + \begin{pmatrix} 0 & (S_2)^f \& 0 \end{pmatrix}$.

With the Haar wavelet we succeeded in finding a function ψ which could be used in the recipe above. Note, however, that there may be many other ways to

define a function ψ which can be used in the recipe. In the next chapter we will follow the recipe in order to contruct other wavelets, and we will try to express which pairs of function ϕ, ψ are most interesting, and which resolution spaces are most interesting.

What you should have learned in this section.

- Definition of a multiresolution analysis.

Exercise 5.35: Prove expression for S_r

Prove Theorem 5.43. Use the proof of Theorem 4.9 as a guide.

Exercise 5.36: Orthonormal basis for the symmetric extensions

In this exercise we will establish an orthonormal basis for the symmetric extensions, as defined by Definition 5.42. This parallels Theorem 4.6.

a) Explain why, if $\boldsymbol{x} \in \mathbb{R}^{2N-2}$ is a symmetric extension (according to Definition 4.1), then $(\widehat{\boldsymbol{x}})_n = z_n e^{-\pi i n}$, where $\boldsymbol{z}$ is a real vectors which satisfies $z_n = z_{2N-2-n}$

b) Show that

$$\left\{ \boldsymbol{e}_0, \left\{ \frac{1}{\sqrt{2}} (\boldsymbol{e}_i + \boldsymbol{e}_{2N-2-i}) \right\}_{n=1}^{N-2}, \boldsymbol{e}_{N-1} \right\} \tag{5.47}$$

is an orthonormal basis for the vectors on the form $\widehat{\boldsymbol{x}}$ with $\boldsymbol{x} \in \mathbb{R}^{2N-2}$ a symmetric extension.

c) Show that

$$\begin{gathered} \frac{1}{\sqrt{2N-2}} \cos\left(2\pi \frac{0}{2N-2} k \right) \\ \left\{ \frac{1}{\sqrt{N-1}} \cos\left(2\pi \frac{n}{2N-2} k \right) \right\}_{n=1}^{N-2} \\ \frac{1}{\sqrt{2N-2}} \cos\left(2\pi \frac{N-1}{2N-2} k \right) \end{gathered} \tag{5.48}$$

is an orthonormal basis for the symmetric extensions in $\mathbb{R}^{2N-2}$.

d) Assume that S is symmetric. Show that the vectors listed in (5.48) in the compendium are eigenvectors for S_r, when the vectors are viewed as vectors in $\mathbb{R}^N$, and that they are linearly independent. This shows that S_r is diagonalizable.

Exercise 5.37: Diagonalizing S_r

Let us explain how the matrix S_r can be diagonalized, similarly to how we previously diagonalized using the DCT. In Exercise 5.36 we showed that the vectors

$$\left\{\cos\left(2\pi\frac{n}{2N-2}k\right)\right\}_{n=0}^{N-1} \tag{5.49}$$

in $\mathbb{R}^N$ is a basis of eigenvectors for S_r when S is symmetric. S_r itself is not symmetric, however, so that this basis can not possibly be orthogonal (S is symmetric if and only if it is orthogonally digonalizable). However, when the vectors are viewed in $\mathbb{R}^{2N-2}$ we showed in Exercise 5.36c) an orthogonality statement which can be written as

$$\sum_{k=0}^{2N-3}\cos\left(2\pi\frac{n_1}{2N-2}k\right)\cos\left(2\pi\frac{n_2}{2N-2}k\right) = (N-1)\times\begin{cases}2 & \text{if } n_1=n_2\in\{0,N-1\}\\ 1 & \text{if } n_1=n_2\notin\{0,N-1\}\\ 0 & \text{if } n_1\neq n_2\end{cases}. \tag{5.50}$$

a) Show that

$$\begin{aligned}&(N-1)\times\begin{cases}1 & \text{if } n_1=n_2\in\{0,N-1\}\\ \frac{1}{2} & \text{if } n_1=n_2\notin\{0,N-1\}\\ 0 & \text{if } n_1\neq n_2\end{cases}\\ =&\frac{1}{\sqrt{2}}\cos\left(2\pi\frac{n_1}{2N-2}\cdot 0\right)\frac{1}{\sqrt{2}}\cos\left(2\pi\frac{n_2}{2N-2}\cdot 0\right)\\ &+\sum_{k=1}^{N-2}\cos\left(2\pi\frac{n_1}{2N-2}k\right)\cos\left(2\pi\frac{n_2}{2N-2}k\right)\\ &+\frac{1}{\sqrt{2}}\cos\left(2\pi\frac{n_1}{2N-2}(N-1)\right)\frac{1}{\sqrt{2}}\cos\left(2\pi\frac{n_2}{2N-2}(N-1)\right).\end{aligned}$$

Hint. Use that $\cos x=\cos(2\pi-x)$ to pair the summands k and $2N-2-k$.

Now, define the vector $\boldsymbol{d}_n^{(\mathrm{I})}$ as

$$d_{n,N}\left(\frac{1}{\sqrt{2}}\cos\left(2\pi\frac{n}{2N-2}\cdot 0\right),\left\{\cos\left(2\pi\frac{n}{2N-2}k\right)\right\}_{k=1}^{N-2},\frac{1}{\sqrt{2}}\cos\left(2\pi\frac{n}{2N-2}(N-1)\right)\right),$$

and define $d_{0,N}^{(\mathrm{I})}=d_{N-1,N}^{(\mathrm{I})}=1/\sqrt{N-1}$, and $d_{n,N}^{(\mathrm{I})}=\sqrt{2/(N-1)}$ when $n>1$. The orthogonal $N\times N$ matrix where the rows are $\boldsymbol{d}_n^{(\mathrm{I})}$ is called the DCT-I,

and we will denote it by $D_N^{(\mathrm{I})}$. DCT-I is also much used, just as the DCT-II of Chapter 4. The main difference from the previous cosine vectors is that $2N$ has been replaced by $2N-2$.

b) Explain that the vectors $\boldsymbol{d}_n^{(\mathrm{I})}$ are orthonormal, and that the matrix

$$\sqrt{\frac{2}{N-1}}\begin{pmatrix} 1/\sqrt{2} & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1/\sqrt{2} \end{pmatrix}\left(\cos\left(2\pi\tfrac{n}{2N-2}k\right)\right)\begin{pmatrix} 1/\sqrt{2} & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1/\sqrt{2} \end{pmatrix}$$

is orthogonal.

c) Explain from b. that $\left(\cos\left(2\pi\tfrac{n}{2N-2}k\right)\right)^{-1}$ can be written as

$$\frac{2}{N-1}\begin{pmatrix} 1/2 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1/2 \end{pmatrix}\left(\cos\left(2\pi\tfrac{n}{2N-2}k\right)\right)\begin{pmatrix} 1/2 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1/2 \end{pmatrix}$$

With the expression we found in c., S_r can now be diagonalized as

$$\left(\cos\left(2\pi\tfrac{n}{2N-2}k\right)\right)D\left(\cos\left(2\pi\tfrac{n}{2N-2}k\right)\right)^{-1}.$$

5.7 Summary

We started this chapter by motivating the theory of wavelets as a different function approximation scheme, which solved some of the shortcomings of Fourier series. While one approximates functions with trigonometric functions in Fourier theory, with wavelets one instead approximates a function in several stages, where one at each stage attempts to capture information at a given resolution, using a function prototype. This prototype is localized in time, contrary to the Fourier basis functions, and this makes the theory of wavelets suitable for time-frequency representations of signals. We used an example based on Google Earth to illustrate that the wavelet-based scheme can represent an image at different resolutions in a scalable way, so that passing from one resolution to another simply mounts to adding some detail information to the lower resolution version of the image. This also made wavelets useful for compression, since the images at different resolutions can serve as compressed versions of the image.

We defined the simplest wavelet, the Haar wavelet, which is a function approximation scheme based on piecewise constant functions, and deduced its properties. We defined the Discrete Wavelet Transform (DWT) as a change of coordinates corresponding to the function spaces we defined. This transform is the crucial object to study when it comes to more general wavelets also, since it is the object which makes wavelets useful for computation. In the following chapters, we will see that reordering of the source and target bases of the DWT will aid in expressing connections between wavelets and filters, and in constructing optimized implementations of the DWT.

We then defined another wavelet, which corresponded to a function approximation scheme based on piecewise linear functions, instead of piecewise constant functions. There were several differences with the new wavelet when compared to the previous one. First of all, the basis functions were not orthonormal, and we did not attempt to make them orthonormal. The resolution spaces we now defined were not defined in terms of orthogonal bases, and we had some freedom on how we defined the detail spaces, since they are not defined as orthogonal complements anymore. Similarly, we had some freedom on how we define the mother wavelet, and we mentioned that we could define it so that it is more suitable for approximation of functions, by adding what we called vanishing moments.

From these examples of wavelets and their properties we made a generalization to what we called a multiresolution analysis (MRA). In an MRA we construct successively refined spaces of functions that may be used to approximate functions arbitrarily well. We will continue in the next chapter to construct even more general wavelets, within the MRA framework.

The book [21] goes through developments for wavelets in detail. While wavelets have been recognized for quite some time, it was with the important work of Daubechies [8, 9] that they found new arenas in the 80's. Since then they found important applications. The main application we will focus on in later chapters is image processing.

Chapter 6

The filter representation of wavelets

Previously we saw that analog filters restricted to the Fourier spaces gave rise to digital filters. These digital filters sent the samples of the input function to the samples of the output function, and are easily implementable, in contrast to the analog filters. We have also seen that wavelets give rise to analog filters. This leads us to believe that the DWT also can be implemented in terms of digital filters. In this chapter we will prove that this is in fact the case.

There are some differences between the Fourier and wavelet settings, however:

- The DWT is not constructed by looking at the samples of a function, but rather by looking at coordinates in a given basis.
- The function spaces we work in (i.e. V_m) are different from the Fourier spaces.
- The DWT gave rise to two different types of analog filters: The filter defined by Equation (7.16) for obtaining $c_{m,n}$, and the filter defined by Equation (7.17) for obtaining $w_{m,n}$. We want both to correspond to digital filters.

Due to these differences, the way we realize the DWT in terms of filters will be a bit different. Despite the differences, this chapter will make it clear that the output of a DWT can be interpreted as the combined output of two different filters, and each filter will have an interpretation in terms of frequency representations. We will also see that the IDWT has a similar interpretation in terms of filters.

In this chapter we will also see that expressing the DWT in terms of filters will also enable us to define more general transforms, where even more filters are used. It is fruitful to think about each filter as concentrating on a particular frequency range, and that these transforms thus simply splits the input into different frequency bands. Such transforms have important applications to the

processing and compression of sound, and we will show that the much used MP3 standard for compression of sound takes use of such transforms.

6.1 The filters of a wavelet transformation

We will make the connection with digital filters by looking again at the different examples of wavelet bases we have seen: The ones for piecewise constant and piecewise linear functions. For the Haar wavelet we have noted that G and H are block-diagonal with $\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$ repeated along the diagonal. For the piecewise linear wavelet, Equation (5.33) gives that the first two columns in $G = P_{\boldsymbol{\phi}_m \leftarrow \mathcal{C}_m}$ take the form

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 1/2 & 1 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 1/2 & 0 \end{pmatrix}. \tag{6.1}$$

The remaining columns are obtained by shifting this, as in a circulant Toeplitz matrix. Similarly, Equation (5.35) gives that the first two columns in $H = P_{\mathcal{C}_m \leftarrow \boldsymbol{\phi}_m}$ take the form

$$\sqrt{2} \begin{pmatrix} 1 & 0 \\ -1/2 & 1 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ -1/2 & 0 \end{pmatrix}. \tag{6.2}$$

Also here, the remaining columns are obtained by shifting this, as in a circulant Toeplitz matrix. For the alternative piecewise linear wavelet, Equation (5.42) give all columns in the change of coordinate matrix $G = P_{\boldsymbol{\phi}_m \leftarrow \mathcal{C}_m}$ also. In particular, the first two columns in this matrix are

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1/4 \\ 1/2 & 3/4 \\ 0 & -1/4 \\ 0 & -1/8 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 1/2 & -1/8 \end{pmatrix}. \tag{6.3}$$

The first column is the same as before, since there was no change in the definition of ϕ. The remaining columns are obtained by shifting this, as in a circulant Toeplitz matrix. We will explain later how the change of coordinate matrix $H = P_{\mathcal{C}_m \leftarrow \boldsymbol{\phi}_m}$ also can be computed.

In each case above it turned out that the kernel transformations $G = P_{\boldsymbol{\phi}_m \leftarrow \mathcal{C}_m}$, $H = P_{\mathcal{C}_m \leftarrow \boldsymbol{\phi}_m}$ had a special structure: They were obtained by repeating the first two columns in a circulant way, similarly to how we did in a circulant Toeplitz matrix. The matrices were not exactly circulant Toeplitz matrices, however, since there are two different columns repeating. The change of coordinate matrices occuring in the stages in a DWT are thus not digital filters, but they seem to be related. Let us start by giving these new matrices names:

Definition 6.1. *MRA-matrices.*

An $N \times N$-matrix T, with N even, is called an MRA-matrix if the columns are translates of the first two columns in alternating order, in the same way as the columns of a circulant Toeplitz matrix.

From our previous calculations it is clear that, once ϕ and ψ are given through an MRA, the corresponding change of coordinate matrices will always be MRA-matrices. The MRA-matrices is our connection between filters and wavelets. Let us make the following definition:

Definition 6.2. *H_0 and H_1.*

We denote by H_0 the (unique) filter with the same first row as H, and by H_1 the (unique) filter with the same second row as H. H_0 and H_1 are also called the *DWT filter components*.

Using this definition it is clear that

$$(H\boldsymbol{c}_m)_k = \begin{cases} (H_0\boldsymbol{c}_m)_k & \text{when } k \text{ is even} \\ (H_1\boldsymbol{c}_m)_k & \text{when } k \text{ is odd,} \end{cases}$$

since the left hand side depends only on row k in the matrix H, and this is equal to row k in H_0 (when k is even) or row k in H_1 (when k is odd). This means that $H\boldsymbol{c}_m$ can be computed with the help of H_0 and H_1 as follows:

Theorem 6.3. *DWT expressed in terms of filters.*

Let $\boldsymbol{c}_m$ be the coordinates in $\boldsymbol{\phi}_m$, and let H_0, H_1 be defined as above. Any stage in a DWT can ble implemented in terms of filters as follows:

- Compute $H_0\boldsymbol{c}_m$. The even-indexed entries in the result are the cordinates $\boldsymbol{c}_{m-1}$ in $\boldsymbol{\phi}_{m-1}$.
- Compute $H_1\boldsymbol{c}_m$. The odd-indexed entries in the result are the coordinates $\boldsymbol{w}_{m-1}$ in $\boldsymbol{\psi}_{m-1}$.

This gives an important connection between wavelets and filters: The DWT corresponds to applying two filters, H_0 and H_1, and the result from the DWT is produced by assembling half of the coordinates from each. Keeping only every second coordinate is called *downsampling* (with a factor of two). Had we not performed downsampling, we would have ended up with twice as many coordinates as we started with. Downsampling with a factor of two means that we end up with the same number of samples as we started with. We also say that the output of the two filters is *critically sampled*. Due to the critical sampling, it is inefficient to compute the full application of the filters. We will return to the issue of making efficient implementations of critically sampled filter banks later.

We can now complement Figure 5.9 by giving names to the arrows as follows:

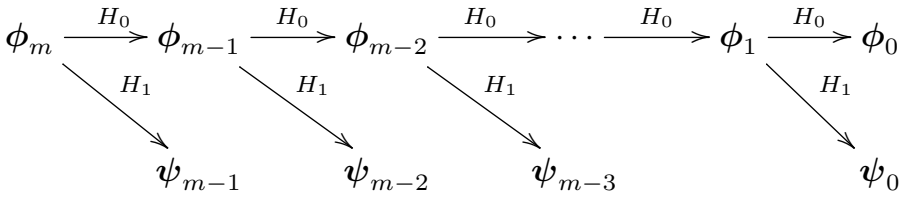

Figure 6.1: Detailed illustration of a wavelet transform.

Let us make a similar anlysis for the IDWT, and let us first make the following definition:

Definition 6.4. *G_0 and G_1.*

We denote by G_0 the (unique) filter with the same first column as G, and by G_1 the (unique) filter with the same second column as G. G_0 and G_1 are also called the *IDWT filter components*.

These filters are uniquely determined, since any filter is uniquely determined from one of its columns. We can now write

$$\begin{aligned}
\boldsymbol{c}_m = G\begin{pmatrix} c_{m-1,0} \\ w_{m-1,0} \\ c_{m-1,1} \\ w_{m-1,1} \\ \cdots \\ c_{m-1,2^{m-1}N-1} \\ w_{m-1,2^{m-1}N-1} \end{pmatrix} &= G\left(\begin{pmatrix} c_{m-1,0} \\ 0 \\ c_{m-1,1} \\ 0 \\ \cdots \\ c_{m-1,2^{m-1}N-1} \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ w_{m-1,0} \\ 0 \\ w_{m-1,1} \\ \cdots \\ 0 \\ w_{m-1,2^{m-1}N-1} \end{pmatrix}\right) \\
&= G\begin{pmatrix} c_{m-1,0} \\ 0 \\ c_{m-1,1} \\ 0 \\ \cdots \\ c_{m-1,2^{m-1}N-1} \\ 0 \end{pmatrix} + G\begin{pmatrix} 0 \\ w_{m-1,0} \\ 0 \\ w_{m-1,1} \\ \cdots \\ 0 \\ w_{m-1,2^{m-1}N-1} \end{pmatrix} \\
&= G_0\begin{pmatrix} c_{m-1,0} \\ 0 \\ c_{m-1,1} \\ 0 \\ \cdots \\ c_{m-1,2^{m-1}N-1} \\ 0 \end{pmatrix} + G_1\begin{pmatrix} 0 \\ w_{m-1,0} \\ 0 \\ w_{m-1,1} \\ \cdots \\ 0 \\ w_{m-1,2^{m-1}N-1} \end{pmatrix}.
\end{aligned}$$

Here we have split a vector into its even-indexed and odd-indexed elements, which correspond to the coefficients from $\boldsymbol{\phi}_{m-1}$ and $\boldsymbol{\psi}_{m-1}$, respectively. In the last equation, we replaced with G_0, G_1, since the multiplications with G depend only on the even and odd columns in that matrix (due to the zeros inserted), and these columns are equal in G_0, G_1. We can now state the following characterization of the inverse Discrete Wavelet transform:

Theorem 6.5. *IDWT expressed in terms of filters.*

Let G_0, G_1 be defined as above. Any stage in an IDWT can be implemented in terms of filters as follows:

$$\boldsymbol{c}_m = G_0\begin{pmatrix} c_{m-1,0} \\ 0 \\ c_{m-1,1} \\ 0 \\ \cdots \\ c_{m-1,2^{m-1}N-1} \\ 0 \end{pmatrix} + G_1\begin{pmatrix} 0 \\ w_{m-1,0} \\ 0 \\ w_{m-1,1} \\ \cdots \\ 0 \\ w_{m-1,2^{m-1}N-1} \end{pmatrix}. \tag{6.4}$$

Making a new vector where zeroes have been inserted in this way is also called *upsampling* (with a factor of two). We can now also complement Figure 5.9 for the IDWT with named arrows. This has bee done in Figure 6.2

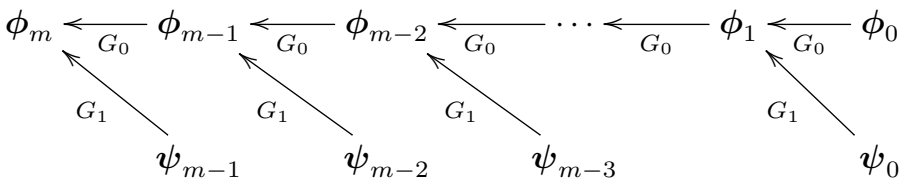

Figure 6.2: Detailed illustration of an IDWT.

Note that the filters G_0, G_1 were defined in terms of the columns of G, while the filters H_0, H_1 were defined in terms of the rows of H. This difference is seen from the computations above to come from that the change of coordinates one way splits the coordinates into two parts, while the inverse change of coordinates performs the opposite. Let us summarize what we have found as follows.

Fact 6.6. *Computing DWT/IDWT through filters.*
The DWT can be computed with the help of two filters H_0, H_1, as explained in Theorem 6.3. Any linear transformation computed from two filters H_0, H_1 in this way is called a *forward filter bank transform*. The IDWT can be computed with the help of two filters G_0, G_1 as explained in Theorem 6.5. Any linear transformation computed from two filters G_0, G_1 in this way is called a *reverse filter bank transform*.

In Chapter 8 we will go through how any forward and reverse filter bank transform can be implemented, once we have the filters H_0, H_1, G_0, and G_1. When we are in a wavelet setting, the filter coefficients in these four filters can be found from the relations between the bases $\boldsymbol{\phi}_1$ and $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$. The filters H_0, H_1, G_0, G_1 can also be constructed from outside a wavelet setting, i.e. that they do not originate from change of coordinate matrices between certain function bases. The important point is that the matrices invert each other, but in a signal processing setting it may also be meaningful to allow for the reverse transform not to invert the forward transform exactly. This corresponds to some loss of information when we attempt to reconstruct the original signal using the reverse transform. A small such loss can, as we will see at the end of this chapter, be acceptable.

That the reverse transform inverts the forward transform means that $GH = I$. If we transpose this expression we get that $H^T G^T = I$. Clearly H^T is a reverse filter bank transform with filters $(H_0)^T, (H_1)^T$, and G^T is a forward filter bank transform with filters $(G_0)^T, (G_1)^T$. Due to their usefulness, these transforms have their own name:

Definition 6.7. *Dual filter bank transforms.*
Assume that H_0, H_1 are the filters of a forward filter bank transform, and that G_0, G_1 are the filters of a reverse filter bank transform. By the *dual transforms* we mean the forward filter bank transform with filters $(G_0)^T, (G_1)^T$, and the reverse filter bank transform with filters $(H_0)^T, (H_1)^T$.

In Section 5.3 we used a parameter `dual` in our call to the DWT amd IDWT kernel functions. This parameter can now be explained as follows:

Fact 6.8. *The `dual`-parameter in DWT kernel functions..*

- If the `dual` parameter is false, the DWT is computed as the forward filter bank transform with filters H_0, H_1, and the IDWT is computed as the reverse filter bank transform with filters G_0, G_1.
- If the `dual` parameter is true, the DWT is computed as the forward filter bank transform with filters $(G_0)^T, (G_1)^T$, and the IDWT is computed as the reverse filter bank transform with filters $(H_0)^T, (H_1)^T$.

Note that, even though the reverse filter bank transform G can be associated with certain function bases, it is not clear if the reverse filter bank transform H^T also can be associated with such bases. We will see in the next chapter that such bases can in many cases be found. We will also denote these bases as *dual bases*.

Note that Figure 6.1 and 6.2 do not indicate the additional downsampling and upsampling steps described in Theorem 6.3 and 6.5. If we indicate downsampling with $\downarrow_2$, and upsampling with $\uparrow_2$, the algorithms given in Theorem 6.3 and 6.5 can be summarized as in Figure 6.3.

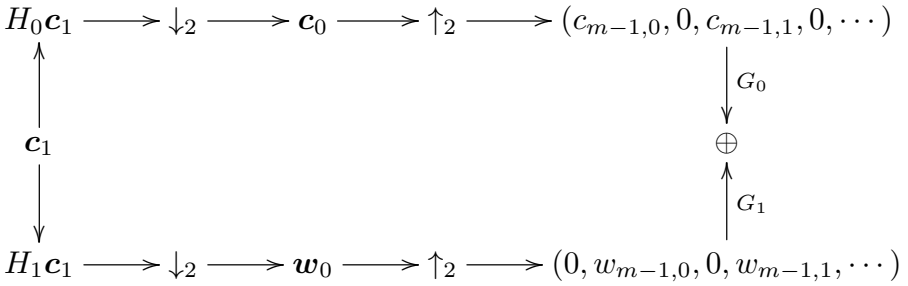

Figure 6.3: Detailed illustration of a DWT.

Here $\oplus$ represents summing the elements which point inwards to the plus sign. In this figure, the left side represents the DWT, the right side the IDWT. In the literature, wavelet transforms are more often illustrated in this way using filters, since it makes alle steps involved in the process more clear. This type of figure also opens for generalization. We will shortly look into this.

There are several reasons why it is smart to express a wavelet transformation in terms of filters. First of all, it enables us to reuse theoretical results from the world of filters in the world of wavelets, and to give useful interpretations of the wavelet transform in terms of frequencies. Secondly, and perhaps most important, it enables us to reuse efficient implementations of filters in order to compute wavelet transformations. A lot of work has been done in order to establish efficient implementations of filters, due to their importance.

In Example 5.19 we argued that the elements in V_{m-1} correspond to frequencies at lower frequencies than those in V_m, since $V_0 = \text{Span}(\{\phi_{0,n}\}_n)$ should be interpreted as content of lower frequency than the $\phi_{1,n}$, with $W_0 = \text{Span}(\{\psi_{0,n}\}_n)$ the remaining high frequency detail. To elaborate more on this, we have that

$$\phi(t) = \sum_{n=0}^{2N-1} (G_0)_{n,0}\phi_{1,n}(t) \tag{6.5}$$

$$\psi(t) = \sum_{n=0}^{2N-1} (G_1)_{n,1}\phi_{1,n}(t)., \tag{6.6}$$

where $(G_k)_{i,j}$ are the entries in the matrix G_k. Similar equations are true for $\phi(t-k), \psi(t-k)$. Due to Equation (6.5), the filter G_0 should have lowpass characteristics, since it extracts the information at lower frequencies. Similarly, G_1 should have highpass characteristics due to Equation (6.6).

Let us verify these lowpass/highpass characteristics of G_0 and G_1 for the wavelets we have considered up to now by plotting their frequency responses. In order to do this we should make a final remark on how these frequency responses can be plotted. For all wavelets we look at the filter coefficients are computed, so that the frequency responses can be easily calculated. However, when we use a wavelet for computation, we applied it by means of a kernel transformation. We will later see that the most efficient such kernel transformations do not apply the filter coefficients directly, but rather a factorization into smaller components (but see Exercise 6.12 on how we can produce kernel transformations which use the filter coefficients directly). So how can we find and plot the frequency response when only the kernel transformation is known? First of all, since the first column of G is identical to the first column of G_0, the first column of G_0 can be obtained by applying the IDWT kernel to the vector $\boldsymbol{e}_0$. We can then use Theorem 3.14 to find the vector frequency response of the filter (i.e. applying an FFT), and then Theorem 3.21 to find the values of the continuous frequency response in the points $2\pi n/N$ for $0 \leq n < N$. The following code can thus be used to plot the frequency response of G_0, when only the IDWT kernel (called `idwtkernel` below) is known.

```
omega = 2*pi*arange(0,N)/float(N)
g0 = concatenate([[1], zeros(N - 1)]) # Creates e_0
idwtkernel(g0, 0, 0)                  # Find the first column of G_0
plot(omega, abs(fft.fft(g0)))
```

A similar procedure can be applied in order to plot the frequency response of G_1 (just replace $\boldsymbol{e}_0$ with $\boldsymbol{e}_1$ in order o exract the second column of G instead). The frequency responses of H_0 and H_1 can be found by considering a dual wavelet transform, since the reverse transform for the dual wavelet has filters $(H_0)^T$ and $(H_1)^T$. In most of the following examples in this book this procedure will be applied to plot all frequency responses. We start with the Haar wavelet.

Example 6.9. *The Haar wavelet.*

For the Haar wavelet we saw that, in G, the matrix

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \tag{6.7}$$

repeated along the diagonal. The filters G_0 and G_1 can be found directly from these columns:

$$G_0 = \{\underline{1/\sqrt{2}}, 1/\sqrt{2}\}$$
$$G_1 = \{1/\sqrt{2}, \underline{-1/\sqrt{2}}\}.$$

We have seen these filters previously: G_0 is a movinge average filter of two elements (up to multiplication with a constant). This is a lowpass filter. G_1 is a bass-reducing filter, which is a highpass filter. Up to a constant, this is also an approximation to the derivative. Since G_1 is constructed from G_0 by adding an alternating sign to the filter coefficients, we know from before that G_1 is the highpass filter corresponding to the lowpass filter G_0, so that the frequency response of the second is given by a shift of frequency with π in the first. The frequency responses are

$$\lambda_{G_0}(\omega) = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}e^{-i\omega} = \sqrt{2}e^{-i\omega/2}\cos(\omega/2)$$
$$\lambda_{G_1}(\omega) = \frac{1}{\sqrt{2}}e^{i\omega} - \frac{1}{\sqrt{2}} = \sqrt{2}ie^{i\omega/2}\sin(\omega/2).$$

The magnitude of these are plotted in Figure 6.4, where the lowpass/highpass characteristics are clearly seen.

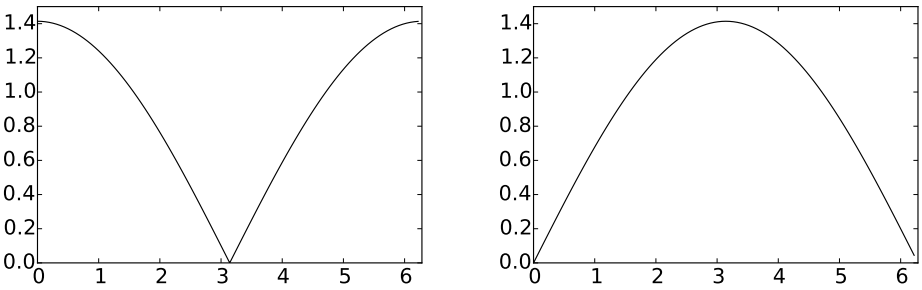

Figure 6.4: The frequency responses $\lambda_{G_0}(\omega)$ (left) and $\lambda_{G_1}(\omega)$ (right) for the Haar wavelet.

By considering the filters where the rows in Equation (6.7), it is clear that

$$H_0 = \{1/\sqrt{2}, \underline{1/\sqrt{2}}\}$$
$$H_1 = \{\underline{-1/\sqrt{2}}, 1/\sqrt{2}\},$$

so that the frequency responses for the DWT have the same lowpass/highpass characteristics.

It turns out that this connection between G_0 and G_1 as lowpass and highpass filters corresponding to each other can be found in all orthonormal wavelets. We will prove this in the next chapter.

Example 6.10. *Wavelet for piecewise linear functions.*

For the wavelet for piecewise linear functions we looked at in the previous section, Equation (6.1) gives that

$$\begin{aligned} G_0 &= \frac{1}{\sqrt{2}}\{1/2, \underline{1}, 1/2\} \\ G_1 &= \frac{1}{\sqrt{2}}\{\underline{1}\}. \end{aligned} \tag{6.8}$$

G_0 is again a filter we have seen before: Up to multiplication with a constant, it is the treble-reducing filter with values from row 2 of Pascal's triangle. We see something different here when compared to the Haar wavelet, in that the filter G_1 is not the highpass filter corresponding to G_0. The frequency responses are now

$$\begin{aligned} \lambda_{G_0}(\omega) &= \frac{1}{2\sqrt{2}}e^{i\omega} + \frac{1}{\sqrt{2}} + \frac{1}{2\sqrt{2}}e^{-i\omega} = \frac{1}{\sqrt{2}}(\cos\omega + 1) \\ \lambda_{G_1}(\omega) &= \frac{1}{\sqrt{2}}. \end{aligned}$$

$\lambda_{G_1}(\omega)$ thus has magnitude $\frac{1}{\sqrt{2}}$ at all points. The magnitude of $\lambda_{G_0}(\omega)$ is plotted in Figure 6.5.

Comparing with Figure 6.4 we see that here also the frequency response has a zero at π. The frequency response seems also to be flatter around π. For the DWT, Equation (6.2) gives us

$$\begin{aligned} H_0 &= \sqrt{2}\{\underline{1}\} \\ H_1 &= \sqrt{2}\{-1/2, \underline{1}, -1/2\}. \end{aligned} \tag{6.9}$$

Even though G_1 was not the highpass filter corresponding to G_0, we see that, up to a constant, H_1 is (it is a bass-reducing filter with values taken from row 2 of Pascals triangle).

Note that the role of H_1 as the highpass filter corresponding to G_0 is the case in both previous examples. We will prove in the next chapter that this is a much more general result which holds for all wavelets, not only for the orthonormal ones.

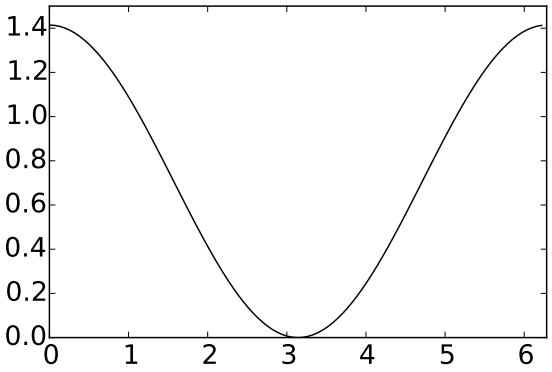

Figure 6.5: The frequency response $\lambda_{G_0}(\omega)$ for the first choice of wavelet for piecewise linear functions.

For the alternative wavelet for piecewise linear functions, we are only able to find expressions for the filters G_0, G_1 at this stage (these can be extracted from Equation (6.3)). In the next chapter we will learn a general technique of computing the transformations the opposite way from these, so this will be handled in the next chapter.

6.1.1 The support of the scaling function and the mother wavelet

The scaling functions and mother wavelets we encounter will turn out to always be functions with compact support. An interesting consequence of equations (6.5) and (6.6) is that we can find the size of these supports from the number of filter coefficients in G_0 and G_1:

Theorem 6.11. *Support size.*

Assume that the filters G_0, G_1 have N_0, N_1 nonzero filter coefficients, respectively, and that ϕ and ψ have compact support. Then the support size of ϕ is $N_0 - 1$, and the support size of ψ is $(N_0 + N_1)/2 - 1$. Moreover, when all the filters are symmetric, the support of ϕ is symmetric around 0, and the support of ψ is symmetric around $1/2$.

Proof. Let q be the support size of ϕ. Then the functions $\phi_{1,n}$ all have support size $q/2$. On the right hand side of Equation (6.5) we thus add N_0 functions, all with support size $q/2$. These functions are translated with $1/2$ with respect to oneanother, so that the sum has support size $q/2 + (N_0 - 1)/2$. Comparing with the support of the left hand side we get the equation $q = q/2 + (N_0 - 1)/2$, so that $q = N_0 - 1$. Similarly, with Equation (6.6), the function on the right hand side has support size $q/2 + (N_1 - 1)/2 = (N_0 + N_1)/2 - 1$, which thus is the support of ψ.

Assume now also that all filters are symmetric, so that the nonzero filter coefficients of G_0 have indices $-(N_0-1)/2, \ldots, (N_0-1)/2$. If ϕ has support $[q_1, q_2]$, $\phi_{1,n}$ has support $[(q_1+n)/2, (q_2+n)/2]$. It follows that the right hand side of Equation (6.5) has support $[(q_1 - (N_0-1)/2)/2, (q_2 + (N_0-1)/2)/2]$, so that we obtain the equations

$$q_1 = (q_1 - (N_0-1)/2)/2, \text{ and } q_2 = (q_2 + (N_0-1)/2)/2.$$

Solving these we obtain that $q_1 = -(N_0-1)/2$, $q_2 = (N_0-1)/2$, so that the support of ϕ is symmetric around 0. Similarly, the right hand side of Equation (6.6) has support

$$\begin{aligned}\left[\frac{q_1 - \frac{N_1-3}{2}}{2}, \frac{q_2 + \frac{N_1+1}{2}}{2}\right] &= \left[\frac{-\frac{N_0-1}{2} - \frac{N_1-3}{2}}{2}, \frac{\frac{N_0-1}{2} + \frac{N_1+1}{2}}{2}\right] \\ &= \left[-\frac{\frac{N_0+N_1}{2} - 1}{2}, \frac{\frac{N_0+N_1}{2} - 1}{2}\right] + 1.\end{aligned}$$

From this it is clear that ψ has support symmetric around $1/2$. □

Let us use this theorem to verify the supports for the scaling functions and mother wavelets we have already encountered:

- For the Haar wavelet, we know that both filters have 2 coefficients. From Theorem 6.11 it follows that both ϕ and ψ have support size 1, which clearly is true.
- For the the piecewise linear wavelet, the filters were symmetric. G_0 has 3 filter coefficients so that ϕ has support size $3-1=2$. G_1 has one filter coefficient, so that the support size of ψ is $(3+1)/2 - 1 = 1$. We should thus have that $\text{supp}(\phi) = [-1, 1]$, and $\text{supp}(\psi) = [0, 1]$. This is clearly true from our previous plots of these functions.
- From Equation (6.3) we see that, for the alternative piecewise linear wavelet, G_0 and G_1 have 3 and 5 filter coefficients, respectively. ψ has thus support size $(3+5)/2 - 1 = 3$, so that the support is $[-1, 2]$, which also can be seen to be the case from Figure 5.18.

6.1.2 Wavelets and symmetric extensions

In practice we want to apply the wavelet transform to a symmetric extension, since then symmetric filters can give a better approximation to the underlying analog filters. In order to achieve this, the following result says that we only need to replace the filters H_0, H_1, G_0, and G_1 in the wavelet transform with $(H_0)_r$, $(H_1)_r$, $(G_0)_r$, and $(G_1)_r$.

Theorem 6.12. *Symmetric filters and symmetric extensions.*

If the filters H_0, H_1, G_0, and G_1 in a wavelet transform are symmetric, then the DWT/IDWT preserve symmetric extensions (as defined in Definition 5.42). Also, applying the filters H_0, H_1, G_0, and G_1 to $\breve{\boldsymbol{x}} \in \mathbb{R}^{2N-2}$ in the DWT/IDWT is equivalent to applying $(H_0)_r$, $(H_1)_r$, $(G_0)_r$, and $(G_1)_r$ to $\boldsymbol{x} \in \mathbb{R}^N$ in the same way.

Proof. Since H_0 and H_1 are symmetric, their output from $\breve{\boldsymbol{x}}$ is also a symmetric vector, and by assembling their outputs as the even- and odd-indexed entries, we see that the output $(c_0, w_0, c_1, w_1, \ldots)$ of the MRA-matrix H also is a symmetric vector. The same then applies for the matrix G, since it inverts the first. This proves the first part.

Now, assume that $\boldsymbol{x} \in \mathbb{R}^N$. By definition of $(H_i)_r$, $(H_i\breve{\boldsymbol{x}})_n = ((H_i)_r\boldsymbol{x})_n$ for $0 \leq n \leq N-1$. This means that we get the same first N output elements in a wavelet transform if we repace H_0, H_1 with $(H_0)_r, (H_1)_r$. Since the vectors $(c_0, 0, c_1, 0, \ldots)$ and $(0, w_0, 0, w_1, \ldots)$ also are symmetric vectors when $(c_0, w_0, c_1, w_1, \ldots)$ is, it follows that $(G_0)_r, (G_1)_r$ will reproduce the same first N elements as G_0, G_1 also. In conclusion, for symmetric vectors, the wavelet transform restricted to the first N elements produces the same result when we replace H_0, H_1, G_0, and G_1 with $(H_0)_r$, $(H_1)_r$, $(G_0)_r$, and $(G_1)_r$. This proves the result. □

As in Chapter 4, it follows that when the filters of a wavelet are symmetric, applying $(H_0)_r$, $(H_1)_r$, $(G_0)_r$, and $(G_1)_r$ to the input better approximates an underlying analog filter.

In Section 5.3 we used a parameter `symm` in our call to the DWT amd IDWT kernel functions. This parameter can now also be explained:

idxDWT kernel parameter `symm`

Fact 6.13. *The `symm`-parameter in DWT kernel functions.*

Assume that the filters H_0, H_1, G_0, and G_1 are symmetric. If the `symm` parameter is true, the symmetric versions $(H_0)_r$, $(H_1)_r$, $(G_0)_r$, and $(G_1)_r$ should be applied in the DWT and IDWT, rather than the filters H_0, H_1, G_0, and G_1 themselves. If `symm` is false, the filters H_0, H_1, G_0, and G_1 are applied

In Chapter 8 we will also see how the symmetric versions $(H_0)_r$, $(H_1)_r$, $(G_0)_r$ can be implemented.

What you should have learned in this section.

- How one can find the filters of a wavelet transformation by considering its matrix and its inverse.
- Forward and reverse filter bank transforms.
- How one can implement the DWT and the IDWT with the help of the filters.

- Plot of the frequency responses for the filters of the wavelets we have considered, and their interpretation as lowpass and highpass filters.

Exercise 6.1: Compute filters and frequency responses 1

Write down the corresponding filters G_0 og G_1 for Exercise 5.32. Plot their frequency responses, and characterize the filters as lowpass- or highpass filters.

Exercise 6.2: Symmetry of MRA matrices vs. symmetry of filters 1

Find two symmetric filters, so that the corresponding MRA-matrix, constructed with alternating rows from these two filters, is not a symmetric matrix.

Exercise 6.3: Symmetry of MRA matrices vs. symmetry of filters 2

Assume that an MRA-matrix is symmetric. Are the corresponding filters H_0, H_1, G_0, G_1 also symmetric? If not, find a counterexample.

Exercise 6.4: Finding H_0, H_1 from the H

Assume that one stage in a DWT is given by the MRA-matrix

$$H = \begin{pmatrix} 1/5 & 1/5 & 1/5 & 0 & 0 & 0 & \cdots & 0 & 1/5 & 1/5 \\ -1/3 & 1/3 & -1/3 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & -1/3 & 1/3 & -1/3 & 0 & \cdots 0 & 0 & & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

Write down the compact form for the corresponding filters H_0, H_1, and compute and plot the frequency responses. Are the filters symmetric?

Exercise 6.5: Finding G_0, G_1 from the G

Assume that one stage in the IDWT is given by the MRA-matrix

$$G = \begin{pmatrix} 1/2 & -1/4 & 0 & 0 & \cdots \\ 1/4 & 3/8 & 1/4 & 1/16 & \cdots \\ 0 & -1/4 & 1/2 & -1/4 & \cdots \\ 0 & 1/16 & 1/4 & 3/8 & \cdots \\ 0 & 0 & 0 & -1/4 & \cdots \\ 0 & 0 & 0 & 1/16 & \cdots \\ 0 & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots \\ 1/4 & 1/16 & 0 & 0 & \cdots \end{pmatrix}$$

Write down the compact form for the filters G_0, G_1, and compute and plot the frequency responses. Are the filters symmetric?

Exercise 6.6: Finding H from H_0, H_1

Assume that $H_0 = \{1/16, 1/4, \underline{3/8}, 1/4, 1/16\}$, and $H_1 = \{-1/4, \underline{1/2}, -1/4\}$. Plot the frequency responses of H_0 and H_1, and verify that H_0 is a lowpass filter, and that H_1 is a highpass filter. Also write down the change of coordinate matrix $P_{\mathcal{C}_1 \leftarrow \boldsymbol{\phi}_1}$ for the wavelet corresponding to these filters.

Exercise 6.7: Finding G from G_0, G_1

Assume that $G_0 = \frac{1}{3}\{1, \underline{1}, 1\}$, and $G_1 = \frac{1}{5}\{1, -1, \underline{1}, -1, 1\}$. Plot the frequency responses of G_0 and G_1, and verify that G_0 is a lowpass filter, and that G_1 is a highpass filter. Also write down the change of coordinate matrix $P_{\boldsymbol{\phi}_1 \leftarrow \mathcal{C}_1}$ for the wavelet corresponding to these filters.

Exercise 6.8: Computing by hand

In Exercise 5.17 we computed the DWT of two very simple vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, using the Haar wavelet.

a) Compute $H_0\boldsymbol{x}_1$, $H_1\boldsymbol{x}_1$, $H_0\boldsymbol{x}_2$, and $H_1\boldsymbol{x}_2$, where H_0 and H_1 are the filters used by the Haar wavelet.

b) Compare the odd-indexed elements in $H_1\boldsymbol{x}_1$ with the odd-indexed elements in $H_1\boldsymbol{x}_2$. From this comparison, attempt to find an explanation to why the two vectors have very different detail components.

Exercise 6.9: Comment code

Suppose that we run the following algorithm on the sound represented by the vector `x`:

```
c = (x[0::2] + x[1::2])/sqrt(2)
w = (x[0::2] - x[1::2])/sqrt(2)

newx = concatenate([c, w])
newx /= abs(newx).max()
play(newx,44100)
```

a) Comment the code and explain what happens. Which wavelet is used? What do the vectors `c` and `w` represent? Describe the sound you believe you will hear.

b) Assume that we add lines in the code above which sets the elements in the vector `w` to 0 before we compute the inverse operation. What will you hear if you play the new sound you then get?

Exercise 6.10: Computing filters and frequency responses 1

Let us return to the piecewise linear wavelet from Exercise 5.31.

a) With $\hat{\psi}$ as defined as in Exercise 5.31b), compute the coordinates of $\hat{\psi}$ in the basis $\boldsymbol{\phi}_1$ (i.e. $[\hat{\psi}]_{\boldsymbol{\phi}_1}$) with $N = 8$, i.e. compute the IDWT of

$$[\hat{\psi}]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} = (-\alpha, -\beta, -\delta, 0, 0, 0, 0, -\gamma) \oplus (1, 0, 0, 0, 0, 0, 0, 0),$$

which is the coordinate vector you computed in Exercise 5.31d). For this, you should use the function `IDWTImpl`, with the kernel of the piecewise linear wavelet without symmetric extension as input. Explain that this gives you the filter coefficients of G_1.

b) Plot the frequency response of G_1.

Exercise 6.11: Computing filters and frequency responses 2

Repeat the previous exercise for the Haar wavelet as in Exercise 5.33, and plot the corresponding frequency responses for $k = 2, 4, 6$.

Exercise 6.12: Implementing with symmetric extension

In Exercise 3.6 we implemented a symmetric filter applied to a vector, i.e. when a periodic extension is assumed. The corresponding function was called `filterS(t, x)`, and used the function `numpy.convolve`.

a) Reimplement the function `filterS` so that it also takes a third parameter `symm`. If `symm` is false a periodic extension of `x` should be performed (i.e. filtering as we have defined it, and as the previous version of `filterS` performs it). If `symm` is true, symmetric extensions should be used (as given by Definition 5.42).

b) Implement functions `DWTKernelFilters(H0, H1, G0, G1, x, symm, dual)` and `IDWTKernelFilters(H0, H1, G0, G1, x, symm, dual)` which compute the DWT and IDWT kernels using theorems 6.3 and 6.5, respectively. This function thus bases itself on that the filters of the wavelet are known. The functions should call the function `filterS` from a). Recall also the definition of the parameter `dual` from this section.

With the functions defined in b. you can now define standard DWT and IDWT kernels in the following way, once the filters are known.

```
f    = lambda x, symm, dual: DWTKernelFilters(H0,H1,G0,G1,x,symm,dual)
invf = lambda x, symm, dual: IDWTKernelFilters(H0,H1,G0,G1,x,symm,dual)
```

6.2 Properties of the filter bank transforms of a wavelet

We have now described the DWT/IDWT as linear transformations G, H so that $GH = I$, and where two filters G_0, G_1 characterize G, two filters H_0, H_1 characterize H. G and H are not Toeplitz matrices, however, so they are not filters. Since filters produce the same output frequency from an input frequency, we must have that G and H produce other (undesired) frequencies in the output than those that are present in the input. We will call this phenomenon *aliasing*. In order for $GH = I$, the undesired frequencies must cancel each other, so that we end up with what we started with. Thus, GH must have what we will refer to as *alias cancellation*. This is the same as saying that GH is a filter. In order for $GH = I$, alias cancellation is not enough: We also need that the amount at the given frequency is unchanged, i.e. that $GH\phi_n = \phi_n$ for any Fourier basis vector ϕ_n. We then say that we have *perfect reconstruction*. Perfect reconstruction is always the case for wavelets by construction, but in signal processing many interesting examples (G_0, G_1, H_0, H_1) exist, for which we do not have perfect reconstruction. Historically, forward and reverse filter bank transforms have been around long before they appeared in a wavelet context. Operations where $GH\phi_n = c_n\phi_n$ for all n may also be useful, in particular when c_n is close to 1 for all n. If c_n is real for all n, we say that we have *no phase distortion*. If we have no phase distortion, the output from GH has the same phase, even if we do not have perfect reconstruction. Such "near-perfect reconstruction systems" have also been around long before many perfect reconstruction wavelet systems were designed. In signal processing, these transforms also exist in more general variants, and we will define these later. Let us summarize as follows.

Definition 6.14. *Alias cancellation, phase distortion, and perfect reconstruction.*

We say that we have *alias cancellation* if, for any n,

$$GH\phi_n = c_n\phi_n,$$

for some constant c_n (i.e. GH is a filter). If all c_n are real, we say that we *no phase distortion*. If $GH = I$ (i.e. $c_n = 1$ for all n) we say that we have

perfect reconstruction. If all c_n are close to 1, we say that we have *near-perfect reconstruction*.

In signal processing, one also says that we have perfect- or near-perfect reconstruction when GH equals E_d, or is close to E_d (i.e. the overall result is a delay). The reason why a delay occurs has to do with that the transforms are used in real-time processing, for which we may not be able to compute the output at a given time instance before we know some of the following samples. Clearly the delay is unproblematic, since one can still can reconstruct the input from the output. We will encounter a useful example of near-perfect reconstruction soon in the MP3 standard.

Let us now find a criterium for alias cancellation: When do we have that $GHe^{2\pi irk/N}$ is a multiplum of $e^{2\pi irk/N}$, for any r? We first remark that

$$H(e^{2\pi irk/N}) = \begin{cases} \lambda_{H_0,r} e^{2\pi irk/N} & k \text{ even} \\ \lambda_{H_1,r} e^{2\pi irk/N} & k \text{ odd.} \end{cases}$$

The frequency response of $H(e^{2\pi irk/N})$ is

$$\begin{aligned} &\sum_{k=0}^{N/2-1} \lambda_{H_0,r} e^{2\pi ir(2k)/N} e^{-2\pi i(2k)n/N} + \sum_{k=0}^{N/2-1} \lambda_{H_1,r} e^{2\pi ir(2k+1)/N} e^{-2\pi i(2k+1)n/N} \\ &= \sum_{k=0}^{N/2-1} \lambda_{H_0,r} e^{2\pi i(r-n)(2k)/N} + \sum_{k=0}^{N/2-1} \lambda_{H_1,r} e^{2\pi i(r-n)(2k+1)/N} \\ &= (\lambda_{H_0,r} + \lambda_{H_1,r} e^{2\pi i(r-n)/N}) \sum_{k=0}^{N/2-1} e^{2\pi i(r-n)k/(N/2)}. \end{aligned}$$

Clearly, $\sum_{k=0}^{N/2-1} e^{2\pi i(r-n)k/(N/2)} = N/2$ if $n = r$ or $n = r + N/2$, and 0 else. The frequency response is thus the vector

$$\frac{N}{2}(\lambda_{H_0,r} + \lambda_{H_1,r})\boldsymbol{e}_r + \frac{N}{2}(\lambda_{H_0,r} - \lambda_{H_1,r})\boldsymbol{e}_{r+N/2},$$

so that

$$H(e^{2\pi irk/N}) = \frac{1}{2}(\lambda_{H_0,r} + \lambda_{H_1,r})e^{2\pi irk/N} + \frac{1}{2}(\lambda_{H_0,r} - \lambda_{H_1,r})e^{2\pi i(r+N/2)k/N}. \tag{6.10}$$

Let us now turn to the reverse filter bank transform. We can write

$$\begin{aligned} (e^{2\pi ir\cdot 0/N}, 0, e^{2\pi ir\cdot 2/N}, 0, \ldots, e^{2\pi ir(N-2)/N}, 0) &= \frac{1}{2}(e^{2\pi irk/N} + e^{2\pi i(r+N/2)k/N}) \\ (0, e^{2\pi ir\cdot 1/N}, 0, e^{2\pi ir\cdot 3/N}, \ldots, 0, e^{2\pi ir(N-1)/N}) &= \frac{1}{2}(e^{2\pi irk/N} - e^{2\pi i(r+N/2)k/N}). \end{aligned}$$

This means that

$$\begin{aligned}
&G(e^{2\pi irk/N}) = G_0\left(\frac{1}{2}\left(e^{2\pi irk/N} + e^{2\pi i(r+N/2)k/N}\right)\right) + G_1\left(\frac{1}{2}\left(e^{2\pi irk/N} - e^{2\pi i(r+N/2)k/N}\right)\right)\\
=&\frac{1}{2}(\lambda_{G_0,r}e^{2\pi irk/N} + \lambda_{G_0,r+N/2}e^{2\pi i(r+N/2)k/N}) + \frac{1}{2}(\lambda_{G_1,r}e^{2\pi irk/N} - \lambda_{G_1,r+N/2}e^{2\pi i(r+N/2)k/N})\\
=&\frac{1}{2}(\lambda_{G_0,r} + \lambda_{G_1,r})e^{2\pi irk/N} + \frac{1}{2}(\lambda_{G_0,r+N/2} - \lambda_{G_1,r+N/2})e^{2\pi i(r+N/2)k/N}. \quad (6.11)
\end{aligned}$$

Now, if we combine equations (6.10) and (6.11), we get

$$\begin{aligned}
&GH(e^{2\pi irk/N})\\
&= \frac{1}{2}(\lambda_{H_0,r} + \lambda_{H_1,r})G(e^{2\pi irk/N}) + \frac{1}{2}(\lambda_{H_0,r} - \lambda_{H_1,r})G(e^{2\pi i(r+N/2)k/N})\\
&= \frac{1}{2}(\lambda_{H_0,r} + \lambda_{H_1,r})\left(\frac{1}{2}(\lambda_{G_0,r} + \lambda_{G_1,r})e^{2\pi irk/N} + \frac{1}{2}(\lambda_{G_0,r+N/2} - \lambda_{G_1,r+N/2})e^{2\pi i(r+N/2)k/N})\right)\\
&+ \frac{1}{2}(\lambda_{H_0,r} - \lambda_{H_1,r})\left(\frac{1}{2}(\lambda_{G_0,r+N/2} + \lambda_{G_1,r+N/2})e^{2\pi i(r+N/2)k/N} + \frac{1}{2}(\lambda_{G_0,r} - \lambda_{G_1,r})e^{2\pi irk/N})\right)\\
&= \frac{1}{4}\left((\lambda_{H_0,r} + \lambda_{H_1,r})(\lambda_{G_0,r} + \lambda_{G_1,r}) + (\lambda_{H_0,r} - \lambda_{H_1,r})(\lambda_{G_0,r} - \lambda_{G_1,r})\right)e^{2\pi irk/N}\\
&+ \frac{1}{4}\left((\lambda_{H_0,r} + \lambda_{H_1,r})(\lambda_{G_0,r+N/2} - \lambda_{G_1,r+N/2}) + (\lambda_{H_0,r} - \lambda_{H_1,r})(\lambda_{G_0,r+N/2} + \lambda_{G_1,r+N/2})\right)e^{2\pi i(r+N/2)k/N}\\
&= \frac{1}{2}(\lambda_{H_0,r}\lambda_{G_0,r} + \lambda_{H_1,r}\lambda_{G_1,r})e^{2\pi irk/N} + \frac{1}{2}(\lambda_{H_0,r}\lambda_{G_0,r+N/2} - \lambda_{H_1,r}\lambda_{G_1,r+N/2})e^{2\pi i(r+N/2)k/N}.
\end{aligned}$$

If we also replace with the continuous frequency response, we obtain the following:

Theorem 6.15. *Expression for aliasing.*
We have that

$$\begin{aligned}
GH(e^{2\pi irk/N}) =&\frac{1}{2}(\lambda_{H_0,r}\lambda_{G_0,r} + \lambda_{H_1,r}\lambda_{G_1,r})e^{2\pi irk/N}\\
&+ \frac{1}{2}(\lambda_{H_0,r}\lambda_{G_0,r+N/2} - \lambda_{H_1,r}\lambda_{G_1,r+N/2})e^{2\pi i(r+N/2)k/N}.
\end{aligned} \quad (6.12)$$

In particular, we have alias cancellation if and only if

$$\lambda_{H_0}(\omega)\lambda_{G_0}(\omega+\pi) = \lambda_{H_1}(\omega)\lambda_{G_1}(\omega+\pi). \quad (6.13)$$

We will refer to this as the *alias cancellation condition.* If in addition

$$\lambda_{H_0}(\omega)\lambda_{G_0}(\omega) + \lambda_{H_1}(\omega)\lambda_{G_1}(\omega) = 2, \quad (6.14)$$

we also have perfect reconstruction. We will refer to as the *condition for perfect reconstruction.*

No phase distortion means that we have alias cancellation, and that

$$\lambda_{H_0}(\omega)\lambda_{G_0}(\omega) + \lambda_{H_1}(\omega)\lambda_{G_1}(\omega) \text{ is real.}$$

Now let us turn to how we can construct wavelets/perfect reconstruction systems from FIR-filters (recall from Chapter 3 that FIR filters where filters with a finite number of filter coefficients). We will have use for some theorems which allow us to construct wavelets from prototype filters. In particular we show that, when G_0 and H_0 are given lowpass filters which satisfy a certain common property, we can define unique (up to a constant) highpass filters H_1 and G_1 so that the collection of these four filters can be used to implement a wavelet. We first state the following general theorem.

Theorem 6.16. *Criteria for perfect reconstruction.*
The following statements are equivalent for FIR filters H_0, H_1, G_0, G_1:

- H_0, H_1, G_0, G_1 give perfect reconstruction,
- there exist $\alpha \in \mathbb{R}$ and $d \in \mathbb{Z}$ so that

$$(H_1)_n = (-1)^n \alpha^{-1} (G_0)_{n-2d} \tag{6.15}$$
$$(G_1)_n = (-1)^n \alpha (H_0)_{n+2d} \tag{6.16}$$
$$2 = \lambda_{H_0,n}\lambda_{G_0,n} + \lambda_{H_0,n+N/2}\lambda_{G_0,n+N/2} \tag{6.17}$$

Let us translate this to continuous frequency responses. We first have that

$$\begin{aligned}\lambda_{H_1}(\omega) &= \sum_k (H_1)_k e^{-ik\omega} = \sum_k (-1)^k \alpha^{-1} (G_0)_{k-2d} e^{-ik\omega} \\ &= \alpha^{-1} \sum_k (-1)^k (G_0)_k e^{-i(k+2d)\omega} = \alpha^{-1} e^{-2id\omega} \sum_k (G_0)_k e^{-ik(\omega+\pi)} \\ &= \alpha^{-1} e^{-2id\omega} \lambda_{G_0}(\omega + \pi).\end{aligned}$$

We have a similar computation for $\lambda_{G_1}(\omega)$. We can thus state the following:

Theorem 6.17. *Criteria for perfect reconstruction.*
The following statements are equivalent for FIR filters H_0, H_1, G_0, G_1:

- H_0, H_1, G_0, G_1 give perfect reconstruction,
- there exist $\alpha \in \mathbb{R}$ and $d \in \mathbb{Z}$ so that

$$\lambda_{H_1}(\omega) = \alpha^{-1} e^{-2id\omega} \lambda_{G_0}(\omega + \pi) \tag{6.18}$$
$$\lambda_{G_1}(\omega) = \alpha e^{2id\omega} \lambda_{H_0}(\omega + \pi) \tag{6.19}$$
$$2 = \lambda_{H_0}(\omega)\lambda_{G_0}(\omega) + \lambda_{H_0}(\omega + \pi)\lambda_{G_0}(\omega + \pi) \tag{6.20}$$

Proof. Let us prove first that equations (6.18)- (6.20) for a FIR filter implies that we have perfect reconstruction. Equations (6.18)-(6.19) mean that the alias cancellation condition (6.13) is satisfied, since

$$\begin{aligned}\lambda_{H_1}(\omega)\lambda_{G_1}(\omega+\pi) =& \alpha^{-1}e^{-2id\omega}\lambda_{G_0}(\omega+\pi)(\alpha)e^{2id(\omega+\pi}\lambda_{H_0}(\omega)\\ =& \lambda_{H_0}(\omega)\lambda_{G_0}(\omega+\pi).\end{aligned}$$

Inserting this in the perfect reconstruction condition (6.20), we get

$$\begin{aligned}2 &= \lambda_{H_0}(\omega)\lambda_{G_0}(\omega)+\lambda_{G_0}(\omega+\pi)\lambda_{H_0}(\omega+\pi)\\ &= \lambda_{H_0}(\omega)\lambda_{G_0}(\omega)+\alpha^{-1}e^{-2id\omega}\lambda_{G_0}(\omega+\pi)\alpha e^{2id\omega}\lambda_{H_0}(\omega+\pi)\\ &= \lambda_{H_0}(\omega)\lambda_{G_0}(\omega)+\lambda_{H_1}(\omega)\lambda_{G_1}(\omega),\end{aligned}$$

which is Equation (6.14), so that equations (6.18)- (6.20) imply perfect reconstruction. We therefore only need to prove that any set of FIR filters which give perfect reconstruction, also satisfy these equations. Due to the calculation above, it is enough to prove that equations (6.18)-(6.19) are satisfied. The proof of this will wait till Section 8.1, since it uses some techniques we have not introduced yet. □

Note that, even though conditions (6.18) and (6.19) together ensure that the alias cancellation condition is satisfied, alias cancellation can occur also if these conditions are not satisfied. Conditions (6.18) and (6.19) thus give a stronger requirement than alias cancellation. We will be particularly concerned with wavelets where the filters are symmetric, for which we can state the following corollary.

Corollary 6.18. *Criteria for perfect reconstruction* .
The following statements are equivalent:

- H_0, H_1, G_0, G_1 are the filters of a symmetric wavelet,
- $\lambda_{H_0}(\omega), \lambda_{H_1}(\omega), \lambda_{G_0}(\omega), \lambda_{G_1}(\omega)$ are real functions, and

$$\lambda_{H_1}(\omega) = \alpha^{-1}\lambda_{G_0}(\omega+\pi) \tag{6.21}$$
$$\lambda_{G_1}(\omega) = \alpha\lambda_{H_0}(\omega+\pi) \tag{6.22}$$
$$2 = \lambda_{H_0}(\omega)\lambda_{G_0}(\omega)+\lambda_{H_0}(\omega+\pi)\lambda_{G_0}(\omega+\pi). \tag{6.23}$$

Thw delay d is thus 0 for symmetric wavelets.

Proof. Since H_0 is symmetric, $(H_0)_n = (H_0)_{-n}$, and from equations (6.15) and (6.16) it follows that

$$\begin{aligned}(G_1)_{n-2d} &= (-1)^{n-2d}\alpha(H_0)_n = (-1)^n\alpha^{-1}(H_0)_{-n}\\ &= (-1)^{(-n-2d)}\alpha^{-1}(H_0)_{(-n-2d)+2d} = (G_1)_{-n-2d}\end{aligned}$$

This shows that G_1 is symmetric about both $-2d$, in addition to being symmetric about 0 (by assumption). We must thus have that $d = 0$, so that $(H_1)_n = (-1)^n \alpha (G_0)_n$ and $(G_1)_n = (-1)^n \alpha^{-1} (H_0)_n$. We now get that

$$\begin{aligned}\lambda_{H_1}(\omega) &= \sum_k (H_1)_k e^{-ik\omega} = \alpha^{-1} \sum_k (-1)^k (G_0)_k e^{-ik\omega} \\ &= \alpha^{-1} \sum_k e^{-ik\pi} (G_0)_k e^{-ik\omega} = \alpha^{-1} \sum_k (G_0)_k e^{-ik(\omega+\pi)} \\ &= \alpha^{-1} \lambda_{G_0}(\omega + \pi),\end{aligned}$$

which proves Equation (6.21). Equation (6.21) follows similarly. □

When constructing a wavelet it may be that we know one of the two pairs (G_0, G_1), (H_0, H_1), and that we would like to construct the other two. This can be achieved if we can find the constants d and α from above. If the filters are symmetric we just saw that $d = 0$. If G_0, G_1 are known, it follows from from equations (6.15) and(6.16) that

$$1 = \sum_n (G_1)_n (H_1)_n = \sum_n (G_1)_n \alpha^{-1} (-1)^n (G_0)_n = \alpha^{-1} \sum_n (-1)^n (G_0)_n (G_1)_n,$$

so that $\alpha = \sum_n (-1)^n (G_0)_n (G_1)_n$. On the other hand, if H_0, H_1 are known instead, we must have that

$$1 = \sum_n (G_1)_n (H_1)_n = \sum_n \alpha (-1)^n (H_0)_n (H_1)_n = \alpha \sum_n (-1)^n (H_0)_n (H_1)_n,$$

so that $\alpha = 1/(\sum_n (-1)^n (H_0)_n (H_1)_n)$. Let us use these observations to state the filters for the alternative wavelet of piecewise linear functions, which is the only wavelet we have gone through we have not computed the filters and the frequency response for.

Example 6.19. *The alternative piecewise linear wavelet.*

In Equation (6.3) we wrote down the first two columns in $P_{\boldsymbol{\phi}_m \leftarrow \mathcal{C}_m}$ for the alternative piecewise linear wavelet. This gives us that the filters G_0 ans G_1 are

$$\begin{aligned}G_0 &= \frac{1}{\sqrt{2}} \{1/2, \underline{1}, 1/2\} \\ G_1 &= \frac{1}{\sqrt{2}} \{-1/8, -1/4, \underline{3/4}, -1/4, -1/8\}.\end{aligned} \tag{6.24}$$

Here G_0 was as for the wavelet of piecewise linear functions since we use the same scaling function. G_1 was changed, however. Let us use Theorem 6.17 and the remark above to compute the two remaining filters H_0 and H_1. These filters

are also symmetric, since G_0, G_1 were. From the simple computation above we get that

$$\alpha = \sum_n (-1)^n (G_0)_n (G_1)_n = \frac{1}{2}\left(-\frac{1}{2}\left(-\frac{1}{4}\right) + 1 \cdot \frac{3}{4} - \frac{1}{2}\left(-\frac{1}{4}\right)\right) = \frac{1}{2}.$$

Theorem 6.17 now gives

$$\begin{aligned}(H_0)_n &= \alpha^{-1}(-1)^n (G_1)_n = 2(-1)^n (G_1)_n \\ (H_1)_n &= \alpha^{-1}(-1)^n (G_0)_n = 2(-1)^n (G_0)_n,\end{aligned} \tag{6.25}$$

so that

$$\begin{aligned}H_0 &= \sqrt{2}\{-1/8, 1/4, \underline{3/4}, 1/4, -1/8\} \\ H_1 &= \sqrt{2}\{-1/2, \underline{1}, -1/2\}.\end{aligned} \tag{6.26}$$

We now have that

$$\begin{aligned}\lambda_{G_1}(\omega) &= -1/(8\sqrt{2})e^{2i\omega} - 1/(4\sqrt{2})e^{i\omega} + 3/(4\sqrt{2}) - 1/(4\sqrt{2})e^{-i\omega} - 1/(8\sqrt{2})e^{-2i\omega} \\ &= -\frac{1}{4\sqrt{2}}\cos(2\omega) - \frac{1}{2\sqrt{2}}\cos\omega + \frac{3}{4\sqrt{2}}.\end{aligned}$$

The magnitude of $\lambda_{G_1}(\omega)$ is plotted in Figure 6.6. Clearly, G_1 now has highpass characteristics, while the lowpass characteristic of G_0 has been preserved.

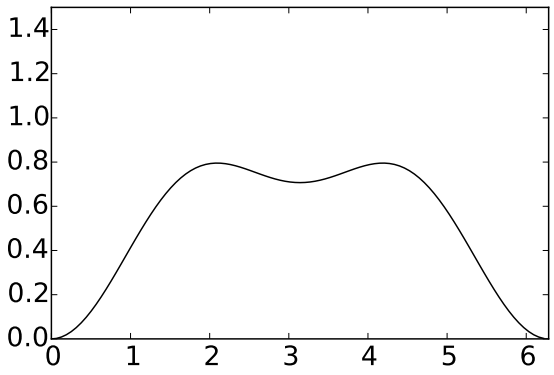

Figure 6.6: The frequency response $\lambda_{G_1}(\omega)$ for the alternative wavelet for piecewise linear functions.

The filters G_0, G_1, H_0, H_1 are particularly important in applications: Apart from the scaling factors $1/\sqrt{2}$, $\sqrt{2}$ in front, we see that the filter coefficients are all dyadic fractions, i.e. they are on the form $\beta/2^j$. Arithmetic operations with dyadic fractions can be carried out exactly on a computer, due to representations as binary numbers in computers. These filters are thus important in applications, since they can be used as transformations for lossless coding. The same argument can be made for the Haar wavelet, but this wavelet had one less vanishing moment.

In the literature, two particular cases of filter banks have been important. They are both refered to as *Quadrature Mirror Filter banks*, or QMF filter banks, and some confusion exist between the two. Let us therefore make precise definitions of the two.

Definition 6.20. *Classical QMF filter banks.*

In the classical definition of a QMF filter banks it is required that $G_0 = H_0$ and $G_1 = H_1$ (i.e. the filters in the forward and reverse transforms are equal), and that

$$\lambda_{H_1}(\omega) = \lambda_{H_0}(\omega + \pi). \tag{6.27}$$

It is straightforward to check that, for a classical QMF filter bank, the forward and reverse transforms are equal (i.e. $G = H$). It is easily checked that conditions (6.18) and (6.19) are satisfied with $\alpha = 1, d = 0$ for a classical QMF filter bank. In particular, the alias cancellation condition is satisfied. The perfect recontruction condition can be written as

$$2 = \lambda_{H_0}(\omega)\lambda_{G_0}(\omega) + \lambda_{H_1}(\omega)\lambda_{G_1}(\omega) = \lambda_{H_0}(\omega)^2 + \lambda_{H_0}(\omega + \pi)^2. \tag{6.28}$$

Unfortunately, it is impossible to find non-trivial FIR-filters which satisfy this quadrature formula (Exercise 6.13). Therefore, classical QMF filter banks which give perfect reconstruction do not exist. Nevertheless, one can construct such filter banks which give close to perfect reconstruction [19], and this together with the fulfillment of the alias cancellation condition still make them useful. In fact, we will see in Section 8.3 that the MP3 standard take use of such filters, and this explains our previous observation that the MP3 standard does not give perfect reconstruction. Note, however, that if the filters in a classical QMF filter bank are symmetric (so that $\lambda_{H_0}(\omega)$ is real), we have no phase distortion.

The second type of QMF filter bank is defined as follows.

Definition 6.21. *Alternative QMF filter banks.*

In the alternative definition of a QMF filter bank it is required that $G_0 = (H_0)^T$ and $G_1 = (H_1)^T$ (i.e. the filter coefficients in the forward and reverse transforms are reverse of oneanother), and that

$$\lambda_{H_1}(\omega) = \overline{\lambda_{H_0}(\omega + \pi)}. \tag{6.29}$$

The perfect reconstruction condition for an alternative QMF filter bank can be written as

$$2 = \lambda_{H_0}(\omega)\lambda_{G_0}(\omega) + \lambda_{H_1}(\omega)\lambda_{G_1}(\omega) = \lambda_{H_0}(\omega)\overline{\lambda_{H_0}(\omega)} + \overline{\lambda_{H_0}(\omega+\pi)}\lambda_{H_0}(\omega+\pi)$$
$$= |\lambda_{H_0}(\omega)|^2 + |\lambda_{H_0}(\omega+\pi)|^2.$$

We see that the perfect reconstruction property of the two definitions of QMF filter banks only differ in that the latter take absolute values. It turns out that the latter also has many interesting solutions, as we will see in Chapter 7. If we in in condition (6.18) substitute $G_0 = (H_0)^T$ we get

$$\lambda_{H_1}(\omega) = \alpha^{-1}e^{-2id\omega}\lambda_{G_0}(\omega+\pi) = \alpha^{-1}e^{-2id\omega}\overline{\lambda_{H_0}(\omega+\pi)}.$$

If we set $\alpha = 1, d = 0$, we get equality here. A similar computation follows for Condition (6.19). In other words, also alternative QMF filter banks satisfy the alias cancellation condition. In the literature, a wavelet is called *orthonormal* if $G_0 = (H_0)^T, G_1 = (H_1)^T$. From our little computation it follows that alternative QMF filter banks with perfect reconstruction are examples of orthonormal wavelets, and correpond to orthonormal wavelets which satisfy $\alpha = 1, d = 0$.

For the Haar wavelet it is easily checked that $G_0 = (H_0)^T, G_1 = (H_1)^T$, but it does not satisfy the relation $\lambda_{H_1}(\omega) = \overline{\lambda_{H_0}(\omega+\pi)}$. Instead it satsifies the relation $\lambda_{H_1}(\omega) = -\overline{\lambda_{H_0}(\omega+\pi)}$. In other words, the Haar wavelet is not an alternative QMF filter bankthe way we have defined them. The difference lies only in a sign, however. This is the reason why the Haar wavelet is still listed as an alternative QMF filter bank in the literature. The additional sign leads to orthonormnal wavelets which satisfy $\alpha = -1, d = 0$ instead.

The following is clear for orthonormal wavelets.

Theorem 6.22. *Orthogonality og the DWT matrix.*

A DWT matrix is orthogonal (i.e. the IDWT equals the transpose of the DWT) if and only if the filters satisfy $G_0 = (H_0)^T, G_1 = (H_1)^T$, i.e. if and only if the MRA equals the dual MRA.

This can be proved simply by observing that, if we transpose the DWT-matrix, Theorem 6.25 says that we get an IDWT matrix with filters $(H_0)^T, (H_1)^T$, and this is equal to the IDWT if and only if $G_0 = (H_0)^T, G_1 = (H_1)^T$. It follows that QMF filter banks with perfect reconstruction give rise to orthonormal wavelets.

Exercise 6.13: Finding FIR filters

Show that it is impossible to find a non-trivial FIR-filter which satisfies Equation (6.28) in the compendium.

Exercise 6.14: The Haar wavelet as an alternative QMF filter bank

Show that the Haar wavelet satisfies $\lambda_{H_1}(\omega) = -\overline{\lambda_{H_0}(\omega + \pi)}$, and $G_0 = (H_0)^T$, $G_1 = (H_1)^T$. The Haar wavelet can thus be considered as an alternative QMF filter bank.

6.3 A generalization of the filter representation, and its use in audio coding

It turns out that the filter representation, which we now have used for an alternative representation of a wavelet transformation, can be generalized in such a way that it also is useful for audio coding. In this section we will first define this generalization. We will then state how the MP3 standard encodes and decodes audio, and see how our generalization is connected to this. Much literature fails to elaborate on this connection. We will call our generalizations *filter bank transforms*, or simply *filter banks*. Just as for wavelets, filters are applied differently for the forward and reverse transforms. The code for this section can be found in a module called `mp3funcs`.

We start by defining the forward filter bank transform and its filters.

Definition 6.23. *Forward filter bank transform.*

Let $H_0, H_1, \ldots, H_{M-1}$ be $N \times N$-filters. A *forward filter bank transform* H produces output $\boldsymbol{z} \in \mathbb{R}^N$ from the input $\boldsymbol{x} \in \mathbb{R}^N$ in the following way:

- $z_{iM} = (H_0\boldsymbol{x})_{iM}$ for any i so that $0 \leq iM < N$.
- $z_{iM+1} = (H_1\boldsymbol{x})_{iM+1}$ for any i so that $0 \leq iM + 1 < N$.
- ...
- $z_{iM+(M-1)} = (H_{M-1}\boldsymbol{x})_{iM+(M-1)}$ for any i so that $0 \leq iM + (M-1) < N$.

In other words, the output of a forward filter bank transform is computed by applying filters $H_0, H_1, \ldots, H_{M-1}$ to the input, and by downsampling and assembling these so that we obtain the same number of output samples as input samples (also in this more general setting this is called critical sampling). $H_0, H_1, \ldots, H_{M-1}$ are also called *analysis filter components*, the output of filter H_i is called channel i *channel*, and M is called the number of channels. The output samples z_{iM+k} are also called the *subband samples* of channel k.

Clearly this definition generalizes the DWT and its analysis filters, since these can be obtained by setting $M = 2$. The DWT is thus a 2-channel forward filter bank transform. While the DWT produces the output $\begin{pmatrix} \boldsymbol{c}_{m-1} \\ \boldsymbol{w}_{m-1} \end{pmatrix}$ from the input $\boldsymbol{c}_m$, an M-channel forward filter bank transform splits the output into M components, instead of 2. Clearly, in the matrix of a forward filter bank

transform the rows repeat cyclically with period M, similarly to MRA-matrices. In practice, the filters in a forward filter bank transform are chosen so that they concentrate on specific frequency ranges. This parallels what we saw for the filters of a wavelet, where one concentrated on high frequencies, one on low frequencies. Using a filter bank to split a signal into frequency components is also called *subband coding*. But the filters in a filter bank are usually not ideal bandpass filters. There exist a variety of different filter banks, for many different purposes [37, 30]. In Chapter 7 we will say more on how one can construct filter banks which can be used for subband coding.

Let us now turn to reverse filter bank transforms.

Definition 6.24. *Reverse filter bank transforms.*

Let $G_0, G_1, \ldots, G_{M-1}$ be $N \times N$-filters. An *reverse filter bank transform* G produces $\boldsymbol{x} \in \mathbb{R}^N$ from $\boldsymbol{z} \in \mathbb{R}^N$ in the following way:

- Define $\boldsymbol{z}_k\mathbb{R}^N$ as the vector where $(\boldsymbol{z}_k)_{iM+k} = \boldsymbol{z}_{iM+k}$ for all i so that $0 \leq iM + k < N$, and $(\boldsymbol{z}_k)_s = 0$ for all other s.

$$\boldsymbol{x} = G_0\boldsymbol{z}_0 + G_1\boldsymbol{z}_1 + \ldots + G_{M-1}\boldsymbol{z}_{M-1}. \tag{6.30}$$

$G_0, G_1, \ldots, G_{M-1}$ are also called *synthesis filter components*.

Again, this generalizes the IDWT and its synthesis filters, and the IDWT can be seen as a 2-channel reverse filter bank transform. Also, in the matrix of a reverse filter bank transform, the columns repeat cyclically with period M, similarly to MRA-matrices. Also in this more general setting the filters G_i are in general different from the filters H_i. But we will see that, just as we saw for the Haar wavelet, there are important special cases where the analysis and synthesis filters are equal, and where their frequency responses are simply shifts of oneanother. It is clear that definitions 6.23 and 6.24 give the diagram for computing forward and reverse filter bank transforms shown in Figure 6.7:

Here $\downarrow_M$ and $\uparrow_M$ means that we extract every M'th element in the vector, and add $M-1$ zeros between the elements, respectively, similarly to how we previously defined $\downarrow_2$ and $\uparrow_2$. Comparing Figure 6.3 with Figure 6.7 makes the similarities between wavelet transformations and the transformation used in the MP3 standard very visible: Although the filters used are different, they are subject to the same kind of processing, and can therefore be subject to the same implementations.

In general it may be that the synthesis filters do not invert exactly the analysis filters. If the synthesis system exactly inverts the analysis system, we say that we have a *perfect reconstruction filter bank*. Since the analysis system introduces undesired frequencies in the different channels, these have to cancel in the inverse transform, in order to reconstruct the input exactly.

We will have use for the following simple connection between forward and reverse filter bank transforms, which follows imemdiately from the definitions.

Theorem 6.25. *Connection between forward and reverse filter bank transforms.*

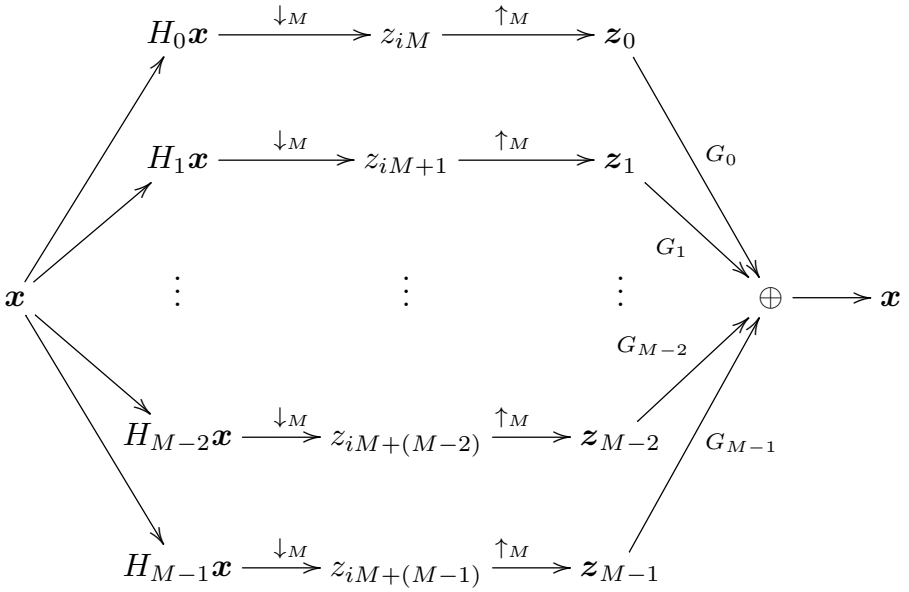

Figure 6.7: Illustration of forward and reverse filter bank transforms.

Assume that H is a forward filter bank transform with filters $H_0, \ldots, H_{N-1}$. Then H^T is a reverse filter bank transform with filters $G_0 = (H_0)^T, \ldots, G_{N-1} = (H_{N-1})^T$.

6.3.1 Forward filter bank transform used for encoding in the MP3 standard

Now, let us turn to the MP3 standard. The MP3 standard document states that it applies a filter bank, and explains the following procedure for applying this filter bank, see p. 67 of the standard document (the procedure is slightly modified with mathematical terminology adapted to this book):

- Input 32 audio samples at a time.
- Build an input sample vector $X \in \mathbb{R}^{512}$, where the 32 new samples are placed first, all other samples are delayed with 32 elements. In particular the 32 last samples are taken out.
- Multiply X componentwise with a vector C (this vector is defined through a table in the standard), to obtain a vector $Z \in \mathbb{R}^{512}$. The standard calls this *windowing*.
- Compute the vector $Y \in \mathbb{R}^{64}$ where $Y_i = \sum_{j=0}^{7} Z_{i+64j}$. The standard calls this a *partical calculation*.

- Calculate $S = MY \in \mathbb{R}^{32}$, where M is the 32×64- matrix where $M_{ik} = \cos((2i+1)(k-16)\pi/64)$. S is called the vector of output samples, or output subband samples. The standard calls this *matrixing*.

The standard does not motivate these steps, and does not put them into the filter bank transform framework which we have established. Also, the standard does not explain how the values in the vector C have been constructed.

Let us start by proving that the steps above really corresponds to applying a forward filter bank transform, and let us state the corresponding filters of this transform. The procedure computes 32 outputs in each iteration, and each of them is associated with a subband. Therefore, from the standard we would guess that we have $M = 32$ channels, and we would like to find the corresponding 32 filters $H_0, H_1, \ldots, H_{31}$.

It may seem strange to use the name matrixing here, for something which obviously is matrix multiplication. The reason for this name must be that the at the origin of the procedure come from outside a linear algebra framework. The name windowing is a bit strange, too. This really does not correspond to applying a window to the sound samples as we explained in Section 3.3.1. We will see that it rather corresponds to applying a filter coefficient to a sound sample. A third and final thing which seems a bit strange is that the order of the input samples is reversed, since we are used to having the first sound samples in time with lowest index. This is perhaps more usual to do in an engineering context, and not so usual in a mathematical context. FIFO.

Clearly, the procedure above defines a linear transformation, and we need to show that this linear transformation coincides with the procedure we defined for a forward filter bank transform, for a set of 32 filters. The input to the transformation are the audio samples, which we will denote by a vector $\boldsymbol{x}$. At iteration s of the procedure above the input audio samples are $x_{32s-512}, x_{32s-510}, \ldots, x_{32s-1}$, and $X_i = x_{32s-i-1}$ due to the reversal of the input samples. The output to the transformation at iteration s of the procedure are the $S_0, \ldots, S_{31}$. We assemble these into a vector $\boldsymbol{z}$, so that the output at iteration s are $z_{32(s-1)} = S_0$, $z_{32(s-1)+1} = S_1, \ldots, z_{32(s-1)+31} = S_{31}$.

We will have use for the following cosine-properties, which are easily verified:

$$\cos\left(2\pi(n+1/2)(k+2Nr)/(2N)\right) = (-1)^r \cos\left(2\pi(n+1/2)k/(2N)\right) \tag{6.31}$$

$$\cos\left(2\pi(n+1/2)(2N-k)/(2N)\right) = -\cos\left(2\pi(n+1/2)k/(2N)\right). \tag{6.32}$$

With the terminology above and using Property (6.31) the transformation can be written as

$$\begin{aligned}
z_{32(s-1)+n} &= \sum_{k=0}^{63} \cos((2n+1)(k-16)\pi/64) Y_k = \sum_{k=0}^{63} \cos((2n+1)(k-16)\pi/64) \sum_{j=0}^{7} Z_{k+64j} \\
&= \sum_{k=0}^{63} \sum_{j=0}^{7} (-1)^j \cos((2n+1)(k+64j-16)\pi/64) Z_{k+64j} \\
&= \sum_{k=0}^{63} \sum_{j=0}^{7} \cos((2n+1)(k+64j-16)\pi/64)(-1)^j C_{k+64j} X_{k+64j} \\
&= \sum_{k=0}^{63} \sum_{j=0}^{7} \cos((2n+1)(k+64j-16)\pi/64)(-1)^j C_{k+64j} x_{32s-(k+64j)-1}.
\end{aligned}$$

Now, if we define $\{h_r\}_{r=0}^{511}$ by $h_{k+64j} = (-1)^j C_{k+64j}$, $0 \leq j < 8, 0 \leq k < 64$, and $h^{(n)}$ as the filter with coefficients $\{\cos((2n+1)(k-16)\pi/64) h_k\}_{k=0}^{511}$, the above can be simplified as

$$\begin{aligned}
z_{32(s-1)+n} &= \sum_{k=0}^{511} \cos((2n+1)(k-16)\pi/64) h_k x_{32s-k-1} = \sum_{k=0}^{511} (h^{(n)})_k x_{32s-k-1} \\
&= (h^{(n)} \boldsymbol{x})_{32s-1} = (E_{n-31} h^{(n)} \boldsymbol{x})_{32(s-1)+n}.
\end{aligned}$$

This means that the output of the procedure stated in the MP3 standard can be computed as a forward filter bank transform, and that we can choose the analysis filters as $H_n = E_{n-31} h^{(n)}$.

Theorem 6.26. *Forward filter bank transform for the MP3 standard.*

Define $\{h_r\}_{r=0}^{511}$ by $h_{k+64j} = (-1)^j C_{k+64j}$, $0 \leq j < 8, 0 \leq k < 64$, and $h^{(n)}$ as the filter with coefficients $\{\cos((2n+1)(k-16)\pi/64) h_k\}_{k=0}^{511}$. If we define $H_n = E_{n-31} h^{(n)}$, the procedure stated in the MP3 standard corresponds to applying the corresponding forward filter bank transform.

The filters H_n were shown in Example 3.37 as examples of filters which concentrate on specific frequency ranges. The h_k are the filter coefficients of what is called a prototype filter. This kind of filter bank is also called a *cosine-modulated filter*. The multiplication with $\cos\left(2\pi(n+1/2)(k-16)/(2N)\right) h_k$, modulated the filter coefficients so that the new filter has a frequency response which is simply shifted in frequency in a symmetric manner: In Exercise 3.30, we saw that, by multiplying with a cosine, we could contruct new filters with real filter coefficients, which also corresponded to shifting a prototype filter in frequency. Of course, multiplication with a complex exponential would also shift the frequency response (such filter banks are called *DFT-modulated filter banks*), but the problem with this is that the new filter has complex coefficients: It will turn out that cosine-modulated filter banks can also be constructed so that they are invertible, and that one can find such filter banks where the inverse is easily found.

The effect of the delay in the definition of H_n is that, for each n, the multiplications with the vector $\boldsymbol{x}$ are "aligned", so that we can save a lot of multiplications by performing this multiplication first, and summing these. We actually save even more multiplications in the sum where j goes from 0 to 7, since we here multiply with the same cosines. The steps defined in the MP3 standard are clearly motivated by the desire to reduce the number of multiplications due to these facts. A simple arithmetic count illutrates these savings: For every 32 output samples, we have the following number of multiplications:

- The first step computes 512 multiplications.
- The second step computes 64 sums of 8 elements each, i.e. a total of $7 \times 64 = 448$ additions (note that $q = 512/64 = 8$).

The standard says nothing about how the matrix multiplication in the third step can be implemented. A direct multiplication would yield $32 \times 64 = 2048$ multiplications, leaving a total number of multiplications at 2560. In a direct implementation of the forward filter bank transform, the computation of 32 samples would need $32 \times 512 = 16384$ multiplications, so that the procedure sketched in the standard gives a big reduction.

The standard does not mention all possibilities for saving multiplications, however: We can reduce the number of multiplications even further, since clearly a DCT-type implementation can be used for the matrixing operation. We already have an efficient implementation for multiplication with a 32×32 type-III cosine matrix (this is simply the IDCT). We have seen that this implementation can be chosen to reduce the number of multiplications to $N \log_2 N/2 = 80$, so that the total number of multiplications is $512 + 80 = 592$. Clearly then, when we use the DCT, the first step is the computationally most intensive part.

6.3.2 Reverse filter bank transform used for decoding in the MP3 standard

Let us now turn to how decoding is specified in the MP3 standard, and see that we can associate this with a reverse filter bank transform. The MP3 standard also states the following procedure for decoding:

- Input 32 new subband samples as the vector S.
- Change vector $V \in \mathbb{R}^{512}$, so that all elements are delayed with 64 elements. In particular the 64 last elements are taken out.
- Set the first 64 elements of V as $NS \in \mathbb{R}^{64}$, where N is the 64×32-matrix where $N_{ik} = \cos((16 + i)(2k + 1)\pi/64)$. The standard also calls this matrixing.
- Build the vector $U \in \mathbb{R}^{512}$ from V from the formulas $U_{64i+j} = V_{128i+j}$, $U_{64i+32+j} = V_{128i+96+j}$ for $0 \leq i \leq 7$ and $0 \leq j \leq 31$, i.e. U is the vector where V is first split into segments of length 132, and U is constructed by assembling the first and last 32 elements of each of these segments.

- Multiply U componentwise with a vector D (this vector is defined in the standard), to obtain a vector $W \in \mathbb{R}^{512}$. The standard also calls this windowing.
- Compute the 32 next sound samples as $\sum_{i=0}^{15} W_{32i+j}$.

To interpret this also in terms of filters, rewrite first steps 4 to 6 as

$$\begin{aligned} x_{32(s-1)+j} &= \sum_{i=0}^{15} W_{32i+j} = \sum_{i=0}^{15} D_{32i+j} U_{32i+j} \\ &= \sum_{i=0}^{7} D_{64i+j} U_{64i+j} + \sum_{i=0}^{7} D_{64i+32+j} U_{64i+32+j} \\ &= \sum_{i=0}^{7} D_{64i+j} V_{128i+j} + \sum_{i=0}^{7} D_{64i+32+j} V_{128i+96+j}. \end{aligned} \tag{6.33}$$

The elements in V are obtained by "matrixing" different segments of the vector $\boldsymbol{z}$. More precisely, at iteration s we have that

$$\begin{pmatrix} V_{64r} \\ V_{64r+1} \\ \vdots \\ V_{64r+63} \end{pmatrix} = N \begin{pmatrix} z_{32(s-r-1)} \\ z_{32(s-r-1)+1} \\ \vdots \\ z_{32(s-r-1)+31} \end{pmatrix},$$

so that

$$V_{64r+j} = \sum_{k=0}^{31} \cos((16+j)(2k+1)\pi/64) z_{32(s-r-1)+k}$$

for $0 \leq j \leq 63$. Since also

$$V_{128i+j} = V_{64(2i)+j} \qquad\qquad V_{128i+96+j} = V_{64(2i+1)+j+32},$$

we can rewrite Equation (6.33) as

$$\begin{aligned} &\sum_{i=0}^{7} D_{64i+j} \sum_{k=0}^{31} \cos((16+j)(2k+1)\pi/64) z_{32(s-2i-1)+k} \\ &+ \sum_{i=0}^{7} D_{64i+32+j} \sum_{k=0}^{31} \cos((16+j+32)(2k+1)\pi/64) z_{32(s-2i-2))+k}. \end{aligned}$$

Again using Relation (6.31), this can be written as

$$\sum_{k=0}^{31}\sum_{i=0}^{7}(-1)^i D_{64i+j}\cos((16+64i+j)(2k+1)\pi/64)z_{32(s-2i-1)+k}$$
$$+\sum_{k=0}^{31}\sum_{i=0}^{7}(-1)^i D_{64i+32+j}\cos((16+64i+j+32)(2k+1)\pi/64)z_{32(s-2i-2)+k}.$$

Now, if we define $\{g_r\}_{r=0}^{511}$ by $g_{64i+s}=(-1)^i C_{64i+s}$, $0\le i<8, 0\le s<64$, and $g^{(k)}$ as the filter with coefficients $\{\cos((r+16)(2k+1)\pi/64)g_r\}_{r=0}^{511}$, the above can be simplified as

$$\begin{aligned}
&\sum_{k=0}^{31}\sum_{i=0}^{7}(g^{(k)})_{64i+j}z_{32(s-2i-1)+k}+\sum_{k=0}^{31}\sum_{i=0}^{7}(g^{(k)})_{64i+j+32}z_{32(s-2i-2)+k}\\
=&\sum_{k=0}^{31}\left(\sum_{i=0}^{7}(g^{(k)})_{32(2i)+j}z_{32(s-2i-1)+k}+\sum_{i=0}^{7}(g^{(k)})_{32(2i+1)+j}z_{32(s-2i-2)+k}\right)\\
=&\sum_{k=0}^{31}\sum_{r=0}^{15}(g^{(k)})_{32r+j}z_{32(s-r-1)+k},
\end{aligned}$$

where we observed that $2i$ and $2i+1$ together run through the values from 0 to 15 when i runs from 0 to 7. Since $\boldsymbol{z}$ has the same values as $\boldsymbol{z}_k$ on the indices $32(s-r-1)+k$, this can be written as

$$\begin{aligned}
&=\sum_{k=0}^{31}\sum_{r=0}^{15}(g^{(k)})_{32r+j}(\boldsymbol{z}_k)_{32(s-r-1)+k}\\
&=\sum_{k=0}^{31}(g^{(k)}\boldsymbol{z}_k)_{32(s-1)+j+k}=\sum_{k=0}^{31}((E_{-k}g^{(k)})\boldsymbol{z}_k)_{32(s-1)+j}.
\end{aligned}$$

By substituting a general s and j we see that $\boldsymbol{x}=\sum_{k=0}^{31}(E_{-k}g^{(k)})\boldsymbol{z}_k$. We have thus proved the following.

Theorem 6.27. *Reverse filter bank transform for the MP3 standard.*

Define $\{g_r\}_{r=0}^{511}$ by $g_{64i+s}=(-1)^i C_{64i+s}$, $0\le i<8, 0\le s<64$, and $g^{(k)}$ as the filter with coefficients $\{\cos((r+16)(2k+1)\pi/64)g_r\}_{r=0}^{511}$. If we define $G_k=E_{-k}g^{(k)}$, the procedure stated in the MP3 standard corresponds to applying the corresponding reverse filter bank transform.

In other words, both procedures for encoding and decoding stated in the MP3 standard both correspond to filter banks: A forward filter bank transform for the encoding, and a reverse filter bank transform for the decoding. Moreover, both filter banks can be constructed by cosine-modulating prototype filters, and the coefficients of these prototype filters are stated in the MP3 standard (up to

multiplication with an alternating sign). Note, however, that the two prototype filters may be different. When we compare the two tables for these coefficients in the standard they do indeed seem to be different. At closer inspection, however, one sees a connection: If you multiply the values in the D-table with 32, and reverse them, you get the values in the C-table. This indicates that the analysis and synthesis prototype filters are the same, up to multiplication with a scalar. This connection will be explained in Section 8.3.

While the steps defined in the MP3 standard for decoding seem a bit more complex than the steps for encoding, they are clearly also motivated by the desire to reduce the number of multiplications. In both cases (encoding and decoding), the window tables (C and D) are in direct connection with the filter coefficients of the prototype filter: one simply adds a sign which alternates for every 64 elements. The standard document does not mention this connection, and it is perhaps not so simple to find this connection in the literature (but see [26]).

The forward and reverse filter bank transforms are clearly very related. The following result clarifies this.

Theorem 6.28. *Connection between the forward and reverse filter bank transforms in the MP3 standard.*

Assume that a forward filter bank transform has filters on the form $H_i = E_{i-31}h^{(i)}$ for a prototype filter h. Then $G = E_{481}H^T$ is a reverse filter bank transform with filters on the form $G_k = E_{-k}g^{(k)}$, where g is a prototype filter where the elements equal the reverse of those in h. Vice versa, $H = E_{481}G^T$.

Proof. From Theorem 6.25 we know that H^T is a reverse filter bank transform with filters

$$(H_i)^T = (E_{i-31}h^{(i)})^T = E_{31-i}(h^{(i)})^T.$$

$(h^{(i)})^T$ has filter coefficients $\cos((2i+1)(-k-16)\pi/64))h_{-k}$. If we delay all $(H_i)^T$ with $481 = 512 - 31$ elements as in the theorem, we get a total delay of $512 - 31 + 31 - i = 512 - i$ elements, so that we get the filter

$$\begin{aligned}
&E_{512-i}\{\cos((2i+1)(-k-16)\pi/64))h_{-k}\}_k \\
&= E_{-i}\{\cos((2i+1)(-(k-512)-16)\pi/64))h_{-(k-512)}\}_k \\
&= E_{-i}\{\cos((2i+1)(k+16)\pi/64))h_{-(k-512)}\}_k.
\end{aligned}$$

Now, we define the prototype filter g with elements $g_k = h_{-(k-512)}$. This has, just as h, its support on $[1, 511]$, and consists of the elements from h in reverse order. If we define $g^{(i)}$ as the filter with coefficients $\cos((2i+1)(k+16)\pi/64))g_k$, we see that $E_{481}H^T$ is a reverse filter bank transform with filters $E_{-i}g^{(i)}$. Since $g^{(k)}$ now has been defined as for the MP3 standard, and its elements are the reverse of those in h, the result follows. □

We will have use for this result in Section 8.3, when we find conditions on the prototype filter in order for the reverse transform to invert the forward

transform. Preferably, the reverse filter bank transform inverts exactly the forward filter bank transform. In Exercise 6.16 we construct examples which show that this is not the case. In the same exercise we also find many examples where the reverse transform does what we would expect. These examples will also be explained in Section 8.3, where we also will see how one can get around this so that we obtain a system with perfect reconstruction. It may seem strange that the MP3 standard does not do this.

In the MP3 standard, the output from the forward filter bank transform is processed further, before the result is compressed using a lossless compression method.

Exercise 6.15: Plotting frequency responses

The values C_q, D_q can be found by calling the functions `mp3ctable`, `mp3dtable` which can be found on the book's webpage.

a) Use your computer to verify the connection we stated between the tables C and D, i.e. that $D_i = 32C_i$ for all i.

b) Plot the frequency responses of the corresponding prototype filters, and verify that they both are lowpass filters. Use the connection from Theorem (6.26) to find the prototype filter coefficients from the C_q.

Exercise 6.16: Implementing forward and reverse filter bank transforms

It is not too difficult to make implementations of the forward and reverse steps as explained in the MP3 standard. In this exercise we will experiment with this. In your code you can for simplicity assume that the input and output vectors to your methods all have lengths which are multiples of 32. Also, use the functions `mp3ctable`, `mp3dtable` mentioned in the previous exercise.

a) Write a function `mp3forwardfbt` which implements the steps in the forward direction of the MP3 standard.

b) Write also a function `mp3reversefbt` which implements the steps in the reverse direction.

6.4 Summary

We started this chapter by noting that, by reordering the target base of the DWT, the change of coordinate matrix took a particular form. From this form we understood that the DWT could be realized in terms of two filters H_0 and H_1, and that the IDWT could be realized in a similar way in terms of two filters G_0 and G_1. This gave rise to what we called the filter representation of wavelets. The filter representation gives an entirely different view on wavelets: instead of constructing function spaces with certain properties and deducing corresponding

filters from these, we can instead construct filters with certain properties (such as alias cancellation and perfect reconstruction), and attempt to construct corresponding mother wavelets, scaling functions, and function spaces. This strategy, which replaces problems from function theory with discrete problems, will be the subject of the next chapter. In practice this is what is done.

We stated what is required for filter bank matrices to invert each other: The frequency responses of the lowpass filters needed to satisfy a certain equation, and once this is satsified the highpass filters can easily be obtained in the same way we previously obtained highpass filters from lowpass filters. We will return to this equation in the next chapter.

A useful consequence of the filter representation was that we could reuse existing implementations of filters to implement the DWT and the IDWT, and reuse existing theory, such as symmetric extensions. For wavelets, symmetric extensions are applied in a slightly different way, when compared to the developments which lead to the DCT. We looked at the frequency responses of the filters for the wavelets we have encountered upto now. From these we saw that G_0, H_0 were lowpass filters, and that G_1, H_1 were highpass filters, and we argued why this is typically the case for other wavelets as well. The filter reprersentation was also easily generalized from 2 to $M > 2$ filters, and such transformations had a similar interpretation in terms of splitting the input into a uniform set of frequencies. Such transforms were generally called filter bank transforms, and we saw that the processing performed by the MP3 standard could be interpreted as a certain filter bank transform, called a cosine-modulated filter bank. This is just one of many possible filter banks. In fact, the filter bank of the MP3 standard is largely outdated, since it is too simple, and as we will see it does not even give perfect reconstruction (only alias cancellation and no phase distortion). It is merely chosen here since it is the simplest to present theoretically, and since it is perhaps the best known standard for compression of sound. Other filters banks with better properties have been constructed, and they are used in more recent standards. In many of these filter banks, the filters do not partition frequencies uniformly, and have been adapted to the way the human auditory system handles the different frequencies. Different contruction methods are used to construct such filter banks. The motivation behind filter bank transforms is that their output is more suitable for further processing, such as compression, or playback in an audio system, and that they have efficient implementations.

We mentioned that the MP3 standard does not say how the prototype filters were chosen. We will have more to say on what dictates their choice in Section 8.3.

There are several differences between the use of wavelet transformations in wavelet theory, and the use of filter bank transforms in signal processing theory One is that wavelet transforms are typically applied in stages, while filter bank transforms often are not. Nevertheless, such use of filter banks also has theoretical importance, and this gives rise to what is called *tree-structured filter banks* [37]. Another difference lies in the use of the term perfect reconstruction system. In wavelet theory this is a direct consequence of the wavelet construction, since the DWT and the IDWT correspond to change of coordinates to and from the same bases. The alternative QMF filter bank was used as an example

of a filter bank which stems from signal processing, and which also shows up in wavelet transformation. In signal processing theory, one has a wider perspective, since one can design many useful systems with fast implementations when one replaces the perfect reconstruction requirement with a near perfect reconstruction requirement. One instead requires that the reverse transform gives alias cancellation. The classical QMF filter banks were an example of this. The original definition of classical QMF filter banks are from [7], and differ only in a sign from how they are defined here.

All filters we encounter in wavelets and filter banks in this book are FIR. This is just done to limit the exposition. Much useful theory has been developed using IIR-filters.

Chapter 7

Constructing interesting wavelets

In the previous chapter, from an MRA with corresponding scaling function and mother wavelet, we defined what we called a forward filter bank transform. We also defined a reverse filter bank transform, but we did not state an MRA connected to this, or prove if any such association could be made. In this chapter we will address this. We will also see, if we start with a forward and reverse filter bank transform, how we can construct corresponding MRA's, and for which transforms we can make this construction. We will see that there is a great deal of flexibility in the filter bank transforms we can construct (as this is a discrete problem). Actually it is so flexible that we can construct scaling functions/mother wavelets with any degree of regularity, and well suited for approximation of functions. This will also explain our previous interest in vanishing moments, and explain how we can find the simplest filters which give rise to a given number of vanishing moments, or a given degree of differentiability.. Answers to these questions certainly transfer much more theory between wavelets and filters. Several of these filters enjoy a widespread use in applications. We will look at two of these. These are used for lossless and lossy compression in JPEG2000, which is a much used standard. These wavelets all have symmetric filters. We end the chapter by looking at a family of orthonormal wavelets with different number of vanishing moments.

7.1 From filters to scaling functions and mother wavelets

Example 7.1. *The alternative piecewise linear wavelet.*

Let us return to the alternative piecewise linear wavelet. In Example 6.19 we found the filters H_0, H_1 for this wavelet, and these determine the dual scaling function and the dual mother wavelet. We already know how the scaling function

and the mother wavelet look, but how do the dual functions look? It turns out that there is usually no way to find analytical expressions for these dual functions (as is the case for the scaling function and the mother wavelet itself in most cases), but that there still is an algorithm we can apply in order to see how these functions look. This algorithm is called the *cascade algorithm*, and works essentially by computing the coordinates of ϕ, ψ (or $\tilde{\phi}, \tilde{\psi}$) in $\boldsymbol{\phi}_m$ (or $\tilde{\boldsymbol{\phi}}_m$). By increasing m, we have previously argued that these coordinates are good approximations to the samples of the functions.

To be more specific, we start with the following observation for the dual functions (similar observations hold for the scaling function and the mother wavelet also):

- the coordinates of $\tilde{\phi}$ in $(\tilde{\boldsymbol{\phi}}_0, \tilde{\boldsymbol{\psi}}_0, \tilde{\boldsymbol{\psi}}_1 \ldots)$ is the vector with 1 first, followed by only zeros,
- the coordinates of $\tilde{\psi}$ in $(\tilde{\boldsymbol{\phi}}_0, \tilde{\boldsymbol{\psi}}_0, \tilde{\boldsymbol{\psi}}_1 \ldots)$ is the vector with N zeros first, then a 1, and then only zeros.

The length of these vectors is $N2^m$. The coordinates in $\tilde{\boldsymbol{\phi}}_m$ for $\tilde{\phi}$ and $\tilde{\psi}$ can be obtained by applying the m-level IDWT for the dual wavelet (i.e. the filters $(H_0)^T$, $(H_1)^T$ are used) to these vectors. In Exercise 7.1 we will study code which uses this approach to approximate the scaling function and mother wavelet. In Figure 7.1 we have plotted the resulting coordinates in $\tilde{\boldsymbol{\phi}}_{10}$, and thus a good approximation to $\tilde{\phi}$ and $\tilde{\psi}$. We see that these functions look very irregular. Also, they are very different from the original scaling function and mother wavelet. We will later argue that this is bad, it would be much better if $\phi \approx \tilde{\phi}$ and $\psi \approx \tilde{\psi}$.

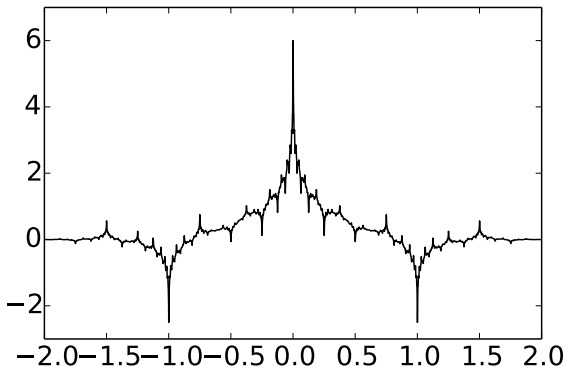

Figure 7.1: Dual scaling function $\tilde{\phi}$ (left) and dual mother wavelet $\tilde{\psi}$ (right) for the alternative piecewise linear wavelet.

From Theorem 6.11 it follows that the support sizes of these dual functions are are 4 and 3, respectively, so that their supports should be $[-2, 2]$ and $[-1, 2]$,

respectively. This is the reason why we have plotted the functions over $[-2, 2]$. The plots seem to confirm the support sizes we have computed.

Let us formalize the cascade algorithm from the previous example as follows.

Definition 7.2. *The cascade algorithm.*

The *cascade algorithm* applies a change of coordinates for the functions $\tilde{\phi}, \tilde{\psi}$ from $(\tilde{\boldsymbol{\phi}}_0, \tilde{\boldsymbol{\psi}}_0, \tilde{\boldsymbol{\psi}}_1 \ldots)$ to $\tilde{\boldsymbol{\phi}}_m$, and uses the new coordinates as an an approximation to the function values of these functions.

7.2 Turning things around: How to construct useful wavelet bases from filters

In our first examples of wavelets in Chapter 5, we started with some bases og functions $\boldsymbol{\phi}_m$, and deduced filters G_0 and G_1 from these. If we instead start with the filters G_0 and G_1, what properties must they fulfill in order for us to make an association the opposite way? We should thus demand that there exist functions ϕ, ψ so that

$$\phi(t) = \sum_{n=0}^{2N-1} (G_0)_{n,0} \phi_{1,n}(t) \tag{7.1}$$

$$\psi(t) = \sum_{n=0}^{2N-1} (G_1)_{n,1} \phi_{1,n}(t) \tag{7.2}$$

Using Equation (7.1), the Fourier transform of ϕ is

$$\begin{aligned}
\hat{\phi}(\omega) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(t) e^{-i\omega t} dt = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \left(\sum_n (G_0)_{n,0} \sqrt{2} \phi(2t - n) \right) e^{-i\omega t} dt \\
&= \frac{1}{\sqrt{2}\sqrt{2\pi}} \sum_n \int_{-\infty}^{\infty} (G_0)_{n,0} \phi(t) e^{-i\omega(t+n)/2} dt \\
&= \frac{1}{\sqrt{2}} \left(\sum_n (G_0)_{n,0} e^{-i\omega n/2} \right) \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(t) e^{-i(\omega/2)t} dt = \frac{\lambda_{G_0}(\omega/2)}{\sqrt{2}} \hat{\phi}(\omega/2).
\end{aligned} \tag{7.3}$$

Clearly this expression can be continued recursively. We can thus state the following result.

Theorem 7.3. g_N.

Define

$$g_N(\omega) = \prod_{s=1}^{N} \frac{\lambda_{G_0}(\omega/2^s)}{\sqrt{2}} \chi_{[0,2\pi]}(2^{-N}\omega). \tag{7.4}$$

Then on $[0, 2\pi 2^N]$ we have that $\hat{\phi}(\nu) = g_N(\nu)\hat{\phi}(\nu/2^N)$.

We can now prove the following.

Lemma 7.4. *$g_N(\nu)$ converges.*

Assume that $\sum_n (G_0)_n = \sqrt{2}$ (i.e. $\lambda_{G_0}(0) = \sqrt{2}$), and that G_0 is a FIR-filter. Then $g_N(\nu)$ converges pointwise as $N \to \infty$ to an infinitely differentiable function.

Proof. We need to verify that the infinite product $\prod_{s=1}^{\infty} \frac{\lambda_{G_0}(2\pi\nu/2^s)}{\sqrt{2}}$ converges. Taking logarithms we get $\sum_s \ln\left(\frac{\lambda_{G_0}(2\pi\nu/2^s)}{\sqrt{2}}\right)$. To see if this series converges, we consider the ratio between two successive terms:

$$\frac{\ln\left(\frac{\lambda_{G_0}(2\pi\nu/2^{s+1})}{\sqrt{2}}\right)}{\ln\left(\frac{\lambda_{G_0}(2\pi\nu/2^s)}{\sqrt{2}}\right)}.$$

Since $\sum_n (G_0)_n = \sqrt{2}$, we see that $\lambda_{G_0}(0) = \sqrt{2}$. Since $\lim_{\nu\to 0} \lambda_{G_0}(\nu) = \sqrt{2}$, both the numerator and the denominator above tends to 0 (to one inside the logarithms), so that we can use L'hospital's rule on $\frac{\ln\left(\frac{\lambda_{G_0}(\nu/2)}{\sqrt{2}}\right)}{\ln\left(\frac{\lambda_{G_0}(\nu)}{\sqrt{2}}\right)}$ to obtain

$$\frac{\lambda_{G_0}(\nu)}{\lambda_{G_0}(\nu/2)} \frac{\sum_n (G_0)_n (-in) e^{-in\nu/2}/2}{\sum_n (G_0)_n (-in) e^{-in\nu}} \to \frac{1}{2} < 1$$

as $\nu \to 0$. It follows that the product converges for any ν. Clearly the convergence is absolute and uniform on compact sets, so that the limit is infinitely differentiable. □

It follows that $\hat{\phi}$, when ϕ exists, must be an infinitely differentiable function also. Similarly we get

$$\begin{aligned}\hat{\psi}(\omega) &= \frac{1}{\sqrt{2}} \left(\sum_n (G_1)_{n-1,0} e^{-i\omega n/2} \right) \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(t) e^{-i(\omega/2)t} dt \\ &= \frac{1}{\sqrt{2}} \left(\sum_n (G_1)_{n,0} e^{-i\omega(n+1)/2} \right) \hat{\phi}(\omega/2) = e^{-i\omega/2} \frac{\lambda_{G_1}(\omega/2)}{\sqrt{2}} \hat{\phi}(\omega/2).\end{aligned}$$

It follows in the same way that $\hat{\psi}$ must be an infinitely differentiable function also.

Now consider the dual filter bank transform,as defined in Chapter 6. Its synthesis filter are $(H_0)^T$ and $(H_1)^T$. If there exist a scaling function $\tilde{\phi}$ and a mother wavelet $\tilde{\psi}$ for the dual transform, they must in the same way be infinitely differentiable. Moreover, $\hat{\phi}, \hat{\psi}, \hat{\tilde{\phi}}, \hat{\tilde{\psi}}$ can be found as infinite products of the known frequency responses. If these functions are in $L^2\mathbb{R}$, then we can find unique functions $\phi, \psi, \tilde{\phi}, \tilde{\psi}$ with these as Fourier transforms.

So, our goal is to find filters so that the derived infinite products of the frequency responses lie in $L^2(\mathbb{R})$, and so that the constructed functions $\phi, \psi, \tilde{\phi}, \tilde{\psi}$ give rise to "nice" wavelet bases. Some more technical requirements will be needed in order for this. In order to state these we should be clear on what we mean by a "nice" basis in this context. First of all, the bases should together span all of $L^2(\mathbb{R})$. But our bases are not orthogonal, so we should have some substitute for this. We will need the following definitions.

Definition 7.5. *Frame.*

Let $\mathcal{H}$ be a Hilbert space. A set of vectors $\{u_n\}_n$ is called a frame of $\mathcal{H}$ if there exist constants $A > 0$ and $B > 0$ so that, for any $f \in \mathcal{H}$,

$$A\|f\|^2 \leq \sum_n |\langle f, u_n\rangle|^2 \leq B\|f\|^2.$$

If $A = B$, the frame is said to be tight.

Note that, for a frame of $\mathcal{H}$, any $f \in \mathcal{H}$ is uniquely characterized by the inner products $\langle f, u_n\rangle$. Indeed, if both $a, b \in \mathcal{H}$ have the same inner products, then $a - b \in \mathcal{H}$ have inner products 0, which implies that $a = b$ from the left inequality.

For every frame one can find a dual frame $\{\tilde{u_n}\}_n$ which satisfies

$$\frac{1}{B}\|f\|^2 \leq \sum_n |\langle f, \tilde{u_n}\rangle|^2 \leq \frac{1}{A}\|f\|^2,$$

and

$$f = \sum_n \langle f, u_n\rangle \tilde{u_n} = \sum_n \langle f, \tilde{u_n}\rangle u_n. \tag{7.5}$$

Thus, if the frame is tight, the dual frame is also tight.

A frame is called a Riesz basis if all its vectors also are linearly independent. One can show that the vectors in the dual frame of a Riesz basis also are linearly independent, so that the dual frame of a Riesz basis also is a Riesz basis. It is also called the dual Riesz basis. We will also need the following definition.

Definition 7.6. *Biorthogonal bases.*

We say that two bases $\{\boldsymbol{f}_n\}_n$, $\{\boldsymbol{g}_m\}_m$ are *biorthogonal* if $\langle \boldsymbol{f}_n, \boldsymbol{g}_m\rangle = 0$ whenever $n \neq m$, and 1 if $n = m$.

From Equation (7.5) and linear independence, it is clear that the vectors in a Riesz basis and in its dual Riesz basis are biorthogonal. In the absence of orthonormal bases for $L^2(\mathbb{R})$, the best we can hope for is dual Riesz bases for $L^2(\mathbb{R})$. The following result explains how we can obtain this from the filters.

Proposition 7.7. *Biorthogonality.*

Assume that the frequency responses λ_{G_0} and λ_{H_0} can be written as.

$$\frac{\lambda_{G_0}(\omega)}{\sqrt{2}} = \left(\frac{1+e^{-i\omega}}{2}\right)^L \mathcal{F}(\omega) \qquad \frac{\lambda_{H_0}(\omega)}{\sqrt{2}} = \left(\frac{1+e^{-i\omega}}{2}\right)^{\tilde{L}} \tilde{\mathcal{F}}(\omega), \tag{7.6}$$

where $\mathcal{F}$ and $\tilde{\mathcal{F}}$ are trigonometric polynomials of finite degree. Assume also that, for some $k, \tilde{k} > 0$,

$$B_k = \max_\omega \left|\mathcal{F}(\omega)\cdots\mathcal{F}(2^{k-1}\omega)\right|^{1/k} < 2^{L-1/2} \tag{7.7}$$

$$\tilde{B_k} = \max_\omega \left|\tilde{\mathcal{F}}(\omega)\cdots\tilde{\mathcal{F}}(2^{\tilde{k}-1}\omega)\right|^{1/\tilde{k}} < 2^{\tilde{L}-1/2} \tag{7.8}$$

Then the following hold:

- $\phi, \tilde{\phi} \in L^2(\mathbb{R})$, and the corresponding bases $\boldsymbol{\phi}_0$ and $\tilde{\boldsymbol{\phi}}_0$ are biorthogonal.
- $\boldsymbol{\psi}_{m,n}$ is a Riesz basis of $L^2(\mathbb{R})$.
- $\tilde{\boldsymbol{\psi}}_{m,n}$ is the dual Riesz basis of $\boldsymbol{\psi}_{m,n}$. Thus, $\boldsymbol{\psi}_m, n$ and $\tilde{\boldsymbol{\psi}}_{m,n}$ are biorthogonal bases, and for any $f \in L^2(\mathbb{R})$,

$$f = \sum_{m,n} \langle f, \tilde{\psi}_{m,n}\rangle \psi_{m,n} = \sum_{m,n} \langle f, \psi_{m,n}\rangle \tilde{\psi}_{m,n}. \tag{7.9}$$

If also

$$B_k < 2^{L-1-m} \qquad \tilde{B_k} < 2^{\tilde{L}-1-\tilde{m}}, \tag{7.10}$$

then

- ϕ, ψ are m times differentiable and $\tilde{\psi}$ has $m+1$ vanishing moments,
- $\tilde{\phi}, \tilde{\psi}$ are $\tilde{m}$ times differentiable and ψ has $\tilde{m}+1$ vanishing moments.

The proof for Proposition 7.7 is long, technical, and split in many stages. The entire proof can be found in [5], and we will not go through all of it, only address some simple parts of it in the following subsections. After that we will see how we can find G_0, H_0 so that equations (7.6), (7.7), (7.8) are fulfilled. Before we continue on this path, several comments are in order.

1. The paper [5] much more general conditions for when filters give rise to a Riesz basis as stated here. The conditions (7.7), (7.8) are simply chosen because they apply or the filters we consider.

2. From Equation (7.6) it follows that the flatness in the frequency responses close to π explains how good the bases are for approximations, since the number of vanishing moments is infered from the multiplictity of the zero at π for the frequency response.

3. From the result we obtain an MRA (with scaling function ϕ), and a dual MRA (with scaling function $\tilde{\phi}$), as well as mother wavelets (ψ and $\tilde{\psi}$), and we can define the resolution spaces V_m and the detail spaces W_m as before, as well as the "dual resolution spaces" $\tilde{V}_m$, (the spaces spanned by $\tilde{\boldsymbol{\phi}}_m = \{\tilde{\phi}_{m,n}\}_n$) and "dual detail spaces" $\tilde{W}_m$ (the spaces spanned by $\tilde{\boldsymbol{\psi}}_m = \{\tilde{\psi}_{m,n}\}_n$). In general V_m is different from $\tilde{V}_m$ (except when $\phi = \tilde{\phi}$), and W_m is in general different from the orthogonal complement of V_{m-1} in V_m (except when $\phi = \tilde{\phi}$, when all bases are orthonormal), although constructed so that $V_m = V_{m-1} \oplus W_{m-1}$. Our construction thus involves two MRA's

$$V_0 \subset V_1 \subset V_2 \subset \cdots \subset V_m \subset \cdots \qquad \tilde{V}_0 \subset \tilde{V}_1 \subset \tilde{V}_2 \subset \cdots \subset \tilde{V}_m \subset \cdots$$

where there are different scaling functions, satisfying a biorthogonality relationship. This is also called a dual multiresolution analysis.

4. The DWT and IDWT are defined as before, so that the same change of coordinates can be applied, as dictated by the filter coefficients. As will be seen below, while proving Proposition 7.7 it also follows that the bases $\boldsymbol{\phi}_0 \oplus \boldsymbol{\psi}_0 \oplus \boldsymbol{\psi}_1 \cdots \boldsymbol{\psi}_{m-1}$ and $\tilde{\boldsymbol{\phi}}_0 \oplus \tilde{\boldsymbol{\psi}}_0 \oplus \tilde{\boldsymbol{\psi}}_1 \cdots \tilde{\boldsymbol{\psi}}_{m-1}$ are biorthogonal (in addition to that $\boldsymbol{\phi}_m$ and $\tilde{\phi}_m$ are biorthogonal, as stated). For $f \in V_m$ this means that

$$f(t) = \sum_n \langle f(t), \tilde{\phi}_{m,n} \rangle \phi_{m,n} = \sum_n \langle f(t), \tilde{\phi}_{0,n} \rangle \phi_{0,n} + \sum_{m' < m,n} \langle f(t), \tilde{\psi}_{m',n} \rangle \psi_{m',n},$$

since this relationship is fulfilled for any linear combination of the $\{\phi_{m,n}\}_n$, or for any of the $\{\phi_{0,n}, \psi_{m',n}\}_{m'<m,n}$, due to biorthogonality. Similarly, for $\tilde{f} \in \tilde{V}_m$

$$\tilde{f}(t) = \sum_n \langle \tilde{f}(t), \phi_{m,n} \rangle \tilde{\phi}_{m,n} = \sum_n \langle \tilde{f}(t), \phi_{0,n} \rangle \tilde{\phi}_{0,n} + \sum_{m' < m,n} \langle \tilde{f}(t), \psi_{m',n} \rangle \tilde{\psi}_{m',n}.$$

It follows that for $f \in V_m$ and for $\tilde{f} \in \tilde{V}_m$ the DWT and the IDWT and their duals can be expressed in terms of inner products as follows.

- The input to the DWT is $c_{m,n} = \langle f, \tilde{\phi}_{m,n} \rangle$. The output of the DWT is $c_{0,n} = \langle f, \tilde{\phi}_{0,n} \rangle$ and $w_{m',n} = \langle f, \tilde{\psi}_{m',n} \rangle$
- The input to the dual DWT is $\tilde{c}_{m,n} = \langle \tilde{f}, \phi_{m,n} \rangle$. The output of the dual DWT is $\tilde{c}_{0,n} = \langle \tilde{f}, \phi_{0,n} \rangle$ and $\tilde{w}_{m',n} = \langle \tilde{f}, \psi_{m',n} \rangle$.
- in the DWT matrix, column k has entries $\langle \phi_{1,k}, \tilde{\phi}_{0,l} \rangle$, and $\langle \phi_{1,k}, \tilde{\psi}_{0,l} \rangle$ (with a similar expression for the dual DWT).
- in the IDWT matrix, column $2k$ has entries $\langle \phi_{0,k}, \tilde{\phi}_{1,l} \rangle$, and column $2k+1$ has entries $\langle \psi_{0,k}, \tilde{\phi}_{1,l} \rangle$ (with a similar expression for the dual IDWT).

Equation (7.9) comes from eliminating the $\phi_{m,n}$ by letting $m \to \infty$.

5. When $\phi = \tilde{\phi}$ (orthonormal MRA's), the approximations (finite sums) above coincide with projections onto the spaces $V_m, \tilde{V}_m, W_m, \tilde{W}_m$. When $\phi \neq \tilde{\phi}$, however, there are no reasons to believe that these approximations equal the best approximations to f from V_m. In this case we have no procedure for computing best approximations. When f is not in $V_m, \tilde{V}_m$ we can, however, consider the approximations

$$\sum_n \langle f(t), \tilde{\phi}_{m,n}\rangle \phi_{m,n}(t) \in V_m \text{ and } \sum_n \langle f(t), \phi_{m,n}\rangle \tilde{\phi}_{m,n}(t) \in \tilde{V}_m$$

(when the MRA is orthonormal, this coincides with the best approximation). Now, we can choose m so large that $f(t) = \sum_n c_n \phi_{m,n}(t) + \epsilon(t)$, with $\epsilon(t)$ a small function. The first approximation can now be written

$$\begin{aligned}\sum_n \langle \sum_{n'} c_{n'}\phi_{m,n'}(t) + \epsilon(t), \tilde{\phi}_{m,n}\rangle \phi_{m,n}(t) &= \sum_n c_n \phi_{m,n}(t) + \sum_n \langle \epsilon(t), \tilde{\phi}_{m,n}\rangle \phi_{m,n}(t) \\ &= f(t) + \sum_n \langle \epsilon(t), \tilde{\phi}_{m,n}\rangle \phi_{m,n}(t) - \epsilon(t).\end{aligned}$$

Clearly, the difference $\sum_n \langle \epsilon(t), \tilde{\phi}_{m,n}\rangle \phi_{m,n}(t) - \epsilon(t)$ from f is small. It may. however, be hard to compute the c_n above, so that instead, as in Theorem 5.40, one uses $\frac{2^{-m}}{\int_0^N \phi_{m,0}(t)dt} f(n/2^m)\phi_{m,n}(t)$ as an approximation to f (i.e. use sample values as c_n) also in this more general setting.

6. Previously we were taught to think in a periodic or folded way, so that we could restrict to an interval $[0, N]$, and to bases of finite dimensions ($\{\phi_{0,n}\}_{n=0}^{N-1}$). But the results above are only stated for wavelet bases of infinite dimension. Let us therefore say something on how the results carry over to our finite dimensional setting. If $f \in L^2(\mathbb{R})$ we can define the function

$$f^{per}(t) = \sum_k f(t + kN) \qquad f^{fold}(t) = \sum_k f(t + 2kN) + \sum_k f(2kN - t).$$

f^{per} and f^{fold} are seen to be periodic with periods N and $2N$. It is easy to see that the restriction of f^{per} to $[0, N]$ is in $L^2([0, N])$, and that the restriction of f^{fold} to $[0, 2N]$ is in $L^2([0, 2N])$. In [4] it is shown that the result above extends to a similar result for the periodized/folded basis (i.e. $\psi^{fold}_{m,n}$), so that we obtain dual Riesz bases for $L^2([0, N])$ and $L^2([0, 2N])$ instead of $L^2(\mathbb{R})$. The result on the vanishing moments does not extend, however. One can, however, alter some of the basis functions so that one achieves this. This simply changes some of the columns in the DWT/IDWT matrices. Note that our extension strategy is not optimal. The extension is usually not differentiable at the boundary, so that the corresponding wavelet coefficients may be large, even though the wavelet has many vanishing moments. The only way to get around this would be to find an extension strategy which gave a more regular extension. However, natural images may not have high regularity, which would make such an extension strategy useless.

7.2.1 Sketch of proof for the biorthogonality in Proposition 7.7 (1)

We first show that $\boldsymbol{\phi}_0$ and $\tilde{\boldsymbol{\phi}}_0$ are biorthogonal. Recall that definition (7.4) said that $g_N(\omega) = \prod_{s=1}^{N} \frac{\lambda_{G_0}(\omega/2^s)}{\sqrt{2}} \chi_{[0,2\pi]}(2^{-N}\omega)$. Let us similarly define $h_N(\omega) = \prod_{s=1}^{N} \frac{\lambda_{H_0}(\omega/2^s)}{\sqrt{2}} \chi_{[0,2\pi]}(2^{-N}\omega)$. Recall that $g_N \to \hat{\phi}$ and $h_N \to \hat{\tilde{\phi}}$ pointwise as $N \to \infty$. We have that

$$g_{N+1}(\omega) = \frac{\lambda_{G_0}(\omega/2)}{\sqrt{2}} g_N(\omega/2) \qquad h_{N+1}(\omega) = \frac{\lambda_{H_0}(\omega/2)}{\sqrt{2}} h_N(\omega/2).$$

g_N, h_N are compactly supported, and equal to trigonometric polynomials on their support, so that $g_N, h_N \in L^2(\mathbb{R})$. Since the Fourier transform also is an isomorphism og $L^2(\mathbb{R})$ onto itself, there exist functions $u_N, v_N \in L^2(\mathbb{R})$ so that $g_N = \hat{u_N}$, $h_N = \hat{v_N}$. Since the above relationship equals that of Equation (7.3), with $\hat{\phi}$ replaced with g_N, we must have that

$$u_{N+1}(t) = \sum_n (G_0)_{n,0} \sqrt{2} u_N(2t-n) \quad v_{N+1}(t) = \sum_n (H_0)_{0,n} \sqrt{2} v_N(2t-n).$$

Now, note that $g_0(\omega) = h_0(\omega) = \chi_{[0,1]}(\omega)$. Since $\langle u_0, v_0 \rangle = \langle g_0, h_0 \rangle$ we get that

$$\int_{\infty}^{-\infty} u_0(t)\overline{v_0(t-k)}dt = \int_{\infty}^{-\infty} g_0(\nu)\overline{h_0(\nu)e^{2\pi i k\nu}}d\nu = \int_0^{2\pi} e^{-2\pi i k\nu} d\nu = \delta_{k,0}.$$

Now assume that we have proved that $\langle u_N(t), v_N(t-k) \rangle = \delta_{k,0}$. We then get that

$$\begin{aligned}
\langle u_{N+1}(t), v_{N+1}(t-k) \rangle &= 2 \sum_{n_1,n_2} (G_0)_{n_1,0}(H_0)_{0,n_2} \langle u_N(2t-n_1), v_N(2(t-k)-n_2) \rangle \\
&= 2 \sum_{n_1,n_2} (G_0)_{n_1,0}(H_0)_{0,n_2} \langle u_N(t), v_N(t+n_1-n_2-2k) \rangle \\
&= \sum_{n_1,n_2 | n_1-n_2=2k} (G_0)_{n_1,0}(H_0)_{0,n_2} = \sum_n (H_0)_{0,n-2k}(G_0)_{n,0} \\
&= \sum_n (H_0)_{2k,n}(G_0)_{n,0} = \sum_n H_{2k,n}G_{n,0} = (HG)_{2k,0} = I_{2k,0} = \delta_{k,0}
\end{aligned}$$

where we did the change of variables $u = 2t - n_1$. There is an extra argument to show that $g_N \to_{L^2} \hat{\phi}$ (stronger than pointwise convergence as was stated above), so that also $u_N \to_{L^2} \phi \in L^2(\mathbb{R})$, since the Fourier transform is an isomorphism of $L^2(\mathbb{R})$ onto itself. It follows that

$$\langle \phi_{m,k}, \tilde{\phi}_{m,l} \rangle = \lim_{N\to\infty} \langle u_N(t-k), v_N(t-l) \rangle = \delta_{k,l}.$$

While proving this one also establishes that

$$|\hat{\phi}(\omega)| \leq C(1+|\omega|)^{-1/2-\epsilon} \qquad |\hat{\tilde{\phi}}(\omega)| \leq C(1+|\omega|)^{-1/2-\epsilon}, \tag{7.11}$$

where $\epsilon = L - 1/2 - \log B_k / \log 2 > 0$ due to Assumption (7.7). In the paper it is proved that this condition implies that the bases constitute dual frames. The biorthogonality is used to show that they also are dual Riesz bases (i.e. that they also are linearly independent).

7.2.2 Sketch of proof for the biorthogonality of in Proposition 7.7 (2)

The biorthogonality of $\boldsymbol{\psi}_{m,n}$ and $\tilde{\boldsymbol{\psi}}_{m,n}$ can be deduced from the biorthogonality of $\boldsymbol{\phi}_0$ and $\tilde{\boldsymbol{\phi}}_0$ as follows. We have that

$$\begin{aligned}
\langle \psi_{0,k}, \tilde{\psi}_{0,l} \rangle &= \sum_{n_1,n_2} (G_1)_{n_1,1}(H_1)_{1,n_2} \langle \phi_{1,n_1+2k}(t) \tilde{\phi}_{1,n_2+2l}(t) \rangle \\
&= \sum_n (G_1)_{n,1}(H_1)_{1,n+2(k-l)} = \sum_n (H_1)_{1+2(l-k),n}(G_1)_{n,1} = \sum_n H_{1+2(l-k),n} G_{n,1} \\
&= (HG)_{1+2(l-k),1} = \delta_{k,0}.
\end{aligned}$$

Similarly,

$$\begin{aligned}
\langle \psi_{0,k} \tilde{\phi}_{0,l} \rangle &= \sum_{n_1,n_2} (G_1)_{n_1,1}(H_0)_{0,n_2} \langle \phi_{1,n_1+2k}(t) \tilde{\phi}_{1,n_2+2l}(t) \rangle = \sum_n (G_1)_{n,1}(H_0)_{0,n+2(k-l)} \\
&= \sum_n (H_0)_{2(l-k),n}(G_1)_{n,1} = \sum_n H_{2(l-k),n} G_{n,1} = (HG)_{2(l-k),1} = 0 \\
\langle \phi_{0,k} \tilde{\psi}_{0,l} \rangle &= \sum_{n_1,n_2} (G_0)_{n_1,0}(H_1)_{1,n_2} \langle \phi_{1,n_1+2k}(t) \tilde{\phi}_{1,n_2+2l}(t) \rangle = \sum_n (G_0)_{n,0}(H_1)_{1,n+2(k-l)} \\
&= \sum_n (H_1)_{1+2(l-k),n}(G_0)_{n,0} = \sum_n H_{1+2(l-k),n} G_{n,0} = (HG)_{1+2(l-k),0} = 0.
\end{aligned}$$

From this we also get with a simple change of coordinates that

$$\langle \psi_{m,k}, \tilde{\psi}_{m,l} \rangle = \langle \psi_{m,k}, \tilde{\phi}_{m,l} \rangle = \langle \phi_{m,k}, \tilde{\psi}_{m,l} \rangle = 0.$$

Finally, if $m' < m$, $\phi_{m',k'}, \psi_{m',\underline{k}}$ can be written as a linear combination of $\phi_{m,l}$, so that $\langle \phi_{m',k}, \tilde{\psi}_{m,l} \rangle = \langle \psi_{m',k}, \tilde{\psi}_{m,l} \rangle = 0$ due to what we showed above. Similarly, $\langle \tilde{\phi}_{m',k}, \psi_{m,l} \rangle = \langle \tilde{\psi}_{m',k}, \psi_{m,l} \rangle = 0$.

7.2.3 Regularity and vanishing moments

Now assume also that $B_k < 2^{L-1-m}$, so that $\log B_k < L - 1 - m$. We have that $\epsilon = L - 1/2 - \log B_k / \log 2 > L - 1/2 - L + 1 + m = m + 1/2$, so that

$|\hat{\phi}(\omega)| < C(1+|\omega|)^{-1/2-\epsilon} = C(1+|\omega|)^{-m-1-\delta}$ for some $\delta > 0$. This implies that $\hat{\phi}(\omega)(1+|\omega|)^m < C(1+|\omega|)^{-1-\delta} \in L^1$. An important property of the Fourier transform is that $\hat{\phi}(\omega)(1+|\omega|)^m \in L^1$ if and only if ϕ is m times differentiable. This property implies that ϕ, and thus ψ is m times differentiable. Similarly, $\tilde{\phi}$, $\tilde{\psi}$ are $\tilde{m}$ times differentiable.

In [5] it is also proved that if

- $\psi_{m,n}$ and $\tilde{\psi}_{m,n}$ are biorthogonal bases,
- ψ is m times differentiable with all derivatives $\psi^{(l)}(t)$ of order $l \leq m$ bounded, and
- $\tilde{\psi}(t) < C(1+|t|)^{m+1}$,

then $\tilde{\psi}$ has $m+1$ vanisning moments. In our case we have that ψ and $\tilde{\psi}$ have compact support, so that these conditions are satisfied. It follows that $\tilde{\psi}$ has $m+1$ vanisning moments.

In the next section we will construct a wide range of forward and reverse filter bank transforms which invert each other, and which give rise to wavelets.

In [5] one checks that many of these wavelets satisfy (7.7) and (7.8) (implying that they give rise to dual Riesz bases for $L^2(\mathbb{R})$), or the more general (7.10) (implying a certain regularity and a certain number of vanishing moments). Requirements on the filters lengths in order to obtain a given number of vanishing moments are also stated.

Exercise 7.1: Implementation of the cascade algorithm

Let us consider the following code, which shows how the cascade algorithm can be used to plot the scaling functions and the mother wavelet of a wavelet and its dual wavelet with given kernels, over the interval $[a, b]$.

```
def plotwaveletfunctions(invf, a, b):
    """
    Plot the scaling functions and mother wavelets of a wavelet
    and its dual wavelet using the cascade algorithm.

    invf: the IDWT kernel
    a: the left point of the plot interval.
    b: the right point of the plot interval.
    """
    nres = 10
    t = linspace(a, b, (b-a)*2**nres)

    coordsvm = zeros((b-a)*2**nres)
    coordsvm[0] = 1
    IDWTImpl(coordsvm, nres, invf, 0, 0)
    coordsvm *= 2**(nres/2)
    plt.subplot(2, 2, 1)
    coordsvm = concatenate([coordsvm[(b*2**nres):((b-a)*2**nres)], \
                            coordsvm[0:(b*2**nres)]])
    plt.plot(t, coordsvm)
    plt.title('$\phi$')
```

```
coordsvm = zeros((b-a)*2**nres)
coordsvm[b - a] = 1
IDWTImpl(coordsvm, nres, invf, 0, 0)
coordsvm *= 2**(nres/2)
plt.subplot(2, 2, 2)
coordsvm = concatenate([coordsvm[(b*2**nres):((b-a)*2**nres)], \
                        coordsvm[0:(b*2**nres)]])
plt.plot(t, coordsvm)
plt.title('$\psi$')

coordsvm = zeros((b-a)*2**nres)
coordsvm[0] = 1
IDWTImpl(coordsvm, nres, invf, 0, 1)
coordsvm *= 2**(nres/2)
plt.subplot(2, 2, 3)
coordsvm = concatenate([coordsvm[(b*2**nres):((b-a)*2**nres)], \
                        coordsvm[0:(b*2**nres)]])
plt.plot(t, coordsvm)
plt.title('Dual $\phi$')

coordsvm = zeros((b-a)*2**nres)
coordsvm[b - a] = 1
IDWTImpl(coordsvm, nres, invf, 0, 1)
coordsvm *= 2**(nres/2)
plt.subplot(2, 2, 4)
coordsvm = concatenate([coordsvm[(b*2**nres):((b-a)*2**nres)], \
                        coordsvm[0:(b*2**nres)]])
plt.plot(t, coordsvm)
plt.title('Dual $\psi$')
plt.show()
```

a) Run the function `plotwaveletfunctions` with the three different kernels `IDWTKernelHaar`, `IDWTKernelpwl0`, and 'IDWTKernelpwl2' to plot all scaling functions and mother wavelets for the Haar wavelet and the two piecewise linear wavelets we have encountered. This should verify the different plots for these we have seen previously in the book.

b) Explain that the input to `IDWTImpl` in the code above are the coordinates of $\phi_{0,0}$, $\psi_{0,0}$, $\tilde{\phi}_{0,0}$, and $\tilde{\psi}_{0,0}$ in the basis $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \boldsymbol{\psi}_1, \boldsymbol{\psi}_2, \cdots, \boldsymbol{\psi}_{m-1})$, respectively.

c) In the code above, we wanted the functions to be plotted on $[a, b]$. Explain from this why the `coordsvm`-vector have been rearranged as on the line where the `plot`-command is called.

d) In the code above, we turned off symmetric extensions (the `symm`-argument is 0). Attempt to use symmetric extensions instead, and observe the new plots you obtain. Can you explain why these new plots do not show the correct functions, while the previous plots are correct?

e) In the code you see that all values are scaled with the factor $2^{m/2}$ before they are plotted. Can you think out an explanation to why this is done?

Exercise 7.2: Using the cascade algorithm

In Exercise 6.10 we constructed a new mother wavelet $\hat{\psi}$ for piecewise linear functions by finding constants $\alpha, \beta, \gamma, \delta$ so that

$$\hat{\psi} = \psi - \alpha\phi_{0,0} - \beta\phi_{0,1} - \delta\phi_{0,2} - \gamma\phi_{0,N-1}.$$

Use the cascade algorithm to plot $\hat{\psi}$. Do this by using the wavelet kernel for the piecewise linear wavelet (do not use the code above, since we have not implemented kernels for this wavelet yet).

Exercise 7.3: Implementing the traspose transforms

Since the dual of a wavelet is constructed by transposing filters, one may suspect that taking the dual is the same as taking the transpose. However, show that the DWT, the dual DWT, the transpose of the DWT, and the transpose of the dual DWT, can be computed as follows:

```
DWTImpl(x,  m, DWTkernel, 1, 0) # DWT
DWTImpl(x,  m, DWTkernel, 1, 1) # Dual DWT
IDWTImpl(x, m, IDWTkernel, 1, 1) # Transpose of the DWT
IDWTImpl(x, m, IDWTkernel, 1, 0) # Transpose of the dual DWT
```

Similar statements hold for the IDWT as well.

7.3 Vanishing moments

The scaling functions and mother wavelets we constructed in Chapter 5 were very simple. They were however, enough to provide scaling functions which were differentiable. This may clearly be important for signal approximation, at least in cases where we know certain things about the regularity of the functions we approximate. However, there seemed to be nothing which dictated how the mother wavelet should be chosen in order to be useul. To see that this may pose a problem, consider the mother wavelet we hose for piecewise linear functions. Set $N = 1$ and consider the space V_{10}, which has dimension 2^{10}. When we apply a DWT, we start with a function $g_{10} \in V_{10}$. This may be a very good representation of the underlying data. However, when we compute g_{m-1} we just pick every other coefficient from g_m. By the time we get to g_0 we are just left with the first and last coefficient from g_{10}. In some situations this may be adequate, but usually not.

Idea 7.8. *Approximation.*

We would like a wavelet basis to be able to represent f efficiently. By this we mean that the approximation $f^{(m)} = \sum_n c_{0,n}\phi_{0,n} + \sum_{m'<m,n} w_{m',n}\psi_{m',n}$ to f from Observation 7.11 should converge quickly for the f we work with, as m increases. This means that, with relatively few $\psi_{m,n}$, we can create good approximations of f.

In this section we will address a property which the mother wavelet must fulfill in order to be useful in this respect. To motivate this property, let us first use decompose $f \in V_m$ as

$$f = \sum_{n=0}^{N-1} \langle f, \tilde{\phi}_{0,n} \rangle \phi_{0,n} + \sum_{r=0}^{m-1} \sum_{n=0}^{2^r N-1} \langle f, \tilde{\psi}_{r,n} \rangle \psi_{r,n}. \tag{7.12}$$

If f is s times differentiable, it can be represented as $f = P_s(x) + Q_s(x)$, where P_s is a polynomial of degree s, and Q_s is a function which is very small (P_s could for instance be a Taylor series expansion of f). If in addition $\langle t^k, \tilde{\psi} \rangle = 0$, for $k = 1, \ldots, s$, we have also that $\langle t^k, \tilde{\psi}_{r,t} \rangle = 0$ for $r \leq s$, so that $\langle P_s, \tilde{\psi}_{r,t} \rangle = 0$ also. This means that Equation (7.12) can be written

$$\begin{aligned}
f &= \sum_{n=0}^{N-1} \langle P_s + Q_s, \tilde{\phi}_{0,n} \rangle \phi_{0,n} + \sum_{r=0}^{m-1} \sum_{n=0}^{2^r N-1} \langle P_s + Q_s, \tilde{\psi}_{r,n} \rangle \psi_{r,n} \\
&= \sum_{n=0}^{N-1} \langle P_s + Q_s, \tilde{\phi}_{0,n} \rangle \phi_{0,n} + \sum_{r=0}^{m-1} \sum_{n=0}^{2^r N-1} \langle P_s, \tilde{\psi}_{r,n} \rangle \psi_{r,n} + \sum_{r=0}^{m-1} \sum_{n=0}^{2^r N-1} \langle Q_s, \tilde{\psi}_{r,n} \rangle \psi_{r,n} \\
&= \sum_{n=0}^{N-1} \langle f, \tilde{\phi}_{0,n} \rangle \phi_{0,n} + \sum_{r=0}^{m-1} \sum_{n=0}^{2^r N-1} \langle Q_s, \tilde{\psi}_{r,n} \rangle \psi_{r,n}.
\end{aligned}$$

Here the first sum lies in V_0. We see that the wavelet coefficients from W_r are $\langle Q_s, \tilde{\psi}_{r,n} \rangle$, which are very small since Q_s is small. This means that the detail in the different spaces W_r is very small, which is exactly what we aimed for. Let us summarize this as follows:

Theorem 7.9. *Vanishing moments.*

If a function $f \in V_m$ is r times differentiable, and $\tilde{\psi}$ has r vanishing moments, then f can be approximated well from V_0. Moreover, the quality of this approximation improves when r increases.

Having many vanishing moments is thus very good for compression, since the corresponding wavelet basis is very efficient for compression. In particular, if f is a polynomial of degree less than or equal to $k-1$ and $\tilde{\psi}$ has k vanishing moments, then the detail coefficients $w_{m,n}$ are exactly 0. Since (ϕ, ψ) and $(\tilde{\phi}, \tilde{\psi})$ both are wavelet bases, it is equally important for both to have vanishing moments. We will in the following concentrate on the number of vanishing moments of ψ.

The Haar wavelet has one vanishing moment, since $\tilde{\psi} = \psi$ and $\int_0^N \psi(t)dt = 0$ as we noted in Observation 5.14. It is an exercise to see that the Haar wavelet has only one vanishing moment, i.e. $\int_0^N t\psi(t)dt \neq 0$.

Theorem 7.10. *Vanishing moments.*

Assume that the filters are chosen so that the scaling functions exist. Then the following hold

- The number of vanishing moments of $\tilde{\psi}$ equals the multiplicity of a zero at $\omega = \pi$ for $\lambda_{G_0}(\omega)$.

- The number of vanishing moments of ψ equals the multiplicity of a zero at $\omega = \pi$ for $\lambda_{H_0}(\omega)$.

number of vanishing moments of ψ, $\tilde{\psi}$ equal the multiplicities of the zeros of the frequency responses $\lambda_{H_0}(\omega)$, $\lambda_{G_0}(\omega)$, respectively, at $\omega = \pi$.

In other words, the flatter the frequency responses $\lambda_{H_0}(\omega)$ and $\lambda_{G_0}(\omega)$ are near high frequencies ($\omega = \pi$), the better the wavelet functions are for approximation of functions. This is analogous to the smoothing filters we constructed previously, where the use of values from Pascals triangle resulted in filters which behaved like the constant function one at low frequencies. The frequency response for the Haar wavelet had just a simple zero at π, so that it cannot represent functions efficiently. The result also proves why we should consider G_0, H_0 as lowpass filters, G_1, H_1 as highpass filters.

Proof. We have that

$$\lambda_{s_{-\tilde{\psi}(-t)}}(\nu) = -\int_{-\infty}^{\infty} \tilde{\psi}(-t) e^{-2\pi i \nu t} dt. \tag{7.13}$$

By differentiating this expression k times w.r.t. ν (differentiate under the integral sign) we get

$$(\lambda_{s_{-\tilde{\psi}(-t)}})^{(k)}(\nu) = -\int (-2\pi i t)^k \tilde{\psi}(t) e^{-2\pi i \nu t} dt. \tag{7.14}$$

Evaluating this at $\nu = 0$ gives

$$(\lambda_{s_{-\tilde{\psi}(-t)}})^{(k)}(0) = -\int (-2\pi i t)^k \tilde{\psi}(t) dt. \tag{7.15}$$

From this expression it is clear that the number of vanishing moments of $\tilde{\psi}$ equals the multiplicity of a zero at $\nu = 0$ for $\lambda_{s_{-\tilde{\psi}(-t)}}(\nu)$, which we have already shown equals the multiplicity of a zero at $\omega = 0$ for $\lambda_{H_1}(\omega)$. Similarly it follows that the number of vanishing moments of ψ equals the multiplicity of a zero at $\omega = 0$ for $\lambda_{G_1}(\omega)$. Since we know that $\lambda_{G_0}(\omega)$ has the same number of zeros at π as $\lambda_{H_1}(\omega)$ has at 0, and $\lambda_{H_0}(\omega)$ has the same number of zeros at π as $\lambda_{G_1}(\omega)$ has at 0, the result follows. □

These results explain how we can construct ϕ, ψ, $\tilde{\phi}$, $\tilde{\psi}$ from FIR-filters H_0, H_1, G_0, G_1 satisfying the perfect reconstruction condition. Also, the results explain how we can obtain such functions with as much differentiability and as many vanishing moments as we want. We will use these results in the next section to construct interesting wavelets. There we will also cover how we can construct the simplest possible such filters.

There are some details which have been left out in this section: We have not addressed why the wavelet bases we have constructed are linearly independent, and why they span $L^2(\mathbb{R})$. Dual Riesz bases. These details are quite technical, and we refer to [5] for them. Let us also express what we have found in terms of analog filters.

Observation 7.11. *Analog filters.*

Let

$$f(t) = \sum_n c_{m,n}\phi_{m,n} = \sum_n c_{0,n}\phi_{0,n} + \sum_{m'<m,n} w_{m',n}\psi_{m',n} \in V_m.$$

$c_{m,n}$ and $w_{m,n}$ can be computed by sampling the output of an analog filter. To be more precise,

$$c_{m,n} = \langle f, \tilde{\phi}_{m,n}\rangle = \int_0^N f(t)\tilde{\phi}_{m,n}(t)dt = \int_0^N (-\tilde{\phi}_{m,0}(-t))f(2^{-m}n - t)dt$$
$$w_{m,n} = \langle f, \tilde{\psi}_{m,n}\rangle = \int_0^N f(t)\tilde{\psi}_{m,n}(t)dt = \int_0^N (-\tilde{\psi}_{m,0}(-t))f(2^{-m}n - t)dt.$$

In other words, $c_{m,n}$ can be obtained by sampling $s_{-\tilde{\phi}_{m,0}(-t)}(f(t))$ at the points $2^{-m}n$, $w_{m,n}$ by sampling $s_{-\tilde{\psi}_{m,0}(-t)}(f(t))$ at $2^{-m}n$, where the analog filters $s_{-\tilde{\phi}_{m,0}(-t)}$, $s_{-\tilde{\psi}_{m,0}(-t)}$ were defined in Theorem 1.39, i.e.

$$s_{-\tilde{\phi}_{m,0}(-t)}(f(t)) = \int_0^N (-\tilde{\phi}_{m,0}(-s))f(t-s)ds \tag{7.16}$$
$$s_{-\tilde{\psi}_{m,0}(-t)}(f(t)) = \int_0^N (-\tilde{\psi}_{m,0}(-s))f(t-s)ds. \tag{7.17}$$

A similar statement can be made for $\tilde{f} \in \tilde{V}_m$. Here the convolution kernels of the filters were as before, with the exception that ϕ, ψ were replaced by $\tilde{\phi}, \tilde{\psi}$. Note also that, if the functions $\tilde{\phi}$, $\tilde{\psi}$ are symmetric, we can increase the precision in the DWT with the method of symmetric extension also in this more general setting.

7.4 Characterization of wavelets w.r.t. number of vanishing moments

We have seen that wavelets are particularly suitable for approximation of functions when the mother wavelet or the dual mother wavelet have vanishing moments. The more vanishing moments they have, the more attractive they are. In this section we will attempt to characterize wavelets which have a given number of vanishing moments. In particular we will characterize the simplest such, those where the filters have few filters coefficients.

There are two particular cases we will look at. First we will consider the case when all filters are symmetric. Then we will look at the case of orthonormal wavelets. It turns out that these two cases are mutually disjoint (except for trivial examples), but that there is a common result which can be used to characterize the solutions to both problems. We will state the results in terms of the multiplicities of the zeros of λ_{H_0}, λ_{G_0} at π, which we proved are the same as the number of vanishing moments.

7.4.1 Symmetric filters

The main result when the filters are symmetric looks as follows.

Theorem 7.12. *Wavelet crirteria.*
Assume that H_0, H_1, G_0, G_1 are the filters of a wavelet, and that

- the filters are symmetric,
- λ_{H_0} has a zero of multiplicity N_1 at π,
- λ_{G_0} has a zero of multiplicity N_2 at π.

Then N_1 and N_2 are even, and there exist a polynomial Q which satisfies

$$u^{(N_1+N_2)/2}Q(1-u) + (1-u)^{(N_1+N_2)/2}Q(u) = 2. \tag{7.18}$$

so that $\lambda_{H_0}(\omega), \lambda_{G_0}(\omega)$ can be written on the form

$$\lambda_{H_0}(\omega) = \left(\frac{1}{2}(1+\cos\omega)\right)^{N_1/2} Q_1\left(\frac{1}{2}(1-\cos\omega)\right) \tag{7.19}$$

$$\lambda_{G_0}(\omega) = \left(\frac{1}{2}(1+\cos\omega)\right)^{N_2/2} Q_2\left(\frac{1}{2}(1-\cos\omega)\right), \tag{7.20}$$

where $Q = Q_1 Q_2$.

Proof. Since the filters are symmetric, $\lambda_{H_0}(\omega) = \lambda_{H_0}(-\omega)$ and $\lambda_{G_0}(\omega) = \lambda_{G_0}(-\omega)$. Since $e^{in\omega} + e^{-in\omega} = 2\cos(n\omega)$, and since $\cos(n\omega)$ is the real part of $(\cos\omega + i\sin\omega)^n$, which is a polynomial in $\cos^k\omega\sin^l\omega$ with l even, and since $\sin^2\omega = 1-\cos^2\omega$, λ_{H_0} and λ_{G_0} can both be written on the form $P(\cos\omega)$, with P a real polynomial.

Note that a zero at π in $\lambda_{H_0}, \lambda_{G_0}$ corresponds to a factor of the form $1+e^{-i\omega}$, so that we can write

$$\lambda_{H_0}(\omega) = \left(\frac{1+e^{-i\omega}}{2}\right)^{N_1} f(e^{i\omega}) = e^{-iN_1\omega/2}\cos^{N_1}(\omega/2) f(e^{i\omega}),$$

where f is a polynomial. In order for this to be real, we must have that $f(e^{i\omega}) = e^{iN_1\omega/2}g(e^{i\omega})$ where g is real-valued, and then we can write $g(e^{i\omega})$ as a real polynomial in $\cos\omega$. This means that $\lambda_{H_0}(\omega) = \cos^{N_1}(\omega/2)P_1(\cos\omega)$, and similarly for $\lambda_{G_0}(\omega)$. Clearly this can be a polynomial in $e^{i\omega}$ only if N_1 is even. Both N_1 and N_2 must then be even, and we can write

$$\begin{aligned}\lambda_{H_0}(\omega) &= \cos^{N_1}(\omega/2)P_1(\cos\omega) = (\cos^2(\omega/2))^{N_1/2}P_1(1-2\sin^2(\omega/2)) \\ &= (\cos^2(\omega/2))^{N_1/2}Q_1(\sin^2(\omega/2)),\end{aligned}$$

where we have used that $\cos\omega = 1 - 2\sin^2(\omega/2)$, and defined Q_1 by the relation $Q_1(x) = P_1(1-2x)$. Similarly we can write $\lambda_{G_0}(\omega) = (\cos^2(\omega/2))^{N_2/2} Q_2(\sin^2(\omega/2))$ for another polynomial Q_2. Using the identities

$$\cos^2\frac{\omega}{2} = \frac{1}{2}(1+\cos\omega) \qquad\qquad \sin^2\frac{\omega}{2} = \frac{1}{2}(1-\cos\omega),$$

we see that λ_{H_0} and λ_{G_0} satisfy equations (7.19) and (7.20). With $Q = Q_1 Q_2$, Equation (6.20) can now be rewritten as

$$\begin{aligned}
2 &= \lambda_{G_0}(\omega)\lambda_{H_0}(\omega) + \lambda_{G_0}(\omega+\pi)\lambda_{H_0}(\omega+\pi) \\
&= \big(\cos^2(\omega/2)\big)^{(N_1+N_2)/2} Q(\sin^2(\omega/2)) + \big(\cos^2((\omega+\pi)/2)\big)^{(N_1+N_2)/2} Q(\sin^2((\omega+\pi)/2)) \\
&= (\cos^2(\omega/2))^{(N_1+N_2)/2} Q(\sin^2(\omega/2)) + (\sin^2(\omega/2))^{(N_1+N_2)/2} Q(\cos^2(\omega/2)) \\
&= (\cos^2(\omega/2))^{(N_1+N_2)/2} Q(1-\cos^2(\omega/2)) + (1-\cos^2(\omega/2))^{(N_1+N_2)/2} Q(\cos^2(\omega/2))
\end{aligned}$$

Setting $u = \cos^2(\omega/2)$ we see that Q must fulfill the equation

$$u^{(N_1+N_2)/2} Q(1-u) + (1-u)^{(N_1+N_2)/2} Q(u) = 2,$$

which is Equation (7.18). This completes the proof. □

While this result characterizes all wavelets with a given number of vanishing moments, it does not say which of these have fewest filter coefficients. The polynomial Q decides the length of the filters H_0, G_0, however, so that what we need to do is to find the polynomial Q of smallest degree. In this direction, note first that the polynomials $u^{N_1+N_2}$ and $(1-u)^{N_1+N_2}$ have no zeros in common. Bezouts theorem, proved in Section 7.4.3, states that the equation

$$u^N q_1(u) + (1-u)^N q_2(u) = 1 \tag{7.21}$$

has unique solutions q_1, q_2 with $\deg(q_1), \deg(q_2) < (N_1+N_2)/2$. To find these solutions, substituting $1-u$ for u gives the following equations:

$$\begin{aligned}
u^N q_1(u) + (1-u)^N q_2(u) &= 1 \\
u^N q_2(1-u) + (1-u)^N q_1(1-u) &= 1,
\end{aligned}$$

and uniqueness in Bezouts theorem gives that $q_1(u) = q_2(1-u)$, and $q_2(u) = q_1(1-u)$. Equation (7.21) can thus be stated as

$$u^N q_2(1-u) + (1-u)^N q_2(u) = 1,$$

and comparing with Equation (7.18) (set $N = (N_1+N_2)/2$) we see that $Q(u) = 2q_2(u)$. $u^N q_1(u) + (1-u)^N q_2(u) = 1$ now gives

$$\begin{aligned} q_2(u) &= (1-u)^{-N}(1-u^N q_1(u)) = (1-u)^{-N}(1-u^N q_2(1-u)) \\ &= \left(\sum_{k=0}^{N-1}\binom{N+k-1}{k}u^k + O(u^N)\right)(1-u^N q_2(1-u)) \\ &= \sum_{k=0}^{N-1}\binom{N+k-1}{k}u^k + O(u^N), \end{aligned}$$

where we have used the first N terms in the Taylor series expansion of $(1-u)^{-N}$ around 0. Since q_2 is a polynomial of degree $N-1$, we must have that

$$Q(u) = 2q_2(u) = 2\sum_{k=0}^{N-1}\binom{N+k-1}{k}u^k. \tag{7.22}$$

Define $Q^{(N)}(u) = 2\sum_{k=0}^{N-1}\binom{N+k-1}{k}u^k$. The first $Q^{(N)}$ are

$$\begin{aligned} Q^{(1)}(u) &= 2 & Q^{(2)}(u) &= 2+4u \\ Q^{(3)}(u) &= 2+6u+12u^2 & Q^{(4)}(u) &= 2+8u+20u^2+40u^3, \end{aligned}$$

for which we compute

$$\begin{aligned} Q^{(1)}\left(\frac{1}{2}(1-\cos\omega)\right) &= 2 \\ Q^{(2)}\left(\frac{1}{2}(1-\cos\omega)\right) &= -e^{-i\omega}+4-e^{i\omega} \\ Q^{(3)}\left(\frac{1}{2}(1-\cos\omega)\right) &= \frac{3}{4}e^{-2i\omega}-\frac{9}{2}e^{-i\omega}+\frac{19}{2}-\frac{9}{2}e^{i\omega}+\frac{3}{4}e^{2i\omega} \\ Q^{(4)}\left(\frac{1}{2}(1-\cos\omega)\right) &= -\frac{5}{8}e^{-3i\omega}+5e^{-2i\omega}-\frac{131}{8}e^{-i\omega}+26-\frac{131}{8}e^{i\omega}+5e^{2i\omega}-\frac{5}{8}e^{3i\omega}, \end{aligned}$$

Thus in order to construct wavelets where $\lambda_{H_0}, \lambda_{G_0}$ have as many zeros at π as possible, and where there are as few filter coefficients as possible, we need to compute the polynomials above, factorize them into polynomials Q_1 and Q_2, and distribute these among λ_{H_0} and λ_{G_0}. Since we need real factorizations, we must in any case pair complex roots. If we do this we obtain the factorizations

$$
\begin{aligned}
Q^{(1)}\left(\frac{1}{2}(1-\cos\omega)\right) &= 2 \\
Q^{(2)}\left(\frac{1}{2}(1-\cos\omega)\right) &= \frac{1}{3.7321}(e^{i\omega}-3.7321)(e^{-i\omega}-3.7321) \\
Q^{(3)}\left(\frac{1}{2}(1-\cos\omega)\right) &= \frac{3}{4}\frac{1}{9.4438}(e^{2i\omega}-5.4255e^{i\omega}+9.4438) \\
&\quad \times (e^{-2i\omega}-5.4255e^{-i\omega}+9.4438) \\
Q^{(4)}\left(\frac{1}{2}(1-\cos\omega)\right) &= \frac{5}{8}\frac{1}{3.0407}\frac{1}{7.1495}(e^{i\omega}-3.0407)(e^{2i\omega}-4.0623e^{i\omega}+7.1495) \\
&\quad \times (e^{-i\omega}-3.0407)(e^{-2i\omega}-4.0623e^{-i\omega}+7.1495), \quad (7.23)
\end{aligned}
$$

The factors in these factorizations can be distributed as factors in the frequency responses of $\lambda_{H_0}(\omega)$, and $\lambda_{G_0}(\omega)$. One possibility is to let one of these frequency responses absorb all the factors, another possibility is to split the factors as evenly as possible across the two. When a frequency response absorbs more factors, the corresponding filter gets more filter coefficients. In the following examples, both factor distribution strategies will be encountered. Note that it is straightforward to use your computer to factor Q into a product of polynomials Q_1 and Q_2. First the `roots` function can be used to find the roots in the polynomials. Then the `conv` function can be used to multiply together factors corresponding to different roots, to obtain the coefficients in the polynomials Q_1 and Q_2.

7.4.2 Orthonormal wavelets

Now we turn to the case of orthonormal wavelets, i.e. where $G_0 = (H_0)^T$, $G_1 = (H_1)^T$. For simplicity we will assume $d = 0, \alpha = -1$ in conditions (6.18) and (6.19) (this corresponded to requiring $\lambda_{H_1}(\omega) = -\overline{\lambda_{H_0}(\omega+\pi)}$ in the definition of alternative QMF filter banks). We will also assume for simplicity that G_0 is *causal*, meaning that $t_{-1}, t_{-2}, \ldots$ all are zero (the other solutions can be derived from this). We saw that the Haar wavelet was such an orthonormal wavelet. We have the following result:

Theorem 7.13. *Criteria for perfect reconstruction.*

Assume that H_0, H_1, G_0, G_1 are the filters of an orthonormal wavelet (i.e. $H_0 = (G_0)^T$ and $H_1 = (G_1)^T$) which also is an alternative QMF filter bank (i.e. $\lambda_{H_1}(\omega) = -\overline{\lambda_{H_0}(\omega+\pi)}$). Assume also that $\lambda_{G_0}(\omega)$ has a zero of multiplicity N at π and that G_0 is causal. Then there exists a polynomial Q which satisfies

$$u^N Q(1-u) + (1-u)^N Q(u) = 2, \tag{7.24}$$

so that if f is another polynomial which satisfies $f(e^{i\omega})f(e^{-i\omega}) = Q\left(\frac{1}{2}(1-\cos\omega)\right)$, $\lambda_{G_0}(\omega)$ can be written on the form

$$\lambda_{G_0}(\omega) = \left(\frac{1+e^{-i\omega}}{2}\right)^N f(e^{-i\omega}), \tag{7.25}$$

We avoided stating $\lambda_{H_0}(\omega)$ in this result, since the relation $H_0 = (G_0)^T$ gives that $\lambda_{H_0}(\omega) = \overline{\lambda_{G_0}(\omega)}$. In particular, $\lambda_{H_0}(\omega)$ also has a zero of multiplicity N at π. That G_0 is causal is included to simplify the expression further.

Proof. The proof is very similar to the proof of Theorem 7.12. N vanishing moments and that G_0 is causal means that we can write

$$\lambda_{G_0}(\omega) = \left(\frac{1+e^{-i\omega}}{2}\right)^N f(e^{-i\omega}) = (\cos(\omega/2))^N e^{-iN\omega/2} f(e^{-i\omega}),$$

where f is a real polynomial. Also

$$\lambda_{H_0}(\omega) = \overline{\lambda_{G_0}(\omega)} = (\cos(\omega/2))^N e^{iN\omega/2} f(e^{i\omega}).$$

Condition (6.20) now says that

$$\begin{aligned} 2 &= \lambda_{G_0}(\omega)\lambda_{H_0}(\omega) + \lambda_{G_0}(\omega+\pi)\lambda_{H_0}(\omega+\pi) \\ &= (\cos^2(\omega/2))^N f(e^{i\omega}) f(e^{-i\omega}) + (\sin^2(\omega/2))^N f(e^{i(\omega+\pi)}) f(e^{-i(\omega+\pi)}). \end{aligned}$$

Now, the function $f(e^{i\omega})f(e^{-i\omega})$ is symmetric around 0, so that it can be written on the form $P(\cos\omega)$ with P a polynomial, so that

$$\begin{aligned} 2 &= (\cos^2(\omega/2))^N P(\cos\omega) + (\sin^2(\omega/2))^N P(\cos(\omega+\pi)) \\ &= (\cos^2(\omega/2))^N P(1-2\sin^2(\omega/2)) + (\sin^2(\omega/2))^N P(1-2\cos^2(\omega/2)). \end{aligned}$$

If we as in the proof of Theorem 7.12 define Q by $Q(x) = P(1-2x)$, we can write this as

$$(\cos^2(\omega/2))^N Q(\sin^2(\omega/2)) + (\sin^2(\omega/2))^N Q(\cos^2(\omega/2)) = 2,$$

which again gives Equation (7.18) for finding Q. What we thus need to do is to compute the polynomial $Q\left(\frac{1}{2}(1-\cos\omega)\right)$ as before, and consider the different factorizations of this on the form $f(e^{i\omega})f(e^{-i\omega})$. Since this polynomial is symmetric, a is a root if and only $1/a$ is, and if and only if $\bar{a}$ is. If the real roots are

$$b_1, \ldots, b_m, 1/b_1, \ldots, 1/b_m,$$

and the complex roots are

$$a_1, \ldots, a_n, \overline{a_1}, \ldots, \overline{a_n} \text{ and } 1/a_1, \ldots, 1/a_n, \overline{1/a_1}, \ldots, \overline{1/a_n},$$

we can write

$$\begin{aligned}
&Q\left(\frac{1}{2}(1-\cos\omega)\right) \\
&= K(e^{-i\omega} - b_1)\ldots(e^{-i\omega} - b_m) \\
&\times (e^{-i\omega} - a_1)(e^{-i\omega} - \overline{a_1})(e^{-i\omega} - a_2)(e^{-i\omega} - \overline{a_2})\cdots(e^{-i\omega} - a_n)(e^{-i\omega} - \overline{a_n}) \\
&\times (e^{i\omega} - b_1)\ldots(e^{i\omega} - b_m) \\
&\times (e^{i\omega} - a_1)(e^{i\omega} - \overline{a_1})(e^{i\omega} - a_2)(e^{i\omega} - \overline{a_2})\cdots(e^{i\omega} - a_n)(e^{i\omega} - \overline{a_n})
\end{aligned}$$

where K is a constant. We now can define the polynomial f by

$$\begin{aligned}
f(e^{i\omega}) =& \sqrt{K}(e^{i\omega} - b_1)\ldots(e^{i\omega} - b_m) \\
&\times (e^{i\omega} - a_1)(e^{i\omega} - \overline{a_1})(e^{i\omega} - a_2)(e^{i\omega} - \overline{a_2})\cdots(e^{i\omega} - a_n)(e^{i\omega} - \overline{a_n})
\end{aligned}$$

in order to obtain a factorization $Q\left(\frac{1}{2}(1-\cos\omega)\right) = f(e^{i\omega})f(e^{-i\omega})$. This concludes the proof. □

In the previous proof we note that the polynomial f is not unique - we could pair the roots in many different ways. The new algorithm is thus as follows:

- As before, write $Q\left(\frac{1}{2}(1-\cos\omega)\right)$ as a polynomial in $e^{i\omega}$, and find the roots.
- Split the roots into the two classes

$$\{b_1, \ldots., b_m, a_1, \ldots, a_n, \overline{a_1}, \ldots, \overline{a_n}\}$$

and

$$\{1/b_1, \ldots, 1/b_m, 1/a_1, \ldots, 1/a_n, \overline{1/a_1}, \ldots, \overline{1/a_n}\},$$

and form the polynomial f as above.

- Compute $\lambda_{G_0}(\omega) = \left(\frac{1+e^{-i\omega}}{2}\right)^N f(e^{-i\omega})$.

Clearly the filters obtained with this strategy are not symmetric since f is not symmetric. In Section 7.7 we will take a closer look at wavelets constructed in this way.

7.4.3 The proof of Bezouts theorem

Theorem 7.14. *Existence of polynomials.*

If p_1 and p_2 are two polynomials, of degrees n_1 and n_2 respectively, with no common zeros, then there exist unique polynomials q_1, q_2, of degree less than n_2, n_1, respectively, so that

$$p_1(x)q_1(x) + p_2(x)q_2(x) = 1. \tag{7.26}$$

Proof. We first establish the existence of q_1, q_2 satisfying Equation (7.26). Denote by $\deg(P)$ the degree of the polynomial P. Renumber the polynomials if necessary, so that $n_1 \geq n_2$. By polynomial division, we can now write

$$p_1(x) = a_2(x)p_2(x) + b_2(x),$$

where $\deg(a_2) = \deg(p_1) - \deg(p_2)$, $\deg(b_2) < deg(p_2)$. Similarly, we can write

$$p_2(x) = a_3(x)b_2(x) + b_3(x),$$

where $\deg(a_3) = \deg(p_2) - \deg(b_2)$, $\deg(b_3) < \deg(b_2)$. We can repeat this procedure, so that we obtain a sequence of polynomials $a_n(x), b_n(x)$ so that

$$b_{n-1}(x) = a_{n+1}(x)b_n(x) + b_{n+1}(x), \tag{7.27}$$

where $\deg a_{n+1} = \deg(b_{n-1}) - \deg(b_n)$, $deg(b_{n+1} < \deg(b_n)$. Since $\deg(b_n)$ is strictly decreasing, we must have that $b_{N+1} = 0$ and $b_N \neq 0$ for some N, i.e. $b_{N-1}(x) = a_{N+1}(x)b_N(x)$. Since $b_{N-2} = a_N b_{N-1} + b_N$, it follows that b_{N-2} can be divided by b_N, and by induction that all b_n can be divided by b_N, in particlar p_1 and p_2 can be divided by b_N. Since p_1 and p_2 have no common zeros, b_N must be a nonzero constant.

Using Equation (7.27), we can write recursively

$$\begin{aligned} b_N &= b_{N-2} - a_N b_{N-1} \\ &= b_{N-2} - a_N(b_{N-3} - a_{N-1}b_{N-2}) \\ &= (1 + a_N a_{N-1})b_{N-2} - a_N b_{N-3}. \end{aligned}$$

By induction we can write

$$b_N = a^{(1)}_{N,k} b_{N-k} + a^{(2)}_{N,k} b_{N-k-1}.$$

We see that the leading order term for $a^{(1)}_{N,k}$ is $a_N \cdots a_{N-k+1}$, which has degree

$$(\deg(b_{N-2}) - \deg(b_{N-1}) + \cdots + (\deg(b_{N-k-1}) - \deg(b_{N-k}) = \deg(b_{N-k-1}) - \deg(b_{N-1}),$$

while the leading order term for $a^{(2)}_{N,k}$ is $a_N \cdots a_{N-k+2}$, which similarly has order $\deg(b_{N-k}) - \deg(b_{N-1})$. For $k = N - 1$ we find

$$b_N = a^{(1)}_{N,N-1} b_1 + a^{(2)}_{N,N-1} b_0 = a^{(1)}_{N,N-1} p_2 + a^{(2)}_{N,N-1} p_1, \tag{7.28}$$

with $\deg(a^{(1)}_{N,N-1}) = \deg(p_1) - \deg(b_{N-1}) < \deg(p_1)$ (since by construction $\deg(b_{N-1}) > 0$), and $\deg(a^{(2)}_{N,N-1}) = \deg(p_2) - \deg(b_{N-1}) < \deg(p_2)$. From Equation (7.28) it follows that $q_1 = a^{(2)}_{N,N-1}/b_N$ and $q_2 a^{(1)}_{N,N-1}/b_N$ satisfies Equation (7.26), and that they satisfy the required degree constraints.

Now we turn to uniquness of solutions q_1, q_2. Assume that r_1, r_2 are two other solutions to Equation (7.26). Then

$$p_1(q_1 - r_1) + p_2(q_2 - r_2) = 0.$$

Since p_1 and p_2 have no zeros in common this means that every zero of p_2 is a zero of $q_1 - r_1$, with at least the same multiplicity. If $q_1 \neq r_1$, this means that $\deg(q_1 - r_1) \geq \deg(p_2)$, which is impossible since $\deg(q_1) < \deg(p_2)$, $\deg(r_1) < \deg(p_2)$. Hence $q_1 = r_1$. Similarly $q_2 = r_2$, establishing uniqueness. □

Exercise 7.4: Compute filters

Compute the filters H_0, G_0 in Theorem 7.12 when $N = N_1 = N_2 = 4$, and $Q_1 = Q^{(4)}$, $Q_2 = 1$. Compute also filters H_1, G_1 so that we have perfect reconstruction (note that these are not unique).

7.5 A design strategy suitable for lossless compression

We choose $Q_1 = Q$, $Q_2 = 1$. In this case there is no need to find factors in Q. The frequency responses of the filters in the filter factorization are

$$\begin{aligned} \lambda_{H_0}(\omega) &= \left(\frac{1}{2}(1+\cos\omega)\right)^{N_1/2} Q^{(N)}\left(\frac{1}{2}(1-\cos\omega)\right) \\ \lambda_{G_0}(\omega) &= \left(\frac{1}{2}(1+\cos\omega)\right)^{N_2/2}, \end{aligned} \tag{7.29}$$

where $N = (N_1 + N_2)/2$. Since $Q^{(N)}$ has degree $N-1$, λ_{H_0} has degree $N_1 + N_1 + N_2 - 2 = 2N_1 + N_2 - 2$, and λ_{G_0} has degree N_2. These are both even numbers, so that the filters have odd length. The names of these filters are indexed by the filter lengths, and are called *Spline wavelets*, since, as we now now will show, the scaling function for this design strategy is the B-spline of order N_2: we have that

$$\lambda_{G_0}(\omega) = \frac{1}{2^{N_2/2}}(1+\cos\omega)^{N_2/2} = \cos(\omega/2)^{N_2}.$$

Letting s be the analog filter with convolution kernel ϕ we can as in Equation (7.3) write

$$\begin{aligned}\lambda_s(f) &= \lambda_s(f/2^k)\prod_{i=1}^{k}\frac{\lambda_{G_0}(2\pi f/2^i)}{2} = \lambda_s(f/2^k)\prod_{i=1}^{k}\frac{\cos^{N_2}(\pi f/2^i)}{2} \\ &= \lambda_s(f/2^k)\prod_{i=1}^{k}\left(\frac{\sin(2\pi f/2^i)}{2\sin(\pi f/2^i)}\right)^{N_2} = \lambda_s(f/2^k)\left(\frac{\sin(\pi f)}{2^k\sin\pi f/2^k}\right)^{N_2},\end{aligned}$$

where we have used the identity $\cos\omega = \frac{\sin(2\omega)}{2\sin\omega}$. If we here let $k\to\infty$, and use the identity $\lim_{f\to 0}\frac{\sin f}{f} = 1$, we get that

$$\lambda_s(f) = \lambda_s(0)\left(\frac{\sin(\pi f)}{\pi f}\right)^{N_2}.$$

On the other hand, the frequency response of $\chi_{[-1/2,1/2)}(t)$

$$\begin{aligned}&= \int_{-1/2}^{1/2} e^{-2\pi i f t}dt = \left[\frac{1}{-2\pi i f}e^{-2\pi i f t}\right]_{-1/2}^{1/2} \\ &= \frac{1}{-2\pi i f}(e^{-\pi i f} - e^{\pi i f}) = \frac{1}{-2\pi i f}2i\sin(-\pi f) = \frac{\sin(\pi f)}{\pi f}.\end{aligned}$$

Due to this $\left(\frac{\sin(\pi f)}{\pi f}\right)^{N_2}$ is the frequency response of $*_{k=1}^{N_2}\chi_{[-1/2,1/2)}(t)$. By the uniqueness of the frequency response we have that $\phi(t) = \hat{\phi}(0)*_{k=1}^{N_2}\chi_{[-1/2,1/2)}(t)$. In Exercise 7.6 you will be asked to show that this scaling function gives rise to the multiresolution analysis of functions which are piecewise polynomials which are differentiable at the borders, also called *splines*. This explains why this type of wavelet is called a spline wavelet. To be more precise, the resolution spaces are as follows.

Definition 7.15. *Resolution spaces of piecewise polynomials.*

We define V_m as the subspace of functions which are $r-1$ times continuously differentiable and equal to a polynomial of degree r on any interval of the form $[n2^{-m},(n+1)2^{-m}]$.

Note that the piecewise linear wavelet can be considered as the first Spline wavelet. This is further considered in the following example.

Example 7.16. *The Spline 5/3 wavelet.*

For the case of $N_1 = N_2 = 2$ when the first design strategy is used, equations (7.19) and (7.20) take the form

$$\lambda_{G_0}(\omega) = \frac{1}{2}(1+\cos\omega) = \frac{1}{4}e^{i\omega} + \frac{1}{2} + \frac{1}{4}e^{-i\omega}$$
$$\lambda_{H_0}(\omega) = \frac{1}{2}(1+\cos\omega)Q^{(1)}\left(\frac{1}{2}(1-\cos\omega)\right) = \frac{1}{4}(2+e^{i\omega}+e^{-i\omega})(4-e^{i\omega}-e^{-i\omega})$$
$$= -\frac{1}{4}e^{2i\omega} + \frac{1}{2}e^{i\omega} + \frac{3}{2} + \frac{1}{2}e^{-i\omega} - \frac{1}{4}e^{-2i\omega}.$$

The filters G_0, H_0 are thus

$$G_0 = \left\{\frac{1}{4}, \underline{\frac{1}{2}}, \frac{1}{4}\right\} \qquad H_0 = \left\{-\frac{1}{4}, \frac{1}{2}, \underline{\frac{3}{2}}, \frac{1}{2}, -\frac{1}{4}\right\}$$

The length of the filters are 3 and 5 in this case, so that this wavelet is called the *Spline 5/3 wavelet*. Up to a constant, the filters are seen to be the same as those of the alternative piecewise linear wavelet, see Example 6.19. Now, how do we find the filters (G_1, H_1)? Previously we saw how to find the constant α in Theorem 6.17 when we knew one of the two pairs (G_0, G_1), (H_0, H_1). This was the last part of information we needed in order to construct the other two filters. Here we know (G_0, H_0) instead. In this case it is even easier to find (G_1, H_1) since we can set $\alpha = 1$. This means that (G_1, H_1) can be obtained simply by adding alternating signs to (G_0, H_0), i.e. they are the corresponding highpass filters. We thus can set

$$G_1 = \left\{-\frac{1}{4}, -\frac{1}{2}, \underline{\frac{3}{2}}, -\frac{1}{2}, -\frac{1}{4}\right\} \qquad H_1 = \left\{-\frac{1}{4}, \underline{\frac{1}{2}}, -\frac{1}{4}\right\}.$$

We have now found all the filters. It is clear that the forward and reverse filter bank transforms here differ only by multiplication with a constant from those of the the alternative piecewise linear wavelet, so that this gives the same scaling function and mother wavelet as that wavelet.

The coefficients for the Spline wavelets are always dyadic fractions, and are therefore suitable for lossless compression, as they can be computed using low precision arithmetic and bitshift operations. The particular Spline wavelet from Example 7.16 is used for lossless compression in the JPEG2000 standard.

Exercise 7.5: Viewing the frequency response

In this exercise we will see how we can view the frequency responses, scaling functions and mother wavelets for any spline wavelet.

a) Write a function which takes N_1 and N_2 as input, computes the filter coefficients of H_0 and G_0 using equation (7.29) in the compendium, and plots the frequency responses of G_0 and H_0. Recall that the frequency response can be obtained from the filter coefficients by taking a DFT. You will have use for the `conv` function here, and that the frequency response $(1+\cos\omega)/2$ corresponds to the filter with coefficients $\{1/4, \underline{1/2}, 1/4\}$.

b) Recall that in Exercise 6.12 we implemented DWT and IDWT kernels, which worked for any set of symmetric filters. Combine these kernels with your computation of the filter coefficients from a), and use the function `plotwaveletfunctions` to plot the corresponding scaling functions and mother wavelets for different N_1 and N_2.

Exercise 7.6: Wavelets based on higher degree polynomials

Show that $B_r(t) = *_{k=1}^{r} \chi_{[-1/2,1/2)}(t)$ is $r-2$ times differentiable, and equals a polynomial of degree $r-1$ on subintervals of the form $[n, n+1]$. Explain why these functions can be used as basis for the spaces V_j of functions which are piecewise polynomials of degree $r-1$ on intervals of the form $[n2^{-m}, (n+1)2^{-m}]$, and $r-2$ times differentiable. B_r is also called the B-spline of order r.

7.6 A design strategy suitable for lossy compression

The factors of Q are split evenly among Q_1 and Q_2. In this case we need to factorize Q into a product of real polynomials. This can be done by finding all roots, and pairing the complex conjugate roots into real second degree polynomials (if Q is real, its roots come in conjugate pairs), and then distribute these as evenly as possible among Q_1 and Q_2. These filters are called the CDF-wavelets, after Cohen, Daubechies, and Feauveau, who discovered them.

Example 7.17. *The CDF 9/7 wavelet.*

We choose $N_1 = N_2 = 4$. In Equation (7.23) we pair inverse terms to obtain

$$\begin{aligned} Q^{(3)}\left(\frac{1}{2}(1-\cos\omega)\right) &= \frac{5}{8}\frac{1}{3.0407}\frac{1}{7.1495}(e^{i\omega}-3.0407)(e^{-i\omega}-3.0407) \\ &\quad\times(e^{2i\omega}-4.0623e^{i\omega}+7.1495)(e^{-2i\omega}-4.0623e^{-i\omega}+7.1495) \\ &= \frac{5}{8}\frac{1}{3.0407}\frac{1}{7.1495}(-3.0407e^{i\omega}+10.2456-3.0407e^{-i\omega}) \\ &\quad\times(7.1495e^{2i\omega}-33.1053e^{i\omega}+68.6168-33.1053e^{-i\omega}+7.1495e^{-2i\omega}). \end{aligned}$$

We can write this as Q_1Q_2 with $Q_1(0) = Q_2(0)$ when

$$\begin{aligned} Q_1(\omega) &= -1.0326e^{i\omega}+3.4795-1.0326e^{-i\omega} \\ Q_2(\omega) &= 0.6053e^{2i\omega}-2.8026e^{i\omega}+5.8089-2.8026e^{-i\omega}+0.6053e^{-2i\omega}, \end{aligned}$$

from which we obtain

$$\begin{aligned}
\lambda_{G_0}(\omega) &= \left(\frac{1}{2}(1+\cos\omega)\right)^2 Q_1(\omega) \\
&= -0.0645e^{3i\omega} - 0.0407e^{2i\omega} + 0.4181e^{i\omega} + 0.7885 \\
&\quad + 0.4181e^{-i\omega} - 0.0407e^{-2i\omega} - 0.0645e^{-3i\omega} \\
\lambda_{H_0}(\omega) &= \left(\frac{1}{2}(1+\cos\omega)\right)^2 40Q_2(\omega) \\
&= 0.0378e^{4i\omega} - 0.0238e^{3i\omega} - 0.1106e^{2i\omega} + 0.3774e^{i\omega} + 0.8527 \\
&\quad + 0.3774e^{-i\omega} - 0.1106e^{-2i\omega} - 0.0238e^{-3i\omega} + 0.0378e^{-4i\omega}.
\end{aligned}$$

The filters G_0, H_0 are thus

$$G_0 = \{0.0645, 0.0407, -0.4181, \underline{-0.7885}, -0.4181, 0.0407, 0.0645\}$$
$$H_0 = \{-0.0378, 0.0238, 0.1106, -0.3774, \underline{-0.8527}, -0.3774, 0.1106, 0.0238, -0.0378\}.$$

The corresponding frequency responses are plotted in Figure 7.2.

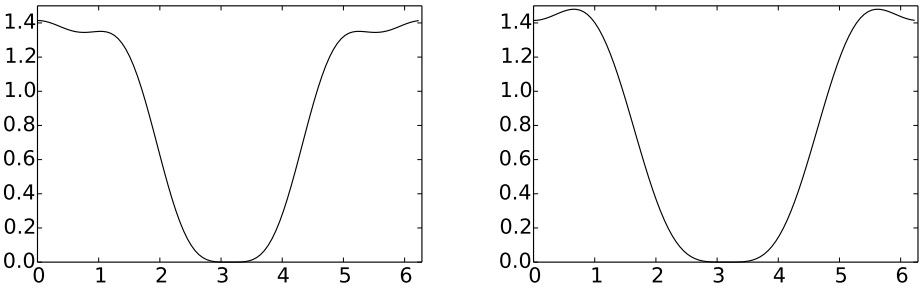

Figure 7.2: The frequency responses $\lambda_{H_0}(\omega)$ (left) and $\lambda_{G_0}(\omega)$ (right) for the CDF 9/7 wavelet.

It is seen that both filters are lowpass filters also here, and that the are closer to an ideal bandpass filter. Here, the frequency response acts even more like the constant zero function close to π, proving that our construction has worked. We also get

$$G_1 = \{-0.0378, -0.0238, 0.1106, 0.3774, \underline{-0.8527}, 0.3774, 0.1106, -0.0238, -0.0378\}$$
$$H_1 = \{-0.0645, 0.0407, 0.4181, \underline{-0.7885}, 0.4181, 0.0407, -0.0645\}.$$

The length of the filters are 9 and 7 in this case, so that this wavelet is called the *CDF 9/7 wavelet*. This wavelet is for instance used for lossy compression with JPEG2000 since it gives a good tradeoff between complexity and compression.

In Example 6.19 we saw that we had analytical expressions for the scaling functions and the mother wavelet, but that we could not obtain this for the dual

functions. For the CDF 9/7 wavelet it turns out that none of the four functions have analytical expressions. Let us therefore use the cascade algorithm, as we did in Example 7.1 to plot these functions. Note first that since G_0 has 7 filter coefficients, and G_1 has 9 filter coefficients, it follows from Theorem 6.11 that $\text{supp}(\phi) = [-3, 3]$, $\text{supp}(\psi) = [-3, 4]$, $\text{supp}(\tilde{\phi}) = [-4, 4]$, and $\text{supp}(\tilde{\psi}) = [-3, 4]$. Plotting the scaling functions and mother wavelets over these supports using the cascade algorithm gives the plots in Figure 7.3. Again they have irregular shapes, but now at least the functions and dual functions more resemble each other.

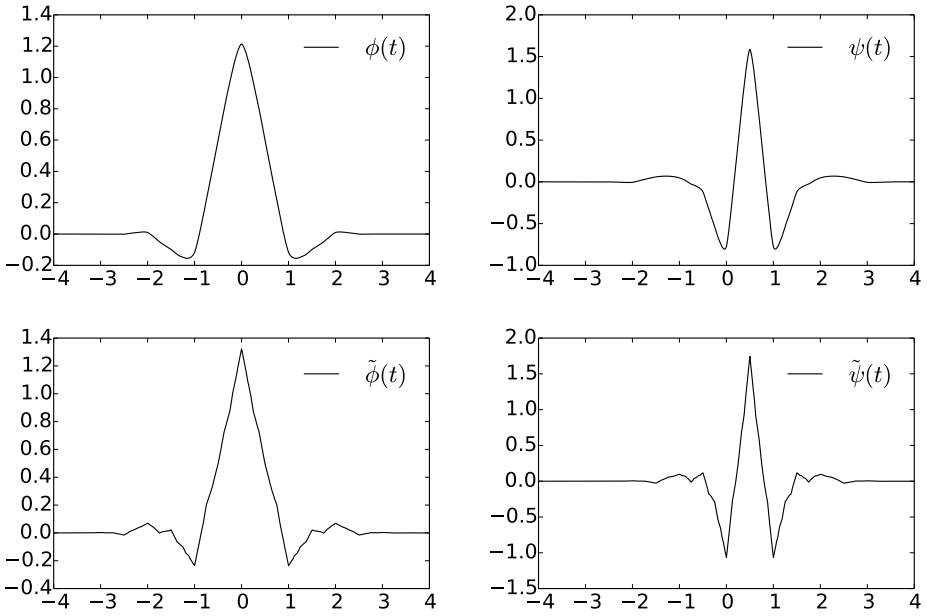

Figure 7.3: Scaling functions and mother wavelets for the CDF 9/7 wavelet.

In the above example there was a unique way of factoring Q into a product of real polynomials. For higher degree polynomials there is no unique way to form to distribute the factors, and we will not go into what strategy can be used for this. In general, the steps we must go through are as follows:

- Compute the polynomial Q, and find its roots.
- Pair complex conjugate roots into real second degree polynomials, and form polynomials Q_1, Q_2.
- Compute the coefficients in equations (7.19) and (7.20).

Exercise 7.7: Generate plots

Generate the plots from Figure 7.3 using the cascade algorithm. Reuse the code from Exercise 7.1 in order to achieve this.

7.7 Orthonormal wavelets

Since the filters here are not symmetric, the method of symmetric extension does not work in the same simple way as before. This partially explains why symmetric filters are used more often: They may not be as efficient in representing functions, since the corresponding basis is not orthogonal, but their simple implementation still makes them attractive.

In Theorem 7.13 we characterized orthonormal wavelets where G_0 was causal. All our filters have an even number, say $2L$, of filter coefficients. We can also find an orthonormal wavelet where H_0 has a minimum possible overweight of filter coefficients with negative indices, H_1 a minimum possible overweight of positive indices, i.e. that the filters can be written with the following compact notation:

$$H_0 = \{t_{-L}, \dots, t_{-1}, \underline{t_0}, t_1, \dots, t_{L-1}\} \quad H_1 = \{s_{-L+1}, \dots, s_{-1}, \underline{s_0}, s_1, \dots, s_L\}. \tag{7.30}$$

To see why, Theorem 6.17 says that we first can shift the filter coefficients of H_0 so that it has this form (we then need to shift G_0 in the opposite direction). H_1, G_1 then can be defined by $\alpha = 1$ and $d = 0$. We will follow this convention for the orthonormal wavelets we look at.

The polynomials $Q^{(0)}$, $Q^{(1)}$, and $Q^{(2)}$ require no further action to obtain the factorization $f(e^{i\omega})f(e^{-i\omega}) = Q\left(\frac{1}{2}(1-\cos\omega)\right)$. The polynomial $Q^{(3)}$ in Equation (7.23) can be factored further as

$$\begin{aligned} Q^{(3)}\left(\frac{1}{2}(1-\cos\omega)\right) = \frac{5}{8}\frac{1}{3.0407}\frac{1}{7.1495}&(e^{-3i\omega} - 7.1029e^{-2i\omega} + 19.5014^{-i\omega} - 21.7391) \\ &\times (e^{3i\omega} - 7.1029e^{2i\omega} + 19.5014^{i\omega} - 21.7391), \end{aligned}$$

which gives that $f(e^{i\omega}) = \sqrt{\frac{5}{8}\frac{1}{3.0407}\frac{1}{7.1495}}(e^{3i\omega} - 7.1029e^{2i\omega} + 19.5014^{i\omega} - 21.7391)$. This factorization is not unique, however. This gives the frequency response $\lambda_{G_0}(\omega) = \left(\frac{1+e^{-i\omega}}{2}\right)^N f(e^{-i\omega})$ as

$$\frac{1}{2}(e^{-i\omega}+1)\sqrt{2}$$
$$\frac{1}{4}(e^{-i\omega}+1)^2\sqrt{\frac{1}{3.7321}}(e^{-i\omega}-3.7321)$$
$$\frac{1}{8}(e^{-i\omega}+1)^3\sqrt{\frac{3}{4}\frac{1}{9.4438}}(e^{-2i\omega}-5.4255e^{-i\omega}+9.4438)$$
$$\frac{1}{16}(e^{-i\omega}+1)^4\sqrt{\frac{5}{8}\frac{1}{3.0407}\frac{1}{7.1495}}(e^{-3i\omega}-7.1029e^{-2i\omega}+19.5014^{-i\omega}-21.7391),$$

which gives the filters

$$G_0=(H_0)^T=(\underline{\sqrt{2}/2},\sqrt{2}/2)$$
$$G_0=(H_0)^T=(-0.4830,\underline{-0.8365},-0.2241,0.1294)$$
$$G_0=(H_0)^T=(0.3327,0.8069,\underline{0.4599},-0.1350,-0.0854,0.0352)$$
$$G_0=(H_0)^T=(-0.2304,-0.7148,-0.6309,\underline{0.0280},0.1870,-0.0308,-0.0329,0.0106)$$

so that we get 2, 4, 6 and 8 filter coefficients in $G_0=(H_0)^T$. We see that the filter coefficients when $N=1$ are those of the Haar wavelet. The three next filters we have not seen before. The filter $G_1=(H_1)^T$ can be obtained from the relation $\lambda_{G_1}(\omega)=-\overline{\lambda_{G_0}(\omega+\pi)}$, i.e. by reversing the elements and adding an alternating sign, plus an extra minus sign, so that

$$G_1=(H_1)^T=(\sqrt{2}/2,\underline{-\sqrt{2}/2})$$
$$G_1=(H_1)^T=(0.1294,0.2241,\underline{-0.8365},0.4830)$$
$$G_1=(H_1)^T=(0.0352,0.0854,-0.1350,\underline{-0.4599},0.8069,-0.3327)$$
$$G_1=(H_1)^T=(0.0106,0.0329,-0.0308,-0.1870,\underline{0.0280},0.6309,-0.7148,0.2304).$$

Frequency responses are shown in Figure 7.4 for $N=1$ to $N=6$. It is seen that the frequency responses get increasingly flatter as N increases. The frequency responses are now complex, so their magnitudes are plotted.

Clearly these filters have lowpass characteristic. We also see that the highpass characteristics resemble the lowpass characteristics. We also see that the frequency response gets flatter near the high and low frequencies, as N increases. One can verify that this is the case also when N is increased further. The shapes for higher N are very similar to the frequency responses of those filters used in the MP3 standard (see Figure 3.12). One difference is that the support of the latter is concentrated on a smaller set of frequencies.

The way we have defined the filters, one can show in the same way as in the proof of Theorem 6.11 that, when all filters have $2N$ coefficients, $\phi=\tilde{\phi}$ has support $[-N+1,N]$, $\psi=\tilde{\psi}$ has support $[-N+1/2,N-1/2]$ (i.e. the support of ψ is symmetric about the origin). In particular we have that

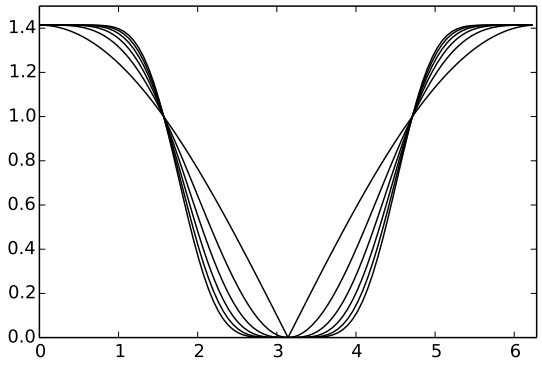

Figure 7.4: The magnitudes $|\lambda_{G_0}(\omega)| = |\lambda_{H_0}(\omega)|$ for the first orthonormal wavelets.

- for $N = 2$: $\text{supp}(\phi) = [-1, 2]$, $\text{supp}(\psi) = [-3/2, 3/2]$,
- for $N = 3$: $\text{supp}(\phi) = [-2, 3]$, $\text{supp}(\psi) = [-5/2, 5/2]$,
- for $N = 4$: $\text{supp}(\phi) = [-3, 4]$, $\text{supp}(\psi) = [-7/2, 7/2]$.

The scaling functions and mother wavelets are shown in Figure 7.5. All functions have been plotted over $[-4, 4]$, so that all these support sizes can be verified. Also here we have used the cascade algorithm to approximate the functions.

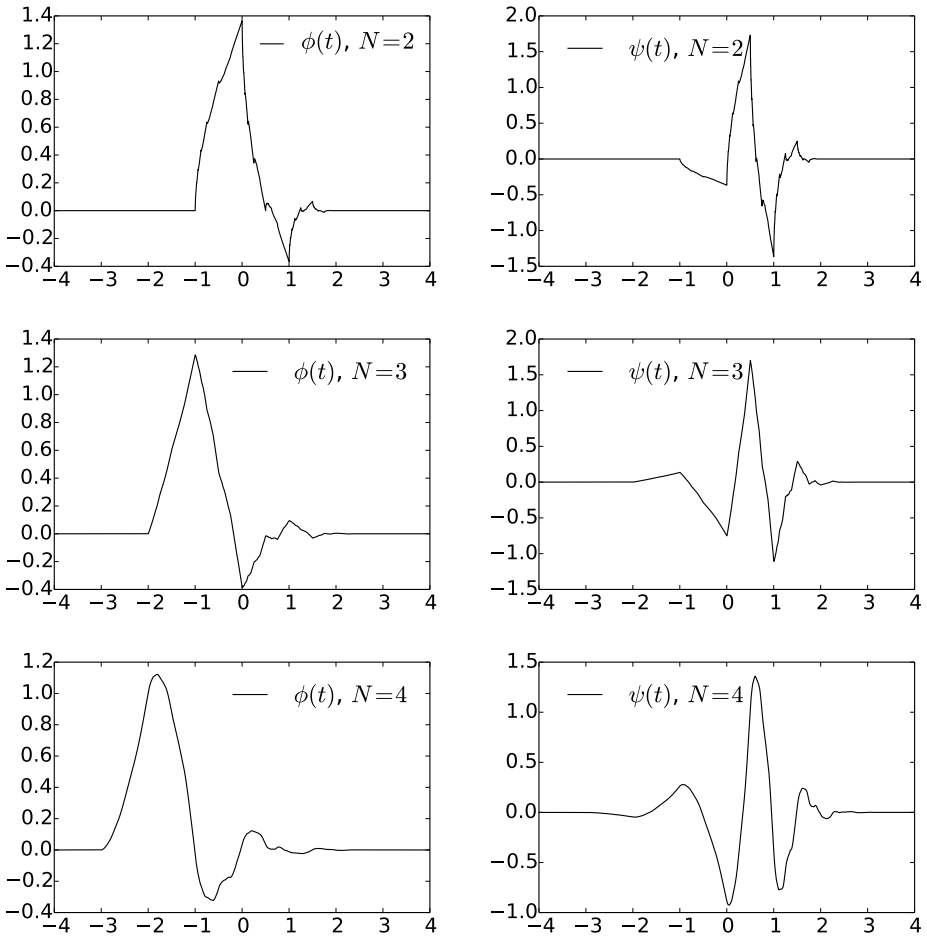

Figure 7.5: The scaling functions and mother wavelets for orthonormal wavelets with N vanishing moments, for different values of N.

7.8 Summary

We started the section by showing how filters from filter bank matrices can give rise to scaling functions and mother wavelets. We saw that we obtained dual function pairs in this way, which satisfied a mutual property called biorthogonality. We then saw how differentiable scaling functions or mother wavelets with vanishing moments could be constructed, and we saw how we could construct the simplest such. These could be found in terms of the frequency responses of the involved filters. Finally we studied some examples with applications to image compression.

For the wavelets we constructed in this chapter, we also plotted the corresponding scaling functions and mother wavelets (see figures 7.1, 7.3, 7.5). The

importance of these functions are that they are particularly suited for approximation of regular functions, and providing a compact representation of these functions which is localized in time. It seems difficult to guess that these strange shapes are connected to such approximation. Moreover, it may seem strange that, although these functions are useful, we can't write down exact expressions for them, and they are only approximated in terms of the Cascade Algorithm.

In the literature, the orthonormal wavelets with compact support we have constructed were first constructed in [8]. Biorthogonal wavelets were first constructed in [5].

Chapter 8

The polyphase representation and wavelets

In Chapter 6 we saw that we could express wavelet transformations and more general transformations in terms of filters. Through this we obtained intuition for what information the different parts of a wavelet transformation represent, in terms of lowpass and highpass filters. We also obtained some insight into the filters used in the transformation used in the MP3 standard. We expressed the DWT and IDWT implementations in terms of what we called kernel transformations, and these were directly obtained from the filters of the wavelet.

We have looked at many wavelets, however, but have only stated the kernel transformation for the Haar wavelet. In order to use these wavelets in sound and image processing, or in order to use the cascade algorithm to plot the corresponding scaling functions and mother wavelets, we need to make these kernel transformations. This will be one of the goals in this chapter. This will be connected to what we will call the *polyphase representation* of the wavelet. This representation will turn out to be useful for different reasons than the filter representation as well. First of all, with the polyphase representation, transformations can be viewed as block matrices where the blocks are filters. This allows us to prove results in a different way than for filter bank transforms, since we can prove results through block matrix manipulation. There will be two major results we will prove in this way.

First, in Section 8.1 we obtain a factorization of a wavelet transformation into sparse matrices, called elementary lifting matrices. We will show that this factorization reduces the number of arithmetic operations, and also enables us to compute the DWT in-place, in a similar way to how the FFT could be computed in-place after a bit-reversal. This is important: recall that we previously factored a filter into a product of smaller filters which is useful for efficient hardware implementations. But this did not address the fact that only every second component of the filters needs to be evaluated in the DWT, something any efficient implementation of the DWT should take into account.

The factorization into sparse matrices will be called the *lifting factorization*, and it will be clear from this factorization how the wavelet kernels and their duals can be implemented. We will also see how we can use the polyphase representation to prove the remaining parts of Theorem 6.17.

Secondly, in Section 8.3 we will use the polyphase representation to analyze how the forward and reverse filter bank transforms from the MP3 standard can be chosen in order for us to have perfect or near perfect reconstruction. Actually, we will obtain a factorization of the polyphase representation into block matrices also here, and the conditions we need to put on the prototype filters will be clear from this.

8.1 The polyphase representation and the lifting factorization

Let us start by defining the basic concepts in the polyphase representation.

Definition 8.1. *Polyphase components and representation.*

Assume that S is a matrix, and that M is a number. By the *polyphase components* of S we mean the matrices $S^{(i,j)}$ defined by $S^{(i,j)}_{r_1,r_2} = S_{i+r_1M,j+r_2M}$, i.e. the matrices obtained by taking every M'th component of S. By the *polyphase representation* of S we mean the block matrix with entries $S^{(i,j)}$.

The polyphase representation applies in particular for vectors. Since a vector $\boldsymbol{x}$ only has one column, we write $\boldsymbol{x}^{(p)}$ for its polyphase components.

Example 8.2. *A 6×6 MRA-matrix.*

Consider the 6×6 MRA-matrix

$$S = \begin{pmatrix} 2 & 3 & 0 & 0 & 0 & 1 \\ 4 & 5 & 6 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 & 0 \\ 0 & 0 & 4 & 5 & 6 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 \\ 6 & 0 & 0 & 0 & 4 & 5 \end{pmatrix}. \tag{8.1}$$

The polyphase components of S are

$$S^{(0,0)} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \qquad S^{(0,1)} = \begin{pmatrix} 3 & 0 & 1 \\ 1 & 3 & 0 \\ 0 & 1 & 3 \end{pmatrix}$$

$$S^{(1,0)} = \begin{pmatrix} 4 & 6 & 0 \\ 0 & 4 & 6 \\ 6 & 0 & 4 \end{pmatrix} \qquad S^{(1,1)} = \begin{pmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{pmatrix}$$

We will mainly be concerned with polyphase representations of MRA matrices. For such matrices we have the following result (this result can be stated more generally for any filter bank transform).

Theorem 8.3. *Similarity.*

When S is an MRA-matrix, the polyphase components $S^{(i,j)}$ are filters (in general different from the filters considered in Chapter 6), i.e. the polyphase representation is a 2×2-block matrix where all blocks are filters. Also, S is similar to its polyphase representation, through a permutation matrix P which places the even-indexed elements first.

To see why, note that when P is the permutation matrix defined above, then PS consists of S with the even-indexed rows grouped first, and since also $SP^T = (PS^T)^T$, SP^T groups the even-indexed columns first. From these observations it is clear that PSP^T is the polyphase representation of S, so that S is similar to its polyphase representation.

We also have the following result on the polyphase representation. This result is easily proved from manipulation with block matrices, and is therefore left to the reader.

Theorem 8.4. *Products and transpose.*

Let A and B be (forward or reverse) filter bank transforms, and denote the corresponding polyphase components by $A^{(i,j)}$, $B^{(i,j)}$. The following hold

- $C = AB$ is also a filter bank transform, with polyphase components $C^{(i,j)} = \sum_k A^{(i,k)} B^{(k,j)}$.
- A^T is also a filter bank transform, with polyphase components $((A^T)^{(i,j)})_{k,l} = (A^{(j,i)})_{l,k}$.

Also, the polyphase components of the identity matrix is the $M \times M$-block matrix with the identity matrix on the diagonal, and $\mathbf{0}$ elsewhere.

To see an application of the polyphase representation, let us prove the final ingredient of Theorem 6.17. We need to prove the following:

Theorem 8.5. *Criteria for perfect reconstruction.*

For any set of FIR filters H_0, H_1, G_0, G_1 which give perfect reconstruction, there exist $\alpha \in \mathbb{R}$ and $d \in \mathbb{Z}$ so that

$$\lambda_{H_1}(\omega) = \alpha^{-1} e^{-2id\omega} \lambda_{G_0}(\omega + \pi) \tag{8.2}$$

$$\lambda_{G_1}(\omega) = \alpha e^{2id\omega} \lambda_{H_0}(\omega + \pi). \tag{8.3}$$

Proof. Let $H^{(i,j)}$ be the polyphase components of H, $G^{(i,j)}$ the polyphase components of G. $GH = I$ means that

$$\begin{pmatrix} G^{(0,0)} & G^{(0,1)} \\ G^{(1,0)} & G^{(1,1)} \end{pmatrix} \begin{pmatrix} H^{(0,0)} & H^{(0,1)} \\ H^{(1,0)} & H^{(1,1)} \end{pmatrix} = \begin{pmatrix} I & \mathbf{0} \\ \mathbf{0} & I \end{pmatrix}.$$

If we here multiply with $\begin{pmatrix} G^{(1,1)} & -G^{(0,1)} \\ -G^{(1,0)} & G^{(0,0)} \end{pmatrix}$ on both sides to the left, or with $\begin{pmatrix} H^{(1,1)} & -H^{(0,1)} \\ -H^{(1,0)} & H^{(0,0)} \end{pmatrix}$ on both sides to the right, we get

$$\begin{pmatrix} G^{(1,1)} & -G^{(0,1)} \\ -G^{(1,0)} & G^{(0,0)} \end{pmatrix} = \begin{pmatrix} (G^{(0,0)}G^{(1,1)} - G^{(1,0)}G^{(0,1)})H^{(0,0)} & (G^{(0,0)}G^{(1,1)} - G^{(1,0)}G^{(0,1)})H^{(0,1)} \\ (G^{(0,0)}G^{(1,1)} - G^{(1,0)}G^{(0,1)})H^{(1,0)} & (G^{(0,0)}G^{(1,1)} - G^{(1,0)}G^{(0,1)})H^{(1,1)} \end{pmatrix}$$
$$\begin{pmatrix} H^{(1,1)} & -H^{(0,1)} \\ -H^{(1,0)} & H^{(0,0)} \end{pmatrix} = \begin{pmatrix} (H^{(0,0)}H^{(1,1)} - H^{(1,0)}H^{(0,1)})G^{(0,0)} & (H^{(0,0)}H^{(1,1)} - H^{(1,0)}H^{(0,1)})G^{(0,1)} \\ (H^{(0,0)}H^{(1,1)} - H^{(1,0)}H^{(0,1)})G^{(1,0)} & (H^{(0,0)}H^{(1,1)} - H^{(1,0)}H^{(0,1)})G^{(1,1)} \end{pmatrix}$$

Now since $G^{(0,0)}G^{(1,1)} - G^{(1,0)}G^{(0,1)}$ and $H^{(0,0)}H^{(1,1)} - H^{(1,0)}H^{(0,1)}$ also are circulant Toeplitz matrices, the expressions above give that

$$\begin{aligned} l(H^{(0,0)}) \leq l(G^{(1,1)}) \leq l(H^{(0,0)}) \\ l(H^{(0,1)}) \leq l(G^{(0,1)}) \leq l(H^{(0,1)}) \\ l(H^{(1,0)}) \leq l(G^{(1,0)}) \leq l(H^{(1,0)}) \end{aligned}$$

so that we must have equality here, and with both

$$G^{(0,0)}G^{(1,1)} - G^{(1,0)}G^{(0,1)} \text{ and } H^{(0,0)}H^{(1,1)} - H^{(1,0)}H^{(0,1)}$$

having only one nonzero diagonal. In particular we can define the diagonal matrix $D = H^{(0,0)}H^{(1,1)} - H^{(0,1)}H^{(1,0)} = \alpha^{-1}E_d$ (for some α, d), and we have that

$$\begin{pmatrix} G^{(0,0)} & G^{(0,1)} \\ G^{(1,0)} & G^{(1,1)} \end{pmatrix} = \begin{pmatrix} \alpha E_{-d}H^{(1,1)} & -\alpha E_{-d}H^{(0,1)} \\ -\alpha E_{-d}H^{(1,0)} & \alpha E_{-d}H^{(0,0)} \end{pmatrix}.$$

The first columns here state a relation between G_0 and H_1, while the second columns state a relation between G_1 and H_0. It is straightforward to show that these relations imply equation (8.2)-(8.3). The details for this can be found in Exercise 8.1. □

In the following we will find factorizations of 2×2-block matrices where the blocks are filters, into simpler such matrices. The importance of Theorem 8.3 is then that MRA-matrices can be written as a product of simpler MRA matrices. These simpler MRA matrices will be called elementary lifting matrices, and will be of the following type.

Definition 8.6. *Elementary lifting matrices.*

A matrix on the form

$$\begin{pmatrix} I & S \\ 0 & I \end{pmatrix}$$

where S is a filter is called an *elementary lifting matrix of even type*. A matrix on the form

$$\begin{pmatrix} I & 0 \\ S & I \end{pmatrix}$$

is called an *elementary lifting matrix of odd type*.

The following are the most useful properties of elementary lifting matrices:

Lemma 8.7. *Lifting lemma.*
The following hold:

$$\begin{pmatrix} I & S \\ 0 & I \end{pmatrix}^T = \begin{pmatrix} I & 0 \\ S^T & I \end{pmatrix}, \text{ and } \begin{pmatrix} I & 0 \\ S & I \end{pmatrix}^T = \begin{pmatrix} I & S^T \\ 0 & I \end{pmatrix},$$

$$\begin{pmatrix} I & S_1 \\ 0 & I \end{pmatrix}\begin{pmatrix} I & S_2 \\ 0 & I \end{pmatrix} = \begin{pmatrix} I & S_1+S_2 \\ 0 & I \end{pmatrix}, \text{ and } \begin{pmatrix} I & 0 \\ S_1 & I \end{pmatrix}\begin{pmatrix} I & 0 \\ S_2 & I \end{pmatrix} = \begin{pmatrix} I & 0 \\ S_1+S_2 & I \end{pmatrix},$$

$$\begin{pmatrix} I & S \\ 0 & I \end{pmatrix}^{-1} = \begin{pmatrix} I & -S \\ 0 & I \end{pmatrix}, \text{ and } \begin{pmatrix} I & 0 \\ S & I \end{pmatrix}^{-1} = \begin{pmatrix} I & 0 \\ -S & I \end{pmatrix}$$

These statements follow directly from Theorem 8.4. Due to Property 2, one can assume that odd and even types of lifting matrices appear in alternating order, since matrices of the same type can be grouped together. The following result states why elementary lifting matrices can be used to factorize general MRA-matrices:

Theorem 8.8. *Multiplying.*
Any invertible matrix on the form $S = \begin{pmatrix} S^{(0,0)} & S^{(0,1)} \\ S^{(1,0)} & S^{(1,1)} \end{pmatrix}$, where the $S^{(i,j)}$ are filters with a finite numer of filter coefficients, can be written on the form

$$\Lambda_1 \cdots \Lambda_n \begin{pmatrix} \alpha_0 E_p & 0 \\ 0 & \alpha_1 E_q \end{pmatrix}, \tag{8.4}$$

where Λ_i are elementary lifting matrices, p, q are integers, α_0, α_1 are nonzero scalars, and E_p, E_q are time delay filters. The inverse is given by

$$\begin{pmatrix} \alpha_0^{-1} E_{-p} & 0 \\ 0 & \alpha_1^{-1} E_{-q} \end{pmatrix} (\Lambda_n)^{-1} \cdots (\Lambda_1)^{-1}. \tag{8.5}$$

Note that $(\Lambda_i)^{-1}$ can be computed with the help of Property 3 of Lemma 8.7.

Proof. The proof will use the concept of the length of a filter, as defined in Definition 3.5. Let $S = \begin{pmatrix} S^{(0,0)} & S^{(0,1)} \\ S^{(1,0)} & S^{(1,1)} \end{pmatrix}$ be an arbitrary invertible matrix. We will incrementally find an elementary lifting matrix Λ_i with filter S_i in the lower left or upper right corner so that $\Lambda_i S$ has filters of lower length in the first column. Assume first that $l(S^{(0,0)}) \geq l(S^{(1,0)})$, where $l(S)$ is the length of a filter as given by Definition 3.5. If Λ_i is of even type, then the first column in $\Lambda_i S$ is

$$\begin{pmatrix} I & S_i \\ 0 & I \end{pmatrix}\begin{pmatrix} S^{(0,0)} \\ S^{(1,0)} \end{pmatrix} = \begin{pmatrix} S^{(0,0)} + S_i S^{(1,0)} \\ S^{(1,0)} \end{pmatrix}. \tag{8.6}$$

S_i can now be chosen so that $l(S^{(0,0)} + S_i S^{(1,0)}) < l(S^{(1,0)})$. To see how, recall that we in Section 3.1 stated that multiplying filters corresponds to multiplying polynomials. S_i can thus be found from polynomial division with remainder: when we divide $S^{(0,0)}$ by $S^{(1,0)}$, we actually find polynomials S_i and P with $l(P) < l(S^{(1,0)})$ so that $S^{(0,0)} = S_i S^{(1,0)} + P$, so that the length of $P = S^{(0,0)} - S_i S^{(1,0)}$ is less than $l(S^{(1,0)})$. The same can be said if Λ_i is of odd type, in which case the first and second components are simply swapped. This procedure can be continued until we arrive at a product

$$\Lambda_n \cdots \Lambda_1 S$$

where either the first or the second component in the first column is 0. If the first component in the first column is 0, the identity

$$\begin{pmatrix} I & 0 \\ -I & I \end{pmatrix} \begin{pmatrix} I & I \\ 0 & I \end{pmatrix} \begin{pmatrix} 0 & X \\ Y & Z \end{pmatrix} = \begin{pmatrix} Y & X+Z \\ 0 & -X \end{pmatrix}$$

explains that we can bring the matrix to a form where the second element in the first column is zero instead, with the help of the additional lifting matrices $\Lambda_{n+1} = \begin{pmatrix} I\&I \\ 0\&I \end{pmatrix}$, and $\Lambda_{n+2} = \begin{pmatrix} I\&0 \\ -I\&I \end{pmatrix}$, so that we always can assume that the second element in the first column is 0, i.e.

$$\Lambda_n \cdots \Lambda_1 S = \begin{pmatrix} P & Q \\ 0 & R \end{pmatrix},$$

for some matrices P, Q, R. From the proof of Theorem 6.17 we will see that in order for S to be invertible, we must have that $S^{(0,0)} S^{(1,1)} - S^{(0,1)} S^{(1,0)} = -\alpha^{-1} E_d$ for some nonzero scalar α and integer d. Since

$$\begin{pmatrix} P & Q \\ 0 & R \end{pmatrix}$$

is also invertible, we must thus have that PR must be on the form αE_n. When the filters have a finite number of filter coefficients, the only possibility for this to happen is when $P = \alpha_0 E_p$ and $R = \alpha_1 E_q$ for some p, q, α_0, α_1. Using this, and also isolating S on one side, we obtain that

$$S = (\Lambda_1)^{-1} \cdots (\Lambda_n)^{-1} \begin{pmatrix} \alpha_0 E_p & Q \\ 0 & \alpha_1 E_q \end{pmatrix}, \tag{8.7}$$

Noting that

$$\begin{pmatrix} \alpha_0 E_p & Q \\ 0 & \alpha_1 E_q \end{pmatrix} = \begin{pmatrix} 1 & \frac{1}{\alpha_1} E_{-q} Q \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha_0 E_p & 0 \\ 0 & \alpha_1 E_q \end{pmatrix},$$

we can rewrite Equation (8.7) as

$$S = (\Lambda_1)^{-1} \cdots (\Lambda_n)^{-1} \begin{pmatrix} 1 & \frac{1}{\alpha_1} E_{-q} Q \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha_0 E_p & 0 \\ 0 & \alpha_1 E_q \end{pmatrix},$$

which is a lifting factorization of the form we wanted to arrive at. The last matrix in the lifting factorization is not really a lifting matrix, but it too can easily be inverted, so that we arrive at Equation (8.5). This completes the proof. □

Factorizations on the form given by Equation (8.4) will be called *lifting factorizations*. Assume that we have applied Theorem 8.8 in order to get a factorization of the polyphase representation of the DWT kernel of the form

$$\Lambda_n \cdots \Lambda_2 \Lambda_1 H = \begin{pmatrix} \alpha & \mathbf{0} \\ \mathbf{0} & \beta \end{pmatrix}. \tag{8.8}$$

Theorem 8.7 then immediately gives us the following factorizations.

$$H = (\Lambda_1)^{-1}(\Lambda_2)^{-1} \cdots (\Lambda_n)^{-1} \begin{pmatrix} \alpha & \mathbf{0} \\ \mathbf{0} & \beta \end{pmatrix} \tag{8.9}$$

$$G = \begin{pmatrix} 1/\alpha & \mathbf{0} \\ \mathbf{0} & 1/\beta \end{pmatrix} \Lambda_n \cdots \Lambda_2 \Lambda_1 \tag{8.10}$$

$$H^T = \begin{pmatrix} \alpha & \mathbf{0} \\ \mathbf{0} & \beta \end{pmatrix} ((\Lambda_n)^{-1})^T ((\Lambda_{n-1})^{-1})^T \cdots ((\Lambda_1)^{-1})^T \tag{8.11}$$

$$G^T = (\Lambda_1)^T (\Lambda_2)^T \cdots (\Lambda_n)^T \begin{pmatrix} 1/\alpha & \mathbf{0} \\ \mathbf{0} & 1/\beta \end{pmatrix}. \tag{8.12}$$

Since H^T and G^T are the kernel transformations of the dual IDWT and the dual DWT, respectively, these formulas give us recipes for computing the DWT, IDWT, dual IDWT, and the dual DWT, respectively. All in all, everything can be computed by combining elementary lifting steps.

In practice, one starts with a given wavelet with certain proved properties such as the ones from Chapter 7, and applies an algorithm to obtain a lifting factorization of the polyphase representation of the kernels. The algorihtm can easily be written down from the proof of Theorem 8.8. The lifting factorization is far from unique, and the algorithm only gives one of them.

It is desirable for an implementation to obtain a lifting factorization where the lifting steps are as simple as possible. Let us restrict to the case of wavelets with symmetric filters, since the wavelets used in most applications are symmetric. In particular this means that $S^{(0,0)}$ is a symmetric matrix, and that $S^{(1,0)}$ is symmetric about $-1/2$ (see Exercise 8.7).

Assume that we in the proof of Theorem 8.8 add an elementary lifting of even type. At this step we then compute $S^{(0,0)} + S_i S^{(1,0)}$ in the first entry of the first column. Since $S^{(0,0)}$ is now assumed symmetric, $S_i S^{(1,0)}$ must also be symmetric in order for the length to be reduced. And since the filter coefficients of $S^{(1,0)}$ are assumed symmetric about $-1/2$, S_i must be chosen with symmetry around $1/2$.

For most of our wavelets we will consider in the following examples it will turn out the filters in the first column differ in the number of filter coefficients by 1 at all steps. When this is the case, we can choose a filter of length 2 to

reduce the length by 2, so that the S_i in an even lifting step can be chosen on the form $S_i = \lambda_i\{\underline{1}, 1\}$. Similarly, for an odd lifting step, S_i can be chosen on the form $S_i = \lambda_i\{1, \underline{1}\}$. Let us summarize this as follows:

Theorem 8.9. *Differing by* 1.

When the filters in a wavelet are symmetric and the lengths of the filters in the first column differ by 1 at all steps in the lifting factorization, the lifting steps of even and odd type take the simplified form

$$\begin{pmatrix} I & \lambda_i\{\underline{1}, 1\} \\ 0 & I \end{pmatrix} \text{ and } \begin{pmatrix} I & 0 \\ \lambda_i\{1, \underline{1}\} & I \end{pmatrix},$$

respectively.

The lifting steps mentioned in this theorem are quickly computed due to their simple structure. The corresponding MRA matrices are

$$\begin{pmatrix} 1 & \lambda & 0 & 0 & \cdots & 0 & 0 & \lambda \\ 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & \lambda & 1 & \lambda & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \lambda & 1 & \lambda \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \lambda & 1 & \lambda & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ \lambda & 0 & 0 & 0 & \cdots & 0 & \lambda & 1 \end{pmatrix},$$

respectively, or

$$\begin{pmatrix} 1 & 2\lambda & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & \lambda & 1 & \lambda & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \lambda & 1 & \lambda \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \lambda & 1 & \lambda & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 2\lambda & 1 \end{pmatrix} \tag{8.13}$$

if we use symmetric extensions as defined by Definition 5.42 (we have used Theorem 5.43). Each lifting step leaves every second element unchanged, while for the remaining elements, we simply add the two neighbours. Clearly these computations can be computed in-place, without the need for extra memory allocations. From this it is also clear how we can compute the entire DWT/IDWT in-place. We simply avoid the reorganizing into the $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$-basis until after all the lifting steps. After the application of the matrices above, we have coordinates in the $\mathcal{C}_m$-basis. Here only the coordinates with indices $(0, 2, 4, \ldots)$ need to be further transformed, so the next step in the algorithm should work directly on these. After the next step only the coordinates with indices $(0, 4, 8, \ldots)$ need to be further transformed, and so on. From this it is clear that

- the $\boldsymbol{\psi}_{m-k}$ coordinates are found at indices $2^{k-1} + r2^k$, i.e. the last k bits are 1 followed by $k-1$ zeros.
- the $\boldsymbol{\phi}_0$ coordinates are found at indices $r2^m$, i.e. the last m bits are 0.

If we place the last k bits of the $\boldsymbol{\psi}_{m-k}$-coordinates in front in reverse order, and the the last m bits of the $\boldsymbol{\phi}_0$-coordinates in front, the coordinates have the same order as in the $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$-basis. This is also called a partial bit-reverse, and is related to the bit-reversal performed in the FFT algorithm.

Clearly, these lifting steps are also MRA-matrices with symmetric filters, so that our procedure factorizes an MRA-matrix with symmetric filters into simpler MRA-matrices which also have symmetric filters.

8.1.1 Reduction in the number of arithmetic operations with the lifting factorization

The number of arithmetic operations needed to apply matrices on the form stated in Equation (8.13) is easily computed. The number of multiplications is $N/2$ if symmetry is exploited as in Observation 4.21 (N if symmetry is not exploited). Similarly, the number of additions is N. Let K be the total number of filter coefficients in H_0, H_1. In the following we will see that each lifting step can be chosen to reduce the number of filter coefficients in the MRA matrix by 4, so that a total number of $K/4$ lifting steps are required. Thus, a total number of $KN/8$ ($KN/4$) multiplications, and $KN/4$ additions are required when a lifting factorization is used. In comparison, a direct implementation would require $KN/4$ ($KN/2$) multiplications, and $KN/2$ additions. For the examples we will consider, we therefore have the following result.

Theorem 8.10. *Reducing arithmetic operations.*

The lifting factorization approximately halves the number of additions and multiplications needed, when compared with a direct implementation (regardless of whether symmetry is exploited or not).

Exercise 8.1: The frequency responses of the polyphase components

Let H and G be MRA-matrices for a DWT/IDWT, with corresponding filters H_0, H_1, G_0, G_1, and polyphase components $H^{(i,j)}$, $G^{(i,j)}$.

a) Show that

$$\begin{aligned}
\lambda_{H_0}(\omega) &= \lambda_{H^{(0,0)}}(2\omega) + e^{i\omega}\lambda_{H^{(0,1)}}(2\omega) \\
\lambda_{H_1}(\omega) &= \lambda_{H^{(1,1)}}(2\omega) + e^{-i\omega}\lambda_{H^{(1,0)}}(2\omega) \\
\lambda_{G_0}(\omega) &= \lambda_{G^{(0,0)}}(2\omega) + e^{-i\omega}\lambda_{G^{(1,0)}}(2\omega) \\
\lambda_{G_1}(\omega) &= \lambda_{G^{(1,1)}}(2\omega) + e^{i\omega}\lambda_{G^{(0,1)}}(2\omega).
\end{aligned}$$

b) In the proof of the last part of Theorem 6.17, we defered the last part, namely that equations (7.29) in the compendium-(8.3) in the compendium follow from

$$\begin{pmatrix} G^{(0,0)} & G^{(0,1)} \\ G^{(1,0)} & G^{(1,1)} \end{pmatrix} = \begin{pmatrix} \alpha E_{-d} H^{(1,1)} & -\alpha E_{-d} H^{(0,1)} \\ -\alpha E_{-d} H^{(1,0)} & \alpha E_{-d} H^{(0,0)} \end{pmatrix}.$$

Prove this based on the result from a).

Exercise 8.2: Finding new filters

Let S be a filter. Show that

a)

$$G \begin{pmatrix} I & \mathbf{0} \\ S & I \end{pmatrix}$$

is an MRA matrix with filters $\tilde{G_0}, G_1$, where

$$\lambda_{\tilde{G_0}}(\omega) = \lambda_{G_0}(\omega) + \lambda_S(2\omega) e^{-i\omega} \lambda_{G_1}(\omega),$$

b)

$$G \begin{pmatrix} I & S \\ \mathbf{0} & I \end{pmatrix}$$

is an MRA matrix with filters $G_0, \tilde{G_1}$, where

$$\lambda_{\tilde{G_1}}(\omega) = \lambda_{G_1}(\omega) + \lambda_S(2\omega) e^{i\omega} \lambda_{G_0}(\omega),$$

c)

$$\begin{pmatrix} I & \mathbf{0} \\ S & I \end{pmatrix} H$$

is an MRA-matrix with filters $H_0, \tilde{H_1}$, where

$$\lambda_{\tilde{H_1}}(\omega) = \lambda_{H_1}(\omega) + \lambda_S(2\omega) e^{-i\omega} \lambda_{H_0}(\omega).$$

d)

$$\begin{pmatrix} I & S \\ \mathbf{0} & I \end{pmatrix} H$$

is an MRA-matrix with filters $\tilde{H}_0, H_1$, where

$$\lambda_{\tilde{H_0}}(\omega) = \lambda_{H_0}(\omega) + \lambda_S(2\omega) e^{i\omega} \lambda_{H_1}(\omega).$$

In summary, this exercise shows that one can think of the steps in the lifting factorization as altering one of the filters of an MRA-matrix in alternating order.

Exercise 8.3: Relating to the polyphase components

Show that S is a filter of length kM if and only if the entries $\{S^{i,j}\}_{i,j=0}^{M-1}$ in the polyphase representation of S satisfy $S^{(i+r) \bmod M,(j+r) \bmod M} = S_{i,j}$. In other words, S is a filter if and only if the polyphase representation of S is a "block-circulant Toeplitz matrix". This implies a fact that we will use: GH is a filter (and thus provides alias cancellation) if blocks in the polyphase representations repeat cyclically as in a Toeplitz matrix (in particular when the matrix is block-diagonal with the same block repeating on the diagonal).

Exercise 8.4: QMF filter banks

Recall from Definition 6.20 that we defined a classical QMF filter bank as one where $M = 2$, $G_0 = H_0$, $G_1 = H_1$, and $\lambda_{H_1}(\omega) = \lambda_{H_0}(\omega + \pi)$. Show that the forward and reverse filter bank transforms of a classical QMF filter bank take the form

$$H = G = \begin{pmatrix} A & -B \\ B & A \end{pmatrix}$$

Exercise 8.5: Alternative QMF filter banks

Recall from Definition 6.21 that we defined an alternative QMF filter bank as one where $M = 2$, $G_0 = (H_0)^T$, $G_1 = (H_1)^T$, and $\lambda_{H_1}(\omega) = \overline{\lambda_{H_0}(\omega + \pi)}$. Show that the forward and reverse filter bank transforms of an alternative QMF filter bank take the form.

$$H = \begin{pmatrix} A^T & B^T \\ -B & A \end{pmatrix} \qquad G = \begin{pmatrix} A & -B^T \\ B & A^T \end{pmatrix} = \begin{pmatrix} A^T & B^T \\ -B & A \end{pmatrix}^T.$$

Exercise 8.6: Alternative QMF filter banks with additional sign

Consider alternative QMF filter banks where we take in an additional sign, so that $\lambda_{H_1}(\omega) = -\overline{\lambda_{H_0}(\omega + \pi)}$ (the Haar wavelet was an example of such a filter bank). Show that the forward and reverse filter bank transforms now take the form

$$H = \begin{pmatrix} A^T & B^T \\ B & -A \end{pmatrix} \qquad G = \begin{pmatrix} A & B^T \\ B & -A^T \end{pmatrix} = \begin{pmatrix} A^T & B^T \\ B & -A \end{pmatrix}^T.$$

It is straightforward to check that also these satisfy the alias cancellation condition, and that the perfect reconstruction condition also here takes the form $|\lambda_{H_0}(\omega)|^2 + |\lambda_{H_0}(\omega + \pi)|^2 = 2$.

8.2 Examples of lifting factorizations

We have seen that the polyphase representations of wavelet kernels can be factored into a product of elementary lifting matrices. In this section we will compute the exact factorizations for the wavelets we have considered. In the exercises we will then complete the implementations, so that we can make actual experiments, such as listening to the low-resolution approximations in sound, or using the cascade algorithm to plot scaling functions and mother wavelets. We will omit the Haar wavelet. One can easily write down a lifting factorization for this as well, but there is little to save in this factorization when compared to the direct form of this we already have considered.

First we will consider the two piecewise linear wavelets we have looked at. It turns out that their lifting factorizations can be obtained in a direct way by considering the polyphase representations as a change of coordinates. To see how, we first define

$$\mathcal{D}_m = \{\phi_{m,0}, \phi_{m,2}, \phi_{m,4} \ldots, \phi_{m,1}, \phi_{m,3}, \phi_{m,5}, \ldots\}, \tag{8.14}$$

$P_{\mathcal{D}_m \leftarrow \boldsymbol{\phi}_m}$ is clearly the permutation matrix P used in the similarity between a matrix and its polyphase representation. Let now H and G be the kernel transformations of a wavelet. The polyphase representation of H is

$$PHP^T = P_{\mathcal{D}_m \leftarrow \boldsymbol{\phi}_m} P_{\mathcal{C}_m \leftarrow \boldsymbol{\phi}_m} P_{\boldsymbol{\phi}_m \leftarrow \mathcal{D}_m} = P_{(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1) \leftarrow \boldsymbol{\phi}_m} P_{\boldsymbol{\phi}_m \leftarrow \mathcal{D}_m} = P_{(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1) \leftarrow \mathcal{D}_m}.$$

Taking inverses here we obtain that $PGP^T = P_{\mathcal{D}_m \leftarrow (\boldsymbol{\phi}_1, \boldsymbol{\psi}_1)}$. We therefore have the following result:

Theorem 8.11. *The polyphase representation.*

The polyphase representation of H equals the change of coordinates matrix $P_{(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1) \leftarrow \mathcal{D}_m}$, and the polyphase representation of G equals the change of coordinates matrix $P_{\mathcal{D}_m \leftarrow (\boldsymbol{\phi}_1, \boldsymbol{\psi}_1)}$.

Example 8.12. *Lifting factorization of the piecewise linear wavelet.*

Let us consider the piecewise linear wavelet from Section 5.4, for which we found that the change of coordinate matrix G was given by Equation (6.1). In the four different polyphase components of G, let us underline the corresponding elements:

$$\frac{1}{\sqrt{2}} \begin{pmatrix} \underline{1} & 0 \\ 1/2 & 1 \\ \underline{0} & 0 \\ \vdots & \vdots \\ \underline{0} & 0 \\ 1/2 & 0 \end{pmatrix}, \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & \underline{0} \\ 1/2 & 1 \\ 0 & \underline{0} \\ \vdots & \vdots \\ 0 & \underline{0} \\ 1/2 & 0 \end{pmatrix}, \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ \underline{1/2} & 1 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ \underline{1/2} & 0 \end{pmatrix}, \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 1/2 & \underline{1} \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 1/2 & \underline{0} \end{pmatrix}. \tag{8.15}$$

we get the following:

- The upper left polyphase component $G^{(0,0)}$ is $\frac{1}{\sqrt{2}}I$.
- The upper right polyphase component $G^{(0,1)}$ is $\mathbf{0}$.
- The lower left polyphase component $G^{(1,0)}$ is $\frac{1}{\sqrt{2}}S_1$, where S_1 is the filter $\{1/2, \underline{1/2}\}$.
- The lower right polyphase component $G^{(1,1)}$ is $\frac{1}{\sqrt{2}}I$.

In other words, the polyphase representation of G is $\frac{1}{\sqrt{2}}\begin{pmatrix} I & \mathbf{0} \\ \frac{1}{2}\{1,\underline{1}\} & I \end{pmatrix}$. Due to Theorem 8.7, the polyphase representation of H is $\sqrt{2}\begin{pmatrix} I & \mathbf{0} \\ -\frac{1}{2}\{1,\underline{1}\} & I \end{pmatrix}$ We can summarize that the polyphase representations of the kernels H and G for the piecewise linear wavelet are

$$\sqrt{2}\begin{pmatrix} I & \mathbf{0} \\ -\frac{1}{2}\{1,\underline{1}\} & I \end{pmatrix} \text{ and } \frac{1}{\sqrt{2}}\begin{pmatrix} I & \mathbf{0} \\ \frac{1}{2}\{1,\underline{1}\} & I \end{pmatrix}, \tag{8.16}$$

respectively.

Example 8.13. *Lifting factorization of the alternative piecewise linear wavelet.*

Let us now consider the alternative piecewise linear wavelet. In this case, Equation (6.3) shows that $P_{\mathcal{D}_1 \leftarrow (\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)}$ (the polyphase representation of H) is not on the form

$$\begin{pmatrix} I & \mathbf{0} \\ S_1 & I \end{pmatrix}$$

for some filter S_1, since there is more than one element in every column. Recall, however, that the alternative piecewise linear wavelet was obtained by constructing a new mother wavelet $\hat{\psi}$ from the old ψ. $\hat{\psi}$ is defined in Section 5.5 by Equation (5.38), which said that

$$\hat{\psi}(t) = \psi(t) - \frac{1}{4}\big(\phi_{0,0}(t) + \phi_{0,1}(t)\big).$$

From this equation it is clear that

$$P_{(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1) \leftarrow (\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)} = \begin{pmatrix} I & S_2 \\ 0 & I \end{pmatrix},$$

where $S_2 = -\frac{1}{4}\{\underline{1}, 1\}$. We can now write the polyphase representation of G as

$$P_{\mathcal{D}_1 \leftarrow (\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)} = P_{\mathcal{D}_1 \leftarrow (\boldsymbol{\phi}_1, \boldsymbol{\psi}_1)} P_{(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1) \leftarrow (\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)}. = \frac{1}{\sqrt{2}}\begin{pmatrix} I & \mathbf{0} \\ \frac{1}{2}\{1,\underline{1}\} & I \end{pmatrix}\begin{pmatrix} I & -\frac{1}{4}\{\underline{1}, 1\} \\ \mathbf{0} & I \end{pmatrix}.$$

In other words, also here the same type of matrix could be used to express the change of coordinates. This matrix is also easily invertible, so that the polyphase representation of H is

$$\sqrt{2}\begin{pmatrix} I & \frac{1}{4}\{\underline{1},1\} \\ \mathbf{0} & I \end{pmatrix}\begin{pmatrix} I & \mathbf{0} \\ -\frac{1}{2}\{1,\underline{1}\} & I \end{pmatrix}.$$

In this case we required one additional lifting step. We can thus conclude that the polyphase representations of the kernels H and G for the alternative piecewise linear wavelet are

$$\sqrt{2}\begin{pmatrix} I & \frac{1}{4}\{\underline{1},1\} \\ \mathbf{0} & I \end{pmatrix}\begin{pmatrix} I & \mathbf{0} \\ -\frac{1}{2}\{1,\underline{1}\} & I \end{pmatrix} \text{ and } \frac{1}{\sqrt{2}}\begin{pmatrix} I & \mathbf{0} \\ \frac{1}{2}\{1,\underline{1}\} & I \end{pmatrix}\begin{pmatrix} I & -\frac{1}{4}\{\underline{1},1\} \\ \mathbf{0} & I \end{pmatrix}, \tag{8.17}$$

respectively.

Example 8.14. *Lifting factorization of the Spline 5/3 wavelet.*

Let us consider the Spline 5/3 wavelet, which we defined in Example 7.16. Let us start by looking at, and we recall that

$$H_0 = \left\{-\frac{1}{4}, \frac{1}{2}, \underline{\frac{3}{2}}, \frac{1}{2}, -\frac{1}{4}\right\} \qquad H_1 = \left\{-\frac{1}{4}, \underline{\frac{1}{2}}, -\frac{1}{4}\right\}.$$

from which we see that the polyphase components of H are

$$\begin{pmatrix} H^{(0,0)} & H^{(0,1)} \\ H^{(1,0)} & H^{(1,1)} \end{pmatrix} = \begin{pmatrix} \{-\frac{1}{4}, \underline{\frac{3}{2}}, -\frac{1}{4}\} & \frac{1}{2}\{\underline{1},1\} \\ -\frac{1}{4}\{1,\underline{1}\} & \frac{1}{2}I \end{pmatrix}$$

We see here that the upper filter has most filter coefficients in the first column, so that we must start with an elementary lifting of even type. We need to find a filter S_1 so that $S_1\{-1/4, \underline{-1/4}\} + \{-1/4, \underline{3/2}, -1/4\}$ has fewer filter coefficients than $\{-1/4, \underline{3/2}, -1/4\}$. It is clear that we can choose $S_1 = \{\underline{-1}, -1\}$, and that

$$\Lambda_1 H = \begin{pmatrix} I & \{\underline{-1}, -1\} \\ \mathbf{0} & I \end{pmatrix}\begin{pmatrix} \{-\frac{1}{4}, \underline{\frac{3}{2}}, -\frac{1}{4}\} & \frac{1}{2}\{\underline{1},1\} \\ -\frac{1}{4}\{1,\underline{1}\} & \frac{1}{2}I \end{pmatrix} = \begin{pmatrix} 2I & 0 \\ -\frac{1}{4}\{1,\underline{1}\} & \frac{1}{2}I \end{pmatrix}$$

Now we need to apply an elementary lifting of odd type, and we need to find a filter S_2 so that $S_2 I - \frac{1}{4}\{1,\underline{1}\} = \mathbf{0}$. Clearly we can choose $S_2 = \{1/8, \underline{1/8}\}$, and we get

$$\Lambda_2\Lambda_1 H = \begin{pmatrix} I & \mathbf{0} \\ \frac{1}{8}\{1,\underline{1}\} & I \end{pmatrix}\begin{pmatrix} 2I & 0 \\ -\frac{1}{4}\{1,\underline{1}\} & \frac{1}{2}I \end{pmatrix} = \begin{pmatrix} 2I & \mathbf{0} \\ \mathbf{0} & \frac{1}{2}I \end{pmatrix}.$$

Multiplying with inverses of elementary lifting steps, we now obtain that the polyphase representations of the kernels for the Spline 5/3 wavelet are

$$H = \begin{pmatrix} I & \{\underline{1},1\} \\ \mathbf{0} & I \end{pmatrix}\begin{pmatrix} I & \mathbf{0} \\ -\frac{1}{8}\{1,\underline{1}\} & I \end{pmatrix}\begin{pmatrix} 2I & \mathbf{0} \\ \mathbf{0} & \frac{1}{2}I \end{pmatrix}$$

and

$$G = \begin{pmatrix} \frac{1}{2}I & \mathbf{0} \\ \mathbf{0} & 2I \end{pmatrix} \begin{pmatrix} I & \mathbf{0} \\ \frac{1}{8}\{1,\underline{1}\} & I \end{pmatrix} \begin{pmatrix} I & \{\underline{-1},-1\} \\ \mathbf{0} & I \end{pmatrix},$$

respectively. Two lifting steps are thus required. We also see that the lifting steps involve only dyadic fractions, just as the filter coefficients did. This means that the lifting factorization also can be used for lossless operations.

Example 8.15. *Lifting factorization of the CDF 9/7 wavelet.*

For the wavelet we considered in Example 7.17, it is more cumbersome to compute the lifting factorization by hand. It is however, straightforward to write an algorithm which computes the lifting steps, as these are performed in the proof of Theorem 8.8. You will be spared the details of this algorithm. Also, when we use these wavelets in implementations later they will use precomputed values of these lifting steps, and you can take these implementations for granted too. If we run the algorithm for computing the lifting factorization we obtain that the polyphase representations of the kernels H and G for the CDF 9/7 wavelet are

$$\begin{pmatrix} I & 0.5861\{\underline{1},1\} \\ 0 & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0.6681\{1,\underline{1}\} & I \end{pmatrix} \begin{pmatrix} I & -0.0700\{\underline{1},1\} \\ 0 & I \end{pmatrix}$$
$$\times \begin{pmatrix} I & 0 \\ -1.2002\{1,\underline{1}\} & I \end{pmatrix} \begin{pmatrix} -1.1496 & 0 \\ 0 & -0.8699 \end{pmatrix} \text{ and}$$
$$\begin{pmatrix} -0.8699 & 0 \\ 0 & -1.1496 \end{pmatrix} \begin{pmatrix} I & 0 \\ 1.2002\{1,\underline{1}\} & I \end{pmatrix} \begin{pmatrix} I & 0.0700\{\underline{1},1\} \\ 0 & I \end{pmatrix}$$
$$\times \begin{pmatrix} I & 0 \\ -0.6681\{1,\underline{1}\} & I \end{pmatrix} \begin{pmatrix} I & -0.5861\{\underline{1},1\} \\ 0 & I \end{pmatrix},$$

respectively. In this case four lifting steps were required.

Perhaps more important than the reduction in the number of arithmetic operations is the fact that the lifting factorization splits the DWT and IDWT into simpler components, each very attractive for hardware implementations since a lifting step only requires the additional value λ_i from Theorem 8.9. Lifting actually provides us with a complete implementation strategy for the DWT and IDWT, in which the λ_i are used as precomputed values.

Finally we will find a lifting factorization for orthonormal wavelets. Note that here the filters H_0 and H_1 are not symmetric, and each of them has an even number of filter coefficients. There are thus a different number of filter coefficients with positive and negative indices, and in Section 7.7 we defined the filters so that the filter coefficients were as symmetric as possible when it came to the number of nonzero filter coefficients with positive and negative indices.

Example 8.16. *Lifting of orthonormal wavelets.*

We will attempt to construct a lifting factorization where the following property is preserved after each lifting step:

P1: $H^{(0,0)}$, $H^{(1,0)}$ have a minimum possible overweight of filter coefficients with negative indices.

This property stems from the assumption in Section 7.7 that H_0 is assumed to have a minimum possible overweight of filter coefficients with negative indices. To see that this holds at the start, assume as before that all the filters have $2L$ nonzero filter coefficients, so that $H0$ and H_1 are on the form given by Equation (7.30). Assume first that L is even. It is clear that

$$\begin{aligned}
H^{(0,0)} &= \{t_{-L}, \ldots, t_{-2}, \underline{t_0}, t_2, \ldots, t_{L-2}\} \\
H^{(0,1)} &= \{t_{-L+1}, \ldots, t_{-3}, \underline{t_{-1}}, t_1, \ldots, t_{L-1}\} \\
H^{(1,0)} &= \{s_{-L+1}, \ldots, s_{-1}, \underline{s_1}, s_3, \ldots, s_{L-1}\} \\
H^{(1,1)} &= \{s_{-L+2}, \ldots, s_{-2}, \underline{s_0}, s_2, \ldots, s_L\}.
\end{aligned}$$

Clearly P1 holds. Assume now that L is odd. It is now clear that

$$\begin{aligned}
H^{(0,0)} &= \{t_{-L+1}, \ldots, t_{-2}, \underline{t_0}, t_2, \ldots, t_{L-1}\} \\
H^{(0,1)} &= \{t_{-L}, \ldots, t_{-3}, \underline{t_{-1}}, t_1, \ldots, t_{L-2}\} \\
H^{(1,0)} &= \{s_{-L+2}, \ldots, s_{-1}, \underline{s_1}, s_3, \ldots, s_L\} \\
H^{(1,1)} &= \{s_{-L+1}, \ldots, s_{-2}, \underline{s_0}, s_2, \ldots, s_{L-1}\}.
\end{aligned}$$

In this case it is seen that all filters have equally many filter coefficients with positive and negative indices, so that P1 holds also here.

Now let us turn to the first lifting step. We will choose it so that the number of filter coefficients in the first column is reduced with 1, and so that $H^{(0,0)}$ has an odd number of coefficients. If L is even, we saw that $H^{(0,0)}$ and $H^{(1,0)}$ had an even number of coefficients, so that the first lifting step must be even. To preserve P1, we must cancel t_{-L}, so that the first lifting step is

$$\Lambda_1 = \begin{pmatrix} I & -t_{-L}/s_{-L+1} \\ \mathbf{0} & I \end{pmatrix}.$$

If L is odd, we saw that $H^{(0,0)}$ and $H^{(1,0)}$ had an odd number of coefficients, so that the first lifting step must be odd. To preserve P1, we must cancel s_L, so that the first lifting step is

$$\Lambda_1 = \begin{pmatrix} I & \mathbf{0} \\ -s_L/t_{L-1} & I \end{pmatrix}.$$

Now that we have a difference of one filter coefficent in the first column, we will reduce the entry with the most filter coefficients with two with a lifting step, until we have $H^{(0,0)} = \{\underline{K}\}$, $H^{(1,0)} = 0$ in the first column.

Assume first that $H^{(0,0)}$ has the most filter coefficients. We then need to apply an even lifting step. Before an even step, the first column has the form

$$\begin{pmatrix} \{t_{-k},\ldots,t_{-1},\underline{t_0},t_1,\ldots,t_k\} \\ \{s_{-k},\ldots,s_{-1},s_0,s_1,\ldots,s_{k-1}\} \end{pmatrix}.$$

We can then choose $\Lambda_i = \begin{pmatrix} I & \{\underline{-t_{-k}/s_{-k}},-t_k/s_{k-1}\} \\ \mathbf{0} & I \end{pmatrix}$ as a lifting step.

Assume then that $H^{(1,0)}$ has the most filter coefficients. We then need to apply an odd lifting step. Before an odd step, the first column has the form

$$\begin{pmatrix} \{t_{-k},\ldots,t_{-1},\underline{t_0},t_1,\ldots,t_k\} \\ \{s_{-k-1},\ldots,s_{-1},s_0,s_1,\ldots,s_k\} \end{pmatrix}.$$

We can then choose $\Lambda_i = \begin{pmatrix} I & \mathbf{0} \\ \{-s_{-k-1}/t_{-k},\underline{-s_k/t_k}\} & I \end{pmatrix}$ as a lifting step.

If L is even we end up with a matrix on the form $\begin{pmatrix} \alpha & \{\underline{0},K\} \\ \mathbf{0} & \beta \end{pmatrix}$, and we can choose the final lifting step as $\Lambda_n = \begin{pmatrix} I & \{\underline{0},-K/\beta\} \\ \mathbf{0} & I \end{pmatrix}$.

If L is odd we end up with a matrix on the form

$$\begin{pmatrix} \alpha & K \\ \mathbf{0} & \beta \end{pmatrix},$$

and we can choose the final lifting step as $\Lambda_n = \begin{pmatrix} I & -K/\beta \\ \mathbf{0} & I \end{pmatrix}$. Again using equations (8.9)-(8.10), this gives us the lifting factorizations.

In summary we see that all even and odd lifting steps take the form $\begin{pmatrix} I & \{\underline{\lambda_1},\lambda_2\} \\ \mathbf{0} & I \end{pmatrix}$ and $\begin{pmatrix} I & \mathbf{0} \\ \lambda_1,\underline{\lambda_2}\} & I \end{pmatrix}$. We see that symmetric lifting steps correspond to the special case when $\lambda_1 = \lambda_2$. The even and odd lifting matrices now used are

$$\begin{pmatrix} 1 & \lambda_1 & 0 & 0 & \cdots & 0 & 0 & \lambda_2 \\ 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & \lambda_2 & 1 & \lambda_1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \lambda_2 & 1 & \lambda_1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \lambda_2 & 1 & \lambda_1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ \lambda_1 & 0 & 0 & 0 & \cdots & 0 & \lambda_2 & 1 \end{pmatrix}, \tag{8.18}$$

respectively. We note that when we reduce elements to the left and right in the upper and lower part of the first column, the same type of reductions must occur in the second column, since the determinant $H^{(0,0)}H^{(1,1)} - H(0,1)H^{(1,0)}$ is a constant after any number of lifting steps.

This example explains the procedure for finding the lifting factorization into steps of the form given in Equation (8.18). You will be spared the details of writing an implementation which applies this procedure. In order to use orthornormal wavelets in implementations, we have implemented a function `liftingfactortho`, which takes N as input, and sets global variables `lambdas`, `alpha`, and `beta`, so that the factorization (8.8) holds. `lambdas` is an $n \times 2$-matrix so that the filter coefficients $\{\underline{\lambda_1}, \lambda_2\}$ or $\{\lambda_1, \underline{\lambda_2}\}$ in the i'th lifting step is found in row i of `lambdas`. In the exercises, you will be asked to implement both these nonsymmetric elementary lifting steps, as well as kernel transformations for orthonormal wavelets, which assume that these global variables have been set, and describe the lifting steps of the wavelet (Exercise 8.10).

Exercise 8.7: Polyphase components for symetric filters

Assume that the filters H_0, H_1 of a wavelet are symmetric, and denote by $S^{(i,j)}$ the polyphase components of the corresponding MRA-matrix H. Show that $S^{(0,0)}$ and $S^{(1,1)}$ are symmetric filters, that the filter coefficients of $S^{(1,0)}$ has symmetry about $-1/2$, and that $S^{(0,1)}$ has symmetry about $1/2$. Also show a similar statement for the MRA-matrix G of the inverse DWT.

Exercise 8.8: Implement elementary lifting steps

Write functions `liftingstepevensymm` and `liftingstepoddsymm` which take λ, a vector $\boldsymbol{x}$, and `symm` as input, and apply the elementary lifting matrices as in Equation (8.13) in the compendium, respectively, to $\boldsymbol{x}$. The parameter `symm` should indicate whether symmetric extensions shall be applied. Your code should handle both when N is odd, and when N is even (as noted previously, when symmetric extensions are not applied, we assume that N is even). The function should not perform matrix multiplication, and apply as few multiplications as possible.

Exercise 8.9: Implementing kernels transformations using lifting

Up to now in this chapter we have obtained lifting factorizations for four different wavelets where the filters are symmetric. Let us now implement the kernel transformations for these wavelets. Your functions should call the functions from Exercise 8.8 in order to compute the individual lifting steps. Recall that the kernel transformations should take the input vector `x`, `symm` (i.e. whether symmetric extension should be applied), and `dual` (i.e. whether the dual wavelet transform should be applied) as input. You will need equations (8.13) in the compendium-(8.12) in the compendium here, in order to complete the kernels for bot the transformations and the dual transformations.

a) Write the DWT and IDWT kernel transformations for the piecewise linear wavelet. Your functions should use the lifting factorizations in (8.16) in the compendium. Call your functions `DWTKernelpwl0` and `IDWTKernelpwl0`.

b) Write the DWT and IDWT kernel transformations for the alternative piecewise linear wavelet. The lifting factorizations are now given by (8.17) in the compendium instead. Call your functions `DWTKernelpwl2` and `IDWTKernelpwl2`.

c) Write the DWT and IDWT kernel transformations for the Spline 5/3 wavelet, using the lifting factorization obtained in Example 8.14. Call your functions `DWTKernel53` and `IDWTKernel53`.

d) Write the DWT and IDWT kernel transformations for the CDF 9/7 wavelet, using the lifting factorization obtained in Example 8.15. Call your functions `DWTKernel97` and `IDWTKernel97`.

e) In Chapter 5, we listened to the low-resolution approximations and detail components in sound for three different wavelets, using the function `playDWT`. Repeat these experiments with the Spline 5/3 and the CDF 9/7 wavelet, using the new kernels we have implemented in this exercise.

f) Use the function `plotwaveletfunctions` from Exercise 7.1 to plot all scaling functions and mother wavelets for the Spline 5/3 and the CDF 9/7 wavelets, using the kernels you have implemented.

Exercise 8.10: Lifting orthonormal wavelets

In this exercise we will implement the kernel transformations for orthonormal wavelets.

a) Write functions `liftingstepeven` and `liftingstepodd` which take λ_1, λ_2 and a vector $\boldsymbol{x}$ as input, and apply the elementary lifting matrices (8.18) in the compendium, respectively, to $\boldsymbol{x}$. Assume that N is even.

b) Write functions `DWTKernelOrtho` and `IDWTKernelOrtho` which take a vector $\boldsymbol{x}$ as input, and apply the DWT and IDWT kernel transformations for orthonormal wavelets to $\boldsymbol{x}$. You should call the functions `liftingstepeven` and `liftingstepodd`. As mentioned, assume that global variables `lambdas`, `alpha`, and `beta` have been set, so that the lifting factorization (8.8) in the compendium holds, where `lambdas` is a $n \times 2$-matrix so that the filter coefficients $\{\underline{\lambda_1}, \lambda_2\}$ or $\{\lambda_1, \underline{\lambda_2}\}$ in the i'th lifting step is found in row i of `lambdas`. Recall that the last lifting step was even.

c) Listen to the low-resolution approximations and detail components in sound for orthonormal wavelets for $N = 1, 2, 3, 4$, again using the function `playDWT`. You need to call the function `liftingfactortho` in order to set the kernel for the different values of N.

d) Use the function `plotwaveletfunctions` from Exercise 7.1 to plot all scaling functions and mother wavelets for orthonormal wavelets for $N = 1, 2, 3, 4$. Since the wavelets are orthonormal, we should have that $\phi = \tilde{\phi}$, and $\psi = \tilde{\psi}$. In other words, you should see that the bottom plots equal the upper plots.

Exercise 8.11: 4 vanishing moments

In Exercise 5.31 we found constants $\alpha, \beta, \gamma, \delta$ which give the coordinates of $\hat{\psi}$ in $(\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)$, where $\hat{\psi}$ had four vanishing moments, and where we worked with the multiresolution analysis of piecewise constant functions.

a) Show that the polyphase representation of G when $\hat{\psi}$ is used as mother wavelet can be factored as

$$\frac{1}{\sqrt{2}} \begin{pmatrix} I & \mathbf{0} \\ \{1/2, \underline{1/2}\} & I \end{pmatrix} \begin{pmatrix} I & \{-\gamma, \underline{-\alpha}, -\beta, -\delta\} \\ \mathbf{0} & I \end{pmatrix}. \tag{8.19}$$

You here need to reconstruct what you did in the lifting factorization for the alternative piecewise linear wavelet, i.e. write

$$P_{\mathcal{D}_1 \leftarrow (\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)} = P_{\mathcal{D}_1 \leftarrow (\boldsymbol{\phi}_1, \boldsymbol{\psi}_1)} P_{(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1) \leftarrow (\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)}.$$

By inversion, find also a lifting factorization of H.

Exercise 8.12: Wavelet based on piecewise quadratic scaling function

In Exercise 7.4 you should have found the filters

$$\begin{aligned} H_0 &= \frac{1}{128}\{-5, 20, -1, -96, 70, \underline{280}, 70, -96, -1, 20, -5\} \\ H_1 &= \frac{1}{16}\{1, -4, \underline{6}, -4, 1\} \\ G_0 &= \frac{1}{16}\{1, 4, \underline{6}, 4, 1\} \\ G_1 &= \frac{1}{128}\{5, 20, 1, -96, -70, \underline{280}, -70, -96, 1, 20, 5\}. \end{aligned}$$

a) Show that

$$\begin{pmatrix} I & -\frac{1}{128}\{5, \underline{-29}, -29, 5\} \\ \mathbf{0} & I \end{pmatrix} \begin{pmatrix} I & \mathbf{0} \\ -\{1, \underline{1}\} & I \end{pmatrix} \begin{pmatrix} I & -\frac{1}{4}\{\underline{1}, 1\} \\ \mathbf{0} & I \end{pmatrix} G = \begin{pmatrix} \frac{1}{4} & \mathbf{0} \\ \mathbf{0} & 4 \end{pmatrix}.$$

From this we can easily derive the lifting factorization of G.

b) Implement the kernels of the wavelet of this exercise using what you did in Exercise 6.12.

c) Listen to the low-resolution approximations and detail components in sound for this wavelet.

d) Use the function `plotwaveletfunctions` from Exercise 7.1 to plot all scaling functions and mother wavelets for this wavelet.

8.3 Cosine-modulated filter banks and the MP3 standard

Previously we saw that the MP3 standard used a certain filter bank, called a cosine-modulated filter bank. We also illustrated that, surprisingly for a much used international standard, the synthesis system did not exactly invert the analysis system, i.e. we do not have perfect reconstruction, only "near-perfect reconstruction". In this section we will first explain how this filter bank can be constructed, and why it can not give perfect reconstruction. In particular it will be clear how the prototype filter can be constructed. We will then construct a very similar filter bank, which actually can give perfect reconstruction. It may seem very surprising that the MP3 standard does not use this filter bank instead due to this. The explanation may lie in that the MP3 standard was established at about the same time as these filter banks were developed, so that the standard did not capture this very similar filter bank with perfect reconstruction.

8.3.1 Polyphase representations of the filter bank transforms of the MP3 standard

The main idea is to find the polyphase representations of the forward and reverse filter bank transforms of the MP3 standard. We start with the expression

$$z_{32(s-1)+n} = \sum_{k=0}^{511} \cos((n+1/2)(k-16)\pi/32) h_k x_{32s-k-1}, \tag{8.20}$$

which lead to the expression of the forward filter bank transform (Theorem 6.26). Using that any $k < 512$ can be written uniquely on the form $k = m + 64r$, where $0 \leq m < 64$, and $0 \leq r < 8$, we can rewrite this as

$$\begin{aligned} &= \sum_{m=0}^{63} \sum_{r=0}^{7} (-1)^r \cos\left(2\pi(n+1/2)(m-16)/64\right) h_{m+64r} x_{32s-(m+64r)-1} \\ &= \sum_{m=0}^{63} \cos\left(2\pi(n+1/2)(m-16)/64\right) \sum_{r=0}^{7} (-1)^r h_{m+32\cdot 2r} x_{32(s-2r)-m-1}. \end{aligned}$$

Here we also used Property (6.31). If we write

$$V^{(m)} = \{\underline{(-1)^0 h_m}, 0, (-1)^1 h_{m+64}, 0, (-1)^2 h_{m+128}, \ldots, (-1)^7 h_{m+7\cdot 64}, 0\}, \tag{8.21}$$

for $0 \leq m \leq 63$, and we can write the expression above as

$$\begin{aligned}
&\sum_{m=0}^{63} \cos\left(2\pi(n+1/2)(m-16)/64\right) \sum_{r=0}^{15} V_r^{(m)} x_{32(s-r)-m-1} \\
&= \sum_{m=0}^{63} \cos\left(2\pi(n+1/2)(m-16)/64\right) \sum_{r=0}^{15} V_r^{(m)} \boldsymbol{x}_{s-1-r}^{(32-m-1)} \\
&= \sum_{m=0}^{63} \cos\left(2\pi(n+1/2)(m-16)/64\right) (V^{(m)} \boldsymbol{x}^{(32-m-1)})_{s-1},
\end{aligned}$$

where we recognized $x_{32(s-r)-m-1}$ in terms of the polyphase components of $\boldsymbol{x}$, and the inner sum as a convolution. We remark that the inner terms $\{(V^{(m)}\boldsymbol{x}^{(32-m-1)})_{s-1}\}_{m=0}^{63}$ here are what the standard calls partial calculations (windowing refers to multiplication with the combined set of filter coefficients of the $V^{(m)}$), and that matrixing here represents the multiplication with the cosine entries. Since $\boldsymbol{z}^{(n)} = \{z_{32(s-1)+n}\}_{s=0}^{\infty}$ is the n'th polyphase component of $\boldsymbol{z}$, this can be written as

$$\boldsymbol{z}^{(n)} = \sum_{m=0}^{63} \cos\left(2\pi(n+1/2)(m-16)/64\right) I V^{(m)} \boldsymbol{x}^{(32-m-1)}.$$

In terms of matrices this can be written as

$$\begin{aligned}
\boldsymbol{z} = &\begin{pmatrix} \cos\left(2\pi(0+1/2)\cdot(-16)/64\right) I & \cdots & \cos\left(2\pi(0+1/2)\cdot(47)/64\right) I \\ \vdots & \ddots & \vdots \\ \cos\left(2\pi(31+1/2)\cdot(-16)/64\right) I & \cdots & \cos\left(2\pi(31+1/2)\cdot(47)/64\right) I \end{pmatrix} \\
&\times \begin{pmatrix} V^{(0)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & V^{(1)} & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & V^{(62)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & V^{(63)} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \boldsymbol{x}^{(30)} \\ \vdots \\ \boldsymbol{x}^{(-32)} \end{pmatrix}.
\end{aligned}$$

If we place the 15 first columns in the cosine matrix last using Property (6.31) (we must then also place the 15 first rows last in the second matrix), we obtain

$$\boldsymbol{z} = \begin{pmatrix} \cos\left(2\pi(0+1/2)\cdot(0)/64\right) I & \cdots & \cos\left(2\pi(0+1/2)\cdot(63)/64\right) I \\ \vdots & \ddots & \vdots \\ \cos\left(2\pi(31+1/2)\cdot(0)/64\right) I & \cdots & \cos\left(2\pi(31+1/2)\cdot(63)/64\right) I \end{pmatrix}$$
$$\times \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & V^{(16)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & V^{(63)} \\ -V^{(0)} & \cdots & \cdots & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \mathbf{0} \\ \mathbf{0} & \cdots & -V^{(15)} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \boldsymbol{x}^{(30)} \\ \vdots \\ \boldsymbol{x}^{(-32)} \end{pmatrix}.$$

Using Equation (6.32) to combine column k and $64-k$ in the cosine matrix (as well as row k and $64-k$ in the second matrix), we can write this as

$$\begin{pmatrix} \cos\left(2\pi(0+1/2)\cdot(0)/64\right) I & \cdots & \cos\left(2\pi(0+1/2)\cdot(31)/64\right) I \\ \vdots & \ddots & \vdots \\ \cos\left(2\pi(31+1/2)\cdot(0)/64\right) I & \cdots & \cos\left(2\pi(31+1/2)\cdot(31)/64\right) I \end{pmatrix} \begin{pmatrix} A' & B' \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \boldsymbol{x}^{(30)} \\ \vdots \\ \boldsymbol{x}^{(-32)} \end{pmatrix}.$$

where

$$A' = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & V^{(16)} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & V^{(15)} & \mathbf{0} & V^{(17)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \mathbf{0} \\ \mathbf{0} & V^{(1)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & V^{(31)} \\ V^{(0)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix}$$

$$B' = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ V_{(32)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & V^{(33)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & -V^{(63)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & V^{(47)} & \mathbf{0} & -V^{(49)} & \cdots & \mathbf{0} \end{pmatrix}.$$

Using Equation (4.3), the cosine matrix here can be written as

$$\sqrt{\frac{M}{2}}(D_M)^T \begin{pmatrix} \sqrt{2} & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}.$$

The above can thus be written as

$$4(D_{32})^T \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \boldsymbol{x}^{(30)} \\ \vdots \\ \boldsymbol{x}^{(-32)} \end{pmatrix},$$

where A and B are the matrices A', B' with the first row multiplied by $\sqrt{2}$ (i.e. replace $V^{(16)}$ with $\sqrt{2}V^{(16)}$ in the matrix A'). Using that $\boldsymbol{x}^{(-i)} = E_1\boldsymbol{x}_i$ for $1 \leq i \leq 32$, we can write this as

$$4(D_{32})^T \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \vdots \\ \boldsymbol{x}^{(0)} \\ E_1\boldsymbol{x}^{(31)} \\ \vdots \\ E_1\boldsymbol{x}^{(0)} \end{pmatrix} = 4(D_{32})^T \left(A \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \vdots \\ \boldsymbol{x}^{(0)} \end{pmatrix} + B \begin{pmatrix} E_1\boldsymbol{x}^{(31)} \\ \vdots \\ E_1\boldsymbol{x}^{(0)} \end{pmatrix} \right),$$

which can be written as

$$4(D_{32})^T \begin{pmatrix} \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \sqrt{2}V^{(16)} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & V^{(15)} & \mathbf{0} & V^{(17)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \\ \mathbf{0} & V^{(1)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & V^{(31)} \\ V^{(0)} + E_1V^{(32)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & E_1V^{(33)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & -E_1V^{(63)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & E_1V^{(47)} & \mathbf{0} & -E_1V^{(49)} & \cdots & \mathbf{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \vdots \\ \boldsymbol{x}^{(0)} \end{pmatrix},$$

which also can be written as

$$4(D_{32})^T \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \sqrt{2}V^{(16)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \cdots & V^{(17)} & \mathbf{0} & V^{(15)} & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ V^{(31)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & V^{(1)} & \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & V^{(0)} + E_1 V^{(32)} \\ -E_1 V^{(63)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & E_1 V^{(33)} & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \cdots & -E_1 V^{(49)} & \mathbf{0} & E_1 V^{(47)} & \cdots & \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(0)} \\ \vdots \\ \boldsymbol{x}^{(31)} \end{pmatrix}.$$

We have therefore proved the following result.

Theorem 8.17. *Polyphase factorization of a forward filter bank transform based on a prototype filter.*

The polyphase form of a forward filter bank transform based on a prototype filter can be factored as

$$4(D_{32})^T \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \sqrt{2}V^{(16)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \cdots & V^{(17)} & \mathbf{0} & V^{(15)} & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ V^{(31)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & V^{(1)} & \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & V^{(0)} + E_1 V^{(32)} \\ -E_1 V^{(63)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & E_1 V^{(33)} & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \cdots & -E_1 V^{(49)} & \mathbf{0} & E_1 V^{(47)} & \cdots & \mathbf{0} & \mathbf{0} \end{pmatrix} \tag{8.22}$$

Due to Theorem 6.28, it is also very simple to write down the polyphase factorization of the reverse filter bank transform as well. Since $E_{481}G^T$ is a forward filter bank transform where the prototype filter has been reversed, $E_{481}G^T$ can be factored as above, with $V^{(m)}$ replaced by $W^{(m)}$, with $W^{(m)}$ being the filters derived from the synthesis prototype filter in reverse order. This means that the polyphase form of G can be factored as

$$4\begin{pmatrix} \mathbf{0} & \mathbf{0} & \cdots & (W^{(31)})^T & \mathbf{0} & -E_{-1}(W^{(63)})^T & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \mathbf{0} & (W^{(17)})^T & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & -E_{-1}(W^{(49)})^T \\ \sqrt{2}(W^{(16)})^T & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & (W^{(15)})^T & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & E_{-1}(W^{(47)})^T \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & (W^{(1)})^T & \mathbf{0} & E_{-1}(W^{(33)})^T & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & (W^{(0)})^T + E_{-1}(W^{(32)})^T & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \times D_{32}E_{481}. \tag{8.23}$$

Now, if we define $U^{(m)}$ as the filters derived from the synthesis prototype filter itself, we have that

$$(W^{(k)})^T = -E_{-14}V^{(64-k)},\ 1 \leq k \leq 15 \qquad (W^{(0)})^T = E_{-16}V^{(0)}.$$

Inserting this in Equation (8.23) we get the following result:

Theorem 8.18. *Polyphase factorization of a reverse filter bank transform based on a prototype filter.*

Assume that G is a reverse filter filter bank transform based on a prototype filter, and that $U^{(m)}$ are the filters derived from this prototype filter. Then the polyphase form of G can be factored as

$$4\begin{pmatrix} \mathbf{0} & \mathbf{0} & \cdots & -U^{(33)} & \mathbf{0} & E_{-1}U^{(1)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \mathbf{0} & -U^{(47)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & E_{-1}U^{(15)} \\ -\sqrt{2}U^{(48)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & -U^{(49)} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & -E_{-1}U^{(17)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & -U^{(63)} & \mathbf{0} & -E_{-1}U^{(31)} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & E_{-2}U^{(0)} - E_{-1}U^{(32)} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \times D_{32}E_{33}. \tag{8.24}$$

Now, consider the matrices

$$\begin{pmatrix} V^{(32-i)} & V^{(i)} \\ -E_1V^{(64-i)} & E_1V^{(32+i)} \end{pmatrix} \text{ and } \begin{pmatrix} -U^{(32+i)} & E_{-1}U^{(i)} \\ -U^{(64-i)} & -E_{-1}U^{(32-i)} \end{pmatrix}. \tag{8.25}$$

for $1 \leq i \leq 15$. These make out submatrices in the matrices in equations (8.22) and (8.24). Clearly, only the product of these matrices influence the result. Since

$$\begin{pmatrix} -U^{(32+i)} & E_{-1}U^{(i)} \\ -U^{(64-i)} & -E_{-1}U^{(32-i)} \end{pmatrix} \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ -E_1V^{(64-i)} & E_1V^{(32+i)} \end{pmatrix}$$
$$= \begin{pmatrix} -U^{(32+i)} & U^{(i)} \\ -U^{(64-i)} & -U^{(32-i)} \end{pmatrix} \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ -V^{(64-i)} & V^{(32+i)} \end{pmatrix} \tag{8.26}$$

we have the following result.

Theorem 8.19. *Filter bank transforms.*

Let H, G be forward and reverse filter bank transforms defined from analysis and synthesis prototype filters. Let also $V^{(k)}$ be the prototype filter of H, and $U^{(k)}$ the reverse of the prototype filter of G. If

$$\begin{pmatrix} -U^{(32+i)} & U^{(i)} \\ -U^{(64-i)} & -U^{(32-i)} \end{pmatrix} \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ -V^{(64-i)} & V^{(32+i)} \end{pmatrix} = c \begin{pmatrix} E_d & \mathbf{0} \\ \mathbf{0} & E_d \end{pmatrix}$$
$$(\sqrt{2}V^{(16)})(-\sqrt{2}U^{(48)}) = cE_d$$
$$(V^{(0)} + E_1V^{(32)})(E_{-2}U^{(0)} - E_{-1}U^{(32)}) = cE_d \tag{8.27}$$

for $1 \leq i \leq 15$, then $GH = 16cE_{33+32d}$.

This result is the key ingredient we need in order to construct forward and reverse systems which together give perfect reconstuction. In Exercise 8.15 we go through how we can use lifting in order to express a wide range of possible (U, V) matrix pairs which satisfy Equation (8.27). This turns the problem of constructing cosine-modulated filter banks which are useful for audio coding into an optimization problem: the optimization variables are values λ_i which characterize lifting steps, and the objective function is the deviation of the corresponding prototype filter from an ideal bandpass filter. This optimization problem has been subject to a lot of research, and we will not go into details on this.

8.3.2 The prototype filters chosen in the MP3 standard

Now, let us return to the MP3 standard. We previously observed that in this standard the coefficients in the synthesis prototype filter seemed to equal 32 times the analysis prototype filter. This indicates that $U^{(k)} = 32V^{(k)}$. A closer inspection also yields that there is a symmetry in the values of the prototype filter: We see that $C_i = -C_{512-i}$ (i.e. antisymmetry) for most values of i. The only exception is for $i = 64, 128, \ldots, 448$, for which $C_i = C_{512-i}$ (i.e. symmetry). The antisymmetry can be translated to that the filter coefficients of $V^{(k)}$ equal those of $V^{(64-k)}$ in reverse order, with a minus sign. The symmetry can be translated to that $V^{(0)}$ is symmetric. These observations can be rewritten as

$$V^{(64-k)} = -E_{14}(V^{(k)})^T, 1 \leq k \leq 15. \tag{8.28}$$
$$V^{(0)} = E_{16}(V^{(0)})^T. \tag{8.29}$$

Inserting first that $U^{(k)} = 32V^{(k)}$ in Equation (8.26) gives

$$\begin{aligned}&\begin{pmatrix} -U^{(32+i)} & U^{(i)} \\ -U^{(64-i)} & -U^{(32-i)} \end{pmatrix} \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ -V^{(64-i)} & V^{(32+i)} \end{pmatrix} \\ =&32 \begin{pmatrix} -V^{(32+i)} & V^{(i)} \\ -V^{(64-i)} & -V^{(32-i)} \end{pmatrix} \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ -V^{(64-i)} & V^{(32+i)} \end{pmatrix}.\end{aligned}$$

Substituting for $V^{(32+i)}$ and $V^{(64-i)}$ after what we found by inspection now gives

$$\begin{aligned}&32 \begin{pmatrix} E_{14}(V^{(32-i)})^T & V^{(i)} \\ E_{14}(V^{(i)})^T & -V^{(32-i)} \end{pmatrix} \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ E_{14}(V^{(i)})^T & -E_{14}(V^{(32-i)})^T \end{pmatrix} \\ =&32 \begin{pmatrix} E_{14} & \mathbf{0} \\ \mathbf{0} & E_{14} \end{pmatrix} \begin{pmatrix} (V^{(32-i)})^T & V^{(i)} \\ (V^{(i)})^T & -V^{(32-i)} \end{pmatrix} \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ (V^{(i)})^T & -(V^{(32-i)})^T \end{pmatrix} \\ =&32 \begin{pmatrix} E_{14} & \mathbf{0} \\ \mathbf{0} & E_{14} \end{pmatrix} \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ (V^{(i)})^T & -(V^{(32-i)})^T \end{pmatrix}^T \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ (V^{(i)})^T & -(V^{(32-i)})^T \end{pmatrix} \\ =&32 \begin{pmatrix} E_{14} & \mathbf{0} \\ \mathbf{0} & E_{14} \end{pmatrix} \begin{pmatrix} V^{(i)}(V^{(i)})^T + V^{(32-i)}(V^{(32-i)})^T & \mathbf{0} \\ \mathbf{0} & V^{(i)}(V^{(i)})^T + V^{(32-i)}(V^{(32-i)})^T \end{pmatrix}.\end{aligned} \tag{8.30}$$

Due to Exercise 8.6 (set $A = (V^{(32-i)})^T, B = (V^{(i)})^T$), with

$$H = \begin{pmatrix} V^{(32-i)} & V^{(i)} \\ (V^{(i)})^T & -(V^{(32-i)})^T \end{pmatrix} \qquad G = \begin{pmatrix} (V^{(32-i)})^T & V^{(i)} \\ (V^{(i)})^T & -V^{(32-i)} \end{pmatrix}$$

we recognize an alternative QMF filter bank. We thus have alias cancellation, with perfect reconstruction only if $|\lambda_{H_0}(\omega)|^2 + |\lambda_{H_0}(\omega + \pi)|^2$. For the two remaining filters we compute

$$\begin{aligned}&(\sqrt{2}V^{(16)})(-\sqrt{2}U^{(48)}) \\ &= -64V^{(16)}V^{(48)} = 64E_{14}V^{(16)}(V^{(16)})^T = 32E_{14}(V^{(16)}(V^{(16)})^T + V^{(16)}(V^{(16)})^T)\end{aligned} \tag{8.31}$$

and

$$\begin{aligned}&(V^{(0)} + E_1V^{(32)})(E_{-2}U^{(0)} - E_{-1}U^{(32)}) \\ &= 32(V^{(0)} + E_1V^{(32)})(E_{-2}V^{(0)} - E_{-1}V^{(32)}) = 32E_{-2}(V^{(0)} + E_1V^{(32)})(V^{(0)} - E_1V^{(32)}) \\ &= 32E_{-2}(V^{(0)})^2 - (V^{(32)})^2) = 32E_{14}((V^{(0)}(V^{(0)})^T + V^{(32)}(V^{(32)})^T)).\end{aligned} \tag{8.32}$$

We see that the filters from equations (8.30)-(8.32) are similar, and that we thus can combine them into

$$\{V^{(i)}(V^{(i)})^T + V^{(32-i)}(V^{(32-i)})^T\}_{i=0}^{16}. \tag{8.33}$$

All of these can be the identity, expect for $1024V^{(16)}(V^{(16)})^T$, since we know that the product of two FIR filters is never the identity, except when both are delays (And all $V^{(m)}$ are FIR, since the prototype filters defined by the MP3 standard are FIR). This single filter is thus what spoils for perfect reconstruction, so that we can only hope for alias cancellation, and this happens when the filters from Equation (8.33) all are equal. Ideally this is close to cI for some scalar c, and we then have that

$$GH = 16 \cdot 32cE_{33+448} = 512cE_{481}I.$$

This explains the observation from the MP3 standard that GH seems to be close to E_{481}. Since all the filters $V^{(i)}(V^{(i)})^T + V^{(32-i)}(V^{(32-i)})^T$ are symmetric, GH is also a symmetric filter due to Theorem 8.4, so that its frequency response is real, so that we have no phase distortion. We can thus summarize our findings as follows.

Observation 8.20. *MP3 standard.*

The prototype filters from the MP3 standard do not give perfect reconstruction. They are found by choosing 17 filters $\{V^{(k)}\}_{k=0}^{16}$ so that the filters from Equation (8.33) are equal, and so that their combination into a prototype filter using equations (8.21) and (8.28) is as close to an ideal bandpass filter as possible. When we have equality the alias cancellation condition is satisfied, and we also have no phase distortion. When the common value is close to $\frac{1}{512}I$, GH is close to E_{481}, so that we have near-perfect reconstruction.

This states clearly the optimization problem which the values stated in the MP3 standard solves.

8.3.3 How can we obtain perfect reconstruction?

How can we overcome the problem that $1024V^{(16)}(V^{(16)})^T \neq I$, which spoiled for perfect reconstruction in the MP3 standard? It turns out that we can address this a simple change in our procedure. In Equation (8.20) we replace with

$$z_{32(s-1)+n} = \sum_{k=0}^{511} \cos((n+1/2)(k+1/2-16)\pi/32)h_k x_{32s-k-1}, \tag{8.34}$$

i.e. $1/2$ is added inside the cosine. We now have the properties

$$\cos\left(2\pi(n+1/2)(k+64r+1/2)/(2N)\right) = (-1)^r \cos\left(2\pi(n+1/2)(k+1/2)/(2N)\right) \tag{8.35}$$

$$\cos\left(2\pi(n+1/2)(2N-k-1+1/2)/(2N)\right) = -\cos\left(2\pi(n+1/2)(k+1/2)/(2N)\right). \tag{8.36}$$

Due to the first property, we can deduce as before that

$$\boldsymbol{z}^{(n)} = \sum_{m=0}^{63} \cos\left(2\pi(n+1/2)(m+1/2-16)/64\right) IV^{(m)} \boldsymbol{x}^{(32-m-1)},$$

where the filters $V^{(m)}$ are defined as before. As before placing the 15 first columns of the cosine-matrix last, but instead using Property (8.36) to combine columns k and $64-k-1$ of the cosine-matrix, we can write this as

$$\begin{pmatrix} \cos\left(2\pi(0+1/2)\cdot(0+1/2)/64\right) I & \cdots & \cos\left(2\pi(0+1/2)\cdot(31+1/2)/64\right) I \\ \vdots & \ddots & \vdots \\ \cos\left(2\pi(31+1/2)\cdot(0+1/2)/64\right) I & \cdots & \cos\left(2\pi(31+1/2)\cdot(31+1/2)/64\right) I \end{pmatrix} \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \vdots \\ \boldsymbol{x}^{(-32)} \end{pmatrix}$$

where

$$A = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \cdots & V^{(15)} & V^{(16)} & \cdots & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & V^{(1)} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & V^{(30)} & \mathbf{0} \\ V^{(0)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \cdots & V^{(31)} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix}$$

$$B = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ V_{(32)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & -V^{(63)} \\ \mathbf{0} & V^{(33)} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & -V^{(62)} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & V^{(47)} & -V^{(48)} & \cdots & \cdots & \mathbf{0} \end{pmatrix}.$$

Since the cosine matrix can be written as $\sqrt{\frac{M}{2}} D_M^{(iv)}$, the above can be written as

$$4D_M^{(iv)} \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(31)} \\ \vdots \\ \boldsymbol{x}^{(-32)} \end{pmatrix}.$$

As before we can rewrite this as

$$4D_M^{(iv)}\begin{pmatrix}A & B\end{pmatrix}\begin{pmatrix}\boldsymbol{x}^{(31)}\\ \vdots\\ \boldsymbol{x}^{(0)}\\ E_1\boldsymbol{x}^{(31)}\\ \vdots\\ E_1\boldsymbol{x}^{(0)}\end{pmatrix} = 4D_M^{(iv)}\left(A\begin{pmatrix}\boldsymbol{x}^{(31)}\\ \vdots\\ \boldsymbol{x}^{(0)}\end{pmatrix} + B\begin{pmatrix}E_1\boldsymbol{x}^{(31)}\\ \vdots\\ E_1\boldsymbol{x}^{(0)}\end{pmatrix}\right),$$

which can be written as

$$4D_M^{(iv)}\begin{pmatrix}\mathbf{0} & \mathbf{0} & \cdots & V^{(15)} & V^{(16)} & \cdots & \cdots & \mathbf{0}\\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots\\ \mathbf{0} & V^{(1)} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & V^{(30)} & \mathbf{0}\\ V^{(0)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \cdots & V^{(31)}\\ E_1V_{(32)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \cdots & -E_1V^{(63)}\\ \mathbf{0} & E_1V^{(33)} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & -E_1V^{(62)} & \mathbf{0}\\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots\\ \mathbf{0} & \mathbf{0} & \cdots & E_1V^{(47)} & -E_1V^{(48)} & \cdots & \cdots & \mathbf{0}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}^{(31)}\\ \vdots\\ \boldsymbol{x}^{(0)}\end{pmatrix},$$

which also can be written as

$$4D_M^{(iv)}\begin{pmatrix}\mathbf{0} & \mathbf{0} & \cdots & V^{(16)} & V^{(15)} & \cdots & \cdots & \mathbf{0}\\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots\\ \mathbf{0} & V^{(30)} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & V^{(1)} & \mathbf{0}\\ V^{(31)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \cdots & V^{(0)}\\ -E_1V_{(63)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \cdots & E_1V^{(32)}\\ \mathbf{0} & -E_1V^{(62)} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & E_1V^{(33)} & \mathbf{0}\\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots\\ \mathbf{0} & \mathbf{0} & \cdots & -E_1V^{(48)} & E_1V^{(47)} & \cdots & \cdots & \mathbf{0}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}^{(0)}\\ \vdots\\ \boldsymbol{x}^{(31)}\end{pmatrix}.$$

We therefore have the following result

Theorem 8.21. *Polyphase factorization of a forward filter bank transform based on a prototype filter, modified version.*

The modified version of the polyphase form of a forward filter bank transform based on a prototype filter can be factored as

$$4D_M^{(iv)}\begin{pmatrix}
\mathbf{0} & \mathbf{0} & \cdots & V^{(16)} & V^{(15)} & \cdots & \cdots & \mathbf{0} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\mathbf{0} & V^{(30)} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & V^{(1)} & \mathbf{0} \\
V^{(31)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \cdots & V^{(0)} \\
-E_1V_{(63)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \cdots & E_1V^{(32)} \\
\mathbf{0} & -E_1V^{(62)} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & E_1V^{(33)} & \mathbf{0} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\mathbf{0} & \mathbf{0} & \cdots & -E_1V^{(48)} & E_1V^{(47)} & \cdots & \cdots & \mathbf{0}
\end{pmatrix} \tag{8.37}$$

Clearly this factorization avoids having two blocks of filters: There are now 16 2×2-polyphase matrices, and as we know, each of them can be invertible, so that the full matrix can be inverted in a similar fashion as before. It is therefore now possible to obtain perfect reconstruction. Although we do not state recipes for implementing this, one has just as efficient implementations as in the MP3 standard.

Since we ended up with the 2×2 polyphase matrices M_k, we can apply the lifting factorization in order to halve the number of multiplications/additions. This is not done in practice, since a lifting factorization requires that we compute all outputs at once. In audio coding it is required that we compute the output progressively, due to the large size of the input vector. The procedure above is therefore mostly useful for providing the requirements for the filters, while the preceding comments can be used for the implementation.

Exercise 8.13: Run forward and reverse transform

Run the forward and then the reverse transform from Exercise 6.16 on the vector $(1, 2, 3, \ldots, 8192)$. Verify that there seems to be a delay on 481 elements, as promised by Therorem 8.20. Do you get the exact same result back?

Exercise 8.14: Verify statement of filters

Use your computer to verify the symmetries we have stated for the symmetries in the prototype filters, i.e. that

$$C_i = \begin{cases} -C_{512-i} & i \neq 64, 128, \ldots, 448 \\ C_{512-i} & i = 64, 128, \ldots, 448. \end{cases}$$

Explain also that this implies that $h_i = h_{512-i}$ for $i = 1, \ldots, 511$. In other words, the prototype filter has symmetry around $(511+1)/2 = 256$, so that it has linear phase.

Exercise 8.15: Lifting

We mentioned that we could use the lifting factorization to construct filters on the form stated in Equation (8.21) in the compendium, so that the matrices on the form given by Equation (8.25) in the compendium, i.e.

$$\begin{pmatrix} V^{(32-i)} & V^{(i)} \\ -V^{(64-i)} & V^{(32+i)} \end{pmatrix},$$

are invertible. Let us see what kind of lifting steps produce such matrices.

a) Show that the lifting steps

$$\begin{pmatrix} I & \lambda E_2 \\ \mathbf{0} & I \end{pmatrix} \text{ and } \begin{pmatrix} I & \mathbf{0} \\ \lambda I & I \end{pmatrix}$$

applied in alternating order to a matrix on the form given by Equation (8.25) in the compendium, where the filters are on the from given by Equation (8.21) in the compendium, again produces matrices and filters on these forms. This explains how we can parametrize a larger number of such matrices with the help of lifting steps.It also explain why the inverse matrix is on the form stated in Equation (8.25) in the compendium with filters on the same form, since the inverse lifting steps are of the same type.

b) Explain that 16 numbers $\{\lambda_i\}_{i=1}^{16}$ are needed (together with what we start with on the diagonal in the lifting construction), in order to construct filters so that the prototype filter has 512 coefficients. Since there are 15 submatrices, this gives 240 optimization variables.

Lifting gives the following strategy for finding a corresponding synthesis prototype filter which gives perfect reconstruction: First compute matrices V, W which are inverses of oneanother using lifting (using the lifting steps of this exercise ensures that all filters will be on the form stated in Equation (8.21) in the compendium), and write

$$\begin{aligned} VW &= \begin{pmatrix} V^{(1)} & V^{(2)} \\ -V^{(3)} & V^{(4)} \end{pmatrix} \begin{pmatrix} W^{(1)} & -W^{(3)} \\ W^{(2)} & W^{(4)} \end{pmatrix} = \begin{pmatrix} V^{(1)} & V^{(2)} \\ -V^{(3)} & V^{(4)} \end{pmatrix} \begin{pmatrix} (W^{(1)})^T & (W^{(2)})^T \\ -(W^{(3)})^T & (W^{(4)})^T \end{pmatrix}^T \\ &= \begin{pmatrix} V^{(1)} & V^{(2)} \\ -V^{(3)} & V^{(4)} \end{pmatrix} \begin{pmatrix} E_{15}(W^{(1)})^T & E_{15}(W^{(2)})^T \\ -E_{15}(W^{(3)})^T & E_{15}(W^{(4)})^T \end{pmatrix}^T \begin{pmatrix} E_{15} & \mathbf{0} \\ \mathbf{0} & E_{15} \end{pmatrix} = I. \end{aligned}$$

Now, the matrices $U^{(i)} = E_{15}(W^{(i)})^T$ are on the form stated in Equation (8.21) in the compendium, and we have that

$$\begin{pmatrix} V^{(1)} & V^{(2)} \\ -V^{(3)} & V^{(4)} \end{pmatrix} \begin{pmatrix} U^{(1)} & U^{(2)} \\ -U^{(3)} & U^{(4)} \end{pmatrix} = \begin{pmatrix} E_{-15} & \mathbf{0} \\ \mathbf{0} & E_{-15} \end{pmatrix}$$

We can now conclude from Theorem 8.19 that if we define the synthesis prototype filter as therein, and set $c = 1, d = -15$, we have that $GH = 16E_{481-32\cdot 15} = 16E_1$.

8.4 Summary

We defined the polyphase representation of a matrix, and proved some useful properties. For filter bank transforms, the polyphase representation was a block matrix where the blocks are filters, and these blocks/filters were called polyphase components. In particular, the filter bank transforms of wavelets were 2×2-block matrices of filters. We saw that, for wavelets, the polyphase representation could be realized through a rearrangement of the wavelet bases, and thus paralleled the development in Chapter 6 for expressing the DWT in terms of filters, where we instead rearranged the target base of the DWT.

We showed with two examples that factoring the polyphase representation into simpler matrices (also refered to as a polyphase factorization) could be a useful technique. First, for wavelets ($M = 2$), we established the lifting factorization. This is useful not only since it factorizes the DWT and the IDWT into simpler operations, but also since it reduces the number of arithmetic operations in these. The lifting factorization is therefore also used in practical implementations, and we applied it to some of the wavelets we constructed in Chapter 7. The JPEG2000 standard document [17] explains a procedure for implementing some of these wavelet transforms using lifting, and the values of the lifting steps used in the standard thus also appear here.

The polyphase representation was also useful for proving the characterization of wavelets we encountered in Chapter 7, which we used to find expressions for many useful wavelets.

The polyphase representation was also useful to explain how the prototype filters of the MP3 standard should be chosen, in order for the reverse filter bank transform to invert the forward filter bank transform. Again this was attacked by factoring the polyphase representation of the forward and reverse filter bank transforms. The parts of the factorization which represented the prototype filters were represented by a sparse matrix, and it was clear from this matrix what properties we needed to put on the prototype filter, in order to have alias cancellation, and no phase distortion. In fact, we proved that the MP3 standard could not possible give perfect reconstruction, but it was very clear from our construction how the filter bank could be modified in order for the overall system to provide perfect reconstruction.

The lifting scheme as introduced here was first proposed by Sweldens [35]. How to use lifting for in-place calculation for the DWT was also suggested by Sweldens [34].

This development concludes the one-dimensional aspect of wavelets in this book. In the following we will extend our theory to also apply for images. Images will be presented in Chapter 9. After that we will define the tensor product concept, which will be the key ingredient to apply wavelets to two-dimensional objects such as images.

Chapter 9

Digital images

Upto now we have presented wavelets in a one-dimensional setting. Images, however, are two-dimensional by nature. This poses another challenge, which we did not encounter in the case of sound signals. In this chapter we will establish the mathematics to handle this, but first we will present some basics on images, as well as how they can be represented and manipulated with simple mathematics. Images are a very important type of digital media, and this material is thus useful, general knowledge for anyone with a digital camera and a computer. For many scientists this material is also an essential tool. As an example, in astrophysics data from both satellites and distant stars and galaxies is collected in the form of images, and information is extracted from the images with advanced image processing techniques. As another example, medical imaging makes it possible to gather different kinds of information in the form of images, even from the inside of the body. By analysing these images it is possible to discover tumours and other disorders.

We will see how filter-based operations extend naturally to the two-dimensional setting of images. Smoothing and edge detections are the two main examples of filter-based operations we will concider for images. The key mathematical concept in this extension is the *tensor product*, which can be thought of as a general tool for constructing two-dimensional objects from one-dimensional counterparts. We will also see that the tensor product allows us to establish an efficient implementation of filtering for images, efficient meaning a complexity substantially less than what is required by general linear transformations.

We will finally consider useful coordinate changes for images. Recall that the DFT, the DCT, and the wavelet transform were all defined as changes of coordinates for vectors or functions of one variable, and therefore cannot be directly applied to two-dimensional data like images. It turns out that the tensor product can also be used to extend changes of coordinates to a two-dimensional setting.

Functionality for accessing images are collected in a module called ‘images.

9.1 What is an image?

Before we do computations with images, it is helpful to be clear about what an image really is. Images cannot be perceived unless there is some light present, so we first review superficially what light is.

9.1.1 Light

Fact 9.1. *Light.*

Light is electromagnetic radiation with wavelengths in the range 400–700 nm (1 nm is 10^{-9} m): Violet has wavelength 400 nm and red has wavelength 700 nm. White light contains roughly equal amounts of all wave lengths.

Other examples of electromagnetic radiation are gamma radiation, ultraviolet and infrared radiation and radio waves, and all electromagnetic radiation travel at the speed of light ($\approx 3 \times 10^8$ m/s). Electromagnetic radiation consists of waves and may be reflected and refracted, just like sound waves (but sound waves are not electromagnetic waves).

We can only see objects that emit light, and there are two ways that this can happen. The object can emit light itself, like a lamp or a computer monitor, or it reflects light that falls on it. An object that reflects light usually absorbs light as well. If we perceive the object as red it means that the object absorbs all light except red, which is reflected. An object that emits light is different; if it is to be perceived as being red it must emit only red light.

9.1.2 Digital output media

Our focus will be on objects that emit light, for example a computer display. A computer monitor consists of a matrix of small dots which emit light. In most technologies, each dot is really three smaller dots, and each of these smaller dots emit red, green and blue light. If the amounts of red, green and blue is varied, our brain merges the light from the three small light sources and perceives light of different colors. In this way the color at each set of three dots can be controlled, and a color image can be built from the total number of dots.

It is important to realise that it is possible to generate most, but not all, colors by mixing red, green and blue. In addition, different computer monitors use slightly different red, green and blue colors, and unless this is taken into consideration, colors will look different on the two monitors. This also means that some colors that can be displayed on one monitor may not be displayable on a different monitor.

Printers use the same principle of building an image from small dots. On most printers however, the small dots do not consist of smaller dots of different colors. Instead as many as 7–8 different inks (or similar substances) are mixed to the right color. This makes it possible to produce a wide range of colors, but not all, and the problem of matching a color from another device like a monitor is at least as difficult as matching different colors across different monitors.

Video projectors builds an image that is projected onto a wall. The final image is therefore a reflected image and it is important that the surface is white so that it reflects all colors equally.

The quality of a device is closely linked to the density of the dots.

Fact 9.2. *Resolution.*

The resolution of a medium is the number of dots per inch (dpi). The number of dots per inch for monitors is usually in the range 70–120, while for printers it is in the range 150–4800 dpi. The horizontal and vertical densities may be different. On a monitor the dots are usually referred to as *pixels* (picture elements).

9.1.3 Digital input media

The two most common ways to acquire digital images is with a digital camera or a scanner. A scanner essentially takes a photo of a document in the form of a matrix of (possibly colored) dots. As for printers, an important measure of quality is the number of dots per inch.

Fact 9.3. *Printers.*

The resolution of a scanner usually varies in the range 75 dpi to 9600 dpi, and the color is represented with up to 48 bits per dot.

For digital cameras it does not make sense to measure the resolution in dots per inch, as this depends on how the image is printed (its size). Instead the resolution is measured in the number of dots recorded.

Fact 9.4. *Pixels.*

The number of pixels recorded by a digital camera usually varies in the range 320×240 to 6000×4000 with 24 bits of color information per pixel. The total number of pixels varies in the range 76 800 to 24 000 000 (0.077 megapixels to 24 megapixels).

For scanners and cameras it is easy to think that the more dots (pixels), the better the quality. Although there is some truth to this, there are many other factors that influence the quality. The main problem is that the measured color information is very easily polluted by noise. And of course high resolution also means that the resulting files become very big; an uncompressed 6000×4000 image produces a 72 MB file. The advantage of high resolution is that you can magnify the image considerably and still maintain reasonable quality.

9.1.4 Definition of digital image

We have already talked about digital images, but we have not yet been precise about what they are. From a mathematical point of view, an image is quite simple.

Fact 9.5. *Digital image.*

A digital image P is a matrix of *intensity values* $\{p_{i,j}\}_{i,j=1}^{M,N}$. For grey-level images, the value $p_{i,j}$ is a single number, while for color images each $p_{i,j}$ is a

vector of three or more values. If the image is recorded in the rgb-model, each $p_{i,j}$ is a vector of three values,

$$p_{i,j} = (r_{i,j}, g_{i,j}, b_{i,j}),$$

that denote the amount of red, green and blue at the point (i, j).

Note that, when referring to the coordinates (i, j) in an image, i will refer to row index, j to column index, in the same was as for matrices. In particular, the top row in the image have coordinates $\{(0, j)\}_{j=0}^{N-1}$, while the left column in the image has coordinates $\{(i, 0)\}_{i=0}^{M-1}$. With this notation, the dimension of the image is $M \times N$. The value $p_{i,j}$ gives the color information at the point (i, j). It is important to remember that there are many formats for this. The simplest case is plain black and white images in which case $p_{i,j}$ is either 0 or 1. For grey-level images the intensities are usually integers in the range 0–255. However, we will assume that the intensities vary in the interval $[0, 1]$, as this sometimes simplifies the form of some mathematical functions. For color images there are many different formats, but we will just consider the rgb-format mentioned in the fact box. Usually the three components are given as integers in the range 0–255, but as for grey-level images, we will assume that they are real numbers in the interval $[0, 1]$ (the conversion between the two ranges is straightforward, see Example 9.10 below).

Figure 9.1: Our test image.

In Figure 9.1 we have shown the test image we will work with, called the *Lena image*. It is named after the girl in the image. This image is also used as a test image in many textbooks on image processing.

In Figure 9.2 we have shown the corresponding black and white, and grey-level versions of the test image.

Fact 9.6. *Intensity.*

Figure 9.2: Black and white (left), and grey-level (right) versions of the image in Figure 9.1.

In these notes the intensity values $p_{i,j}$ are assumed to be real numbers in the interval $[0, 1]$. For color images, each of the red, green, and blue intensity values are assumed to be real numbers in $[0, 1]$.

Figure 9.3: 18×18 pixels excerpt of the color image in Figure 9.1. The grid indicates the borders between the pixels.

If we magnify the part of the color image in Figure 9.1 around one of the eyes, we obtain the images in figures 9.3-9.4. As we can see, the pixels have been magnified to big squares. This is a standard representation used by many programs — the actual shape of the pixels will depend on the output medium.

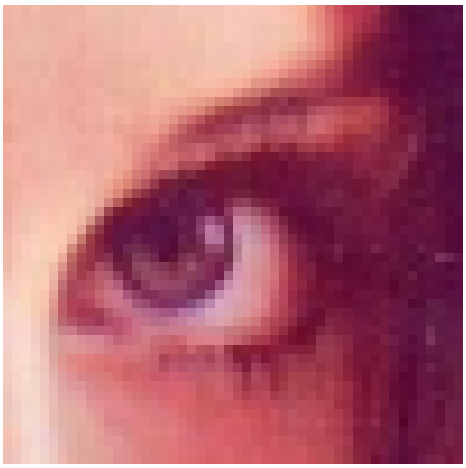

Figure 9.4: 50×50 pixels excerpt of the color image in Figure 9.1.

Nevertheless, we will consider the pixels to be square, with integer coordinates at their centers, as indicated by the grids in figures 9.3-9.4.

Fact 9.7. *Shape of pixel.*

The pixels of an image are assumed to be square with sides of length one, with the pixel with value $p_{i,j}$ centered at the point (i, j).

9.2 Some simple operations on images

Images are two-dimensional matrices of numbers, contrary to the sound signals we considered in the previous section. In this respect it is quite obvious that we can manipulate an image by performing mathematical operations on the numbers. In this section we will consider some of the simpler operations. In later sections we will go through more advanced operations, and explain how the theory for these can be generalized from the corresponding theory for one-dimensional (sound) signals (which we wil go through first).

In order to perform these operations, we need to be able to use images with a programming environment.

9.2.1 Images and Python

An image can also be thought of as a matrix, by associating each pixel with an element in a matrix. The matrix indices thus correspond to positions in the pixel grid. Black and white images correspond to matrices where the elements are natural numbers between 0 and 255. To store a color image, we need 3 matrices, one for each color component. We will also view this as a 3-dimensional matrix. In the following, operations on images will be implemented in such a way that they are applied to each color component simultaneously. This is similar to the

FFT and the DWT, where the operations were applied to each sound channel simultaneously.

Since images are viewed as 2-dimensional or 3-dimensional matrices, we can use any linear algebra software in order to work with images. After we now have made the connection with matrices, we can create images from mathematical formulas, just as we could with sound in the previuos sections. But what we also need before we go through operations on images, is, as in the sections on sound, means of reading an image from a file so that its contents are accessible as a matrix, and write images represented by a matrix which we have constructed ourself to file. Reading a function from file can be done with help of the function `imread`. If we write

```
X = double(imread('filename.fmt', 'fmt'))
```

the image with the given path and format is read, and stored in the matrix which we call `X`. 'fmt' can be 'jpg','tif', 'gif', 'png', and so on. This parameter is optional: If it is not present, the program will attempt to determine the format from the first bytes in the file, and from the filename. After the call to `imread`, we have a matrix where the entries represent the pixel values, and of integer data type (more precisely, the data type `uint8`). To perform operations on the image, we must first convert the entries to the data type `double`, as shown above. Similarly, the function `imwrite`

can be used to write the image represented by a matrix to file. If we write

```
imwrite(uint8(X), 'filename.fmt', 'fmt')
```

the image represented by the matrix `X` is written to the given path, in the given format. Before the image is written to file, you see that we have converted the matrix values back to the integer data type. In other words: `imread` and `imwrite` both assume integer matrix entries, while operations on matrices assume double matrix entries. If you want to print images you have created yourself, you can use this function first to write the image to a file, and then send that file to the printer using another program. Finally, we need an alternative to playing a sound, namely displaying an image. The function `imshow(uint8(X))` displays the matrix `X` as an image in a separate window. Also here we needed to convert the samples using the function `uint8`.

The following examples go through some much used operations on images.

Example 9.8. *Normalising the intensities.*

We have assumed that the intensities all lie in the interval $[0, 1]$, but as we noted, many formats in fact use integer values in the range [0,255]. And as we perform computations with the intensities, we quickly end up with intensities outside $[0, 1]$ even if we start out with intensities within this interval. We therefore need to be able to *normalise* the intensities. This we can do with the simple linear function

$$g(x) = \frac{x - a}{b - a}, \quad a < b,$$

which maps the interval $[a, b]$ to $[0, 1]$. A simple case is mapping $[0, 255]$ to $[0, 1]$ which we accomplish with the scaling $g(x) = x/255$. More generally, we typically perform computations that result in intensities outside the interval $[0, 1]$. We can then compute the minimum and maximum intensities $p_{\min}$ and $p_{\max}$ and map the interval $[p_{\min}, p_{\max}]$ back to $[0, 1]$. Below we have shown a function `mapto01` which achieves this task.

```
def mapto01(X):
    minval, maxval = X.min(), X.max()
    X -= minval
    X /= (maxval-minval)

def contrastadjust(X,epsilon):
    """
    Assumes that the values are in [0,255]
    """
    X /= 255.
    X += epsilon
    log(X, X)
    X -= log(epsilon)
    X /= (log(1+epsilon)-log(epsilon))
    X *= 255

def contrastadjust0(X,n):
    """
    Assumes that the values are in [0,255]
    """
    X /= 255.
    X -= 1/2.
    X *= n
    arctan(X, X)
    X /= (2*arctan(n/2.))
    X += 1/2.0
    X *= 255 # Maps the values back to [0,255]
```

Several examples of using this function will be shown below. A good question here is why the functions `min` and `max` are called three times in succession. The reason is that there is a third "dimension" in play, besides the spatial x- and y-directions. This dimension describes the coulor components in each pixel, which are usually the red-, green-, and blue color components.

Example 9.9. *Extracting the different colors.*

If we have a color image

$$P = (r_{i,j}, g_{i,j}, b_{i,j})_{i,j=1}^{m,n},$$

it is often useful to manipulate the three color components separately as the three images

$$P_r = (r_{i,j})_{i,j=1}^{m,n}, \quad P_r = (g_{i,j})_{i,j=1}^{m,n}, \quad P_r = (b_{i,j})_{i,j=1}^{m,n}.$$

As an example, let us first see how we can produce three separate images, showing the R,G, and B color components, respectively. Let us take the image `lena.png` used in Figure 9.1. When the image is read (first line below), the returned object has three dimensions. The first two dimensions represent the spatial

directions (the row-index and column-index). The third dimension represents the color component. One can therefore view images representing the different color components with the help of the following code:

```
X1 = zeros_like(img)
X1[:, :, 0] = img[:, :, 0]

X2 = zeros_like(img)
X2[:, :, 1] = img[:, :, 1]

X3 = zeros_like(img)
X3[:, :, 2] = img[:, :, 2]
```

The resulting images are shown in Figure 9.5.

Figure 9.5: The red, green, and blue components of the color image in Figure 9.1.

Example 9.10. *Converting from color to grey-level.*

If we have a color image we can convert it to a grey-level image. This means that at each point in the image we have to replace the three color values (r, g, b) by a single value p that will represent the grey level. If we want the grey-level image to be a reasonable representation of the color image, the value p should somehow reflect the intensity of the image at the point. There are several ways to do this.

It is not unreasonable to use the largest of the three color components as a measure of the intensity, i.e, to set $p = \max(r, g, b)$. The result of this can be seen in the left image of Figure 9.6.

An alternative is to use the sum of the three values as a measure of the total intensity at the point. This corresponds to setting $p = r + g + b$. Here we have to be a bit careful with a subtle point. We have required each of the r, g and b values to lie in the range $[0, 1]$, but their sum may of course become as large as 3. We also require our grey-level values to lie in the range $[0, 1]$ so after having computed all the sums we must normalise as explained above. The result can be seen in the middle images of Figure 9.6.

A third possibility is to think of the intensity of (r, g, b) as the length of the color vector, in analogy with points in space, and set $p = \sqrt{r^2 + g^2 + b^2}$. Again,

we may end up with values in the range $[0, \sqrt{3}]$ so we have to normalise like we did in the second case. The result is shown in the right image of Figure 9.6.

Let us sum this up as follows: A color image $P = (r_{i,j}, g_{i,j}, b_{i,j})_{i,j=1}^{m,n}$ can be converted to a grey level image $Q = (q_{i,j})_{i,j=1}^{m,n}$ by one of the following three operations:

- Set $q_{i,j} = \max(r_{i,j}, g_{i,j}, b_{i,j})$ for all i and j.
- Compute $\hat{q}_{i,j} = r_{i,j} + g_{i,j} + b_{i,j}$ for all i and j.
- Transform all the values to the interval $[0, 1]$ by setting

$$q_{i,j} = \frac{\hat{q}_{i,j}}{\max_{k,l} \hat{q}_{k,l}}.$$

- Compute $\hat{q}_{i,j} = \sqrt{r_{i,j}^2 + g_{i,j}^2 + b_{i,j}^2}$ for all i and j.
- Transform all the values to the interval $[0, 1]$ by setting

$$q_{i,j} = \frac{\hat{q}_{i,j}}{\max_{k,l} \hat{q}_{k,l}}.$$

If `img` is an $M \times N$ image, this can be implemented by using most of the code from the previous example, and replacing with the lines

```
mx = maximum(img[:, :, 0], img[:, :, 1])
X1 = maximum(img[:, :, 2], mx)

X2 = img[:, :, 0] + img[ :, :, 1] + img[ :, :, 2]
mapto01(X2); X2 *= 255

X3 = sqrt(img[:,:,0]**2 + img[:, :, 1]**2 + img[:, :, 2]**2)
mapto01(X3); X3 *= 255
```

respectively. In practice one of the last two methods are usually preferred, perhaps with a preference for the last method, but the actual choice depends on the application.

Example 9.11. *Computing the negative image.*

In film-based photography a negative image was obtained when the film was developed, and then a positive image was created from the negative. We can easily simulate this and compute a negative digital image.

Suppose we have a grey-level image $P = (p_{i,j})_{i,j=1}^{m,n}$ with intensity values in the interval $[0, 1]$. Here intensity value 0 corresponds to black and 1 corresponds to white. To obtain the negative image we just have to replace an intensity p by its 'mirror value' $1 - p$. This is also easily translated to code as above. The resulting image is shown in Figure 9.7.

Figure 9.6: Alternative ways to convert the color image in Figure 9.1 to a grey level image. The result is mapped to $(0, 1)$.

Figure 9.7: The negative versions of the corresponding images in Figure 9.6.

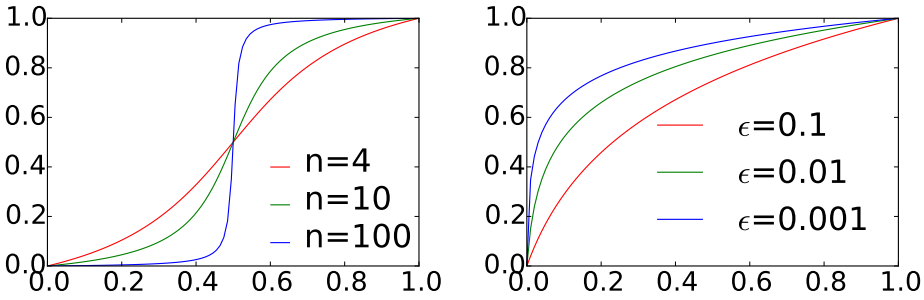

Figure 9.8: Some functions that can be used to improve the contrast of an image.

Example 9.12. *Increasing the contrast.*

A common problem with images is that the contrast often is not good enough. This typically means that a large proportion of the grey values are concentrated in a rather small subinterval of $[0, 1]$. The obvious solution to this problem is to somehow spread out the values. This can be accomplished by applying a function f to the intensity values, i.e., new intensity values are computed by the

Figure 9.9: The middle functions in Figure 9.8 have been applied to a grey-level version of the test image.

formula

$$\hat{p}_{i,j} = f(p_{i,j})$$

for all i and j. If we choose f so that its derivative is large in the area where many intensity values are concentrated, we obtain the desired effect.

Figure 9.8 shows some examples. The functions in the left plot have quite large derivatives near $x = 0.5$ and will therefore increase the contrast in images with a concentration of intensities with value around 0.5. The functions are all on the form

$$f_n(x) = \frac{\arctan\bigl(n(x - 1/2)\bigr)}{2\arctan(n/2)} + \frac{1}{2}. \tag{9.1}$$

For any $n \neq 0$ these functions satisfy the conditions $f_n(0) = 0$ and $f_n(1) = 1$. The three functions in the left plot in Figure 9.8 correspond to $n = 4$, 10, and 100. In the left plot in Figure 9.9 the middle function has been applied to the image in Figure 9.6(c). Since the image was quite well balanced, this has made the dark areas too dark and the bright areas too bright.

Functions of the kind shown in the right plot have a large derivative near $x = 0$ and will therefore increase the contrast in an image with a large proportion of small intensity values, i.e., very dark images. These functions are given by

$$g_\epsilon(x) = \frac{\ln(x + \epsilon) - \ln \epsilon}{\ln(1 + \epsilon) - \ln \epsilon}, \tag{9.2}$$

and the ones shown in the plot correspond to $\epsilon = 0.1$, 0.01, and 0.001. In the right plot in Figure 9.9 the middle function has been applied to the same image. This has made the image as a whole too bright, but has brought out the details of the road which was very dark in the original.

Increasing the contrast is easy to implement. The following function uses the contrast adjusting function from Equation (9.2), with ϵ as in that equation as parameter

```
def contrastadjust(X,epsilon):
    """
    Assumes that the values are in [0,255]
    """
    X /= 255.
    X += epsilon
    log(X, X)
    X -= log(epsilon)
    X /= (log(1+epsilon)-log(epsilon))
    X *= 255

def contrastadjust0(X,n):
    """
    Assumes that the values are in [0,255]
    """
    X /= 255.
    X -= 1/2.
    X *= n
    arctan(X, X)
    X /= (2*arctan(n/2.))
    X += 1/2.0
    X *= 255 # Maps the values back to [0,255]
```

This has been used to generate the right image in Figure 9.9.

What you should have learned in this section.

- How to read, write, and show images on your computer.
- How to extract different color components.
- How to convert from color to grey-level images.
- How to use functions for adjusting the contrast.

Exercise 9.1: Generate black and white images

Black and white images can be generated from greyscale images (with values between 0 and 255) by replacing each pixel value with the one of 0 and 255 which is closest. Use this strategy to generate the black and white image shown in Figure 9.2(b).

Exercise 9.2: Adjust contrast in images 1

Generate the right image in Figure 9.9 on your own by writing code which uses the function `contrastadjust`.

Exercise 9.3: Adjust contrast in images 2

Let us also consider the second way we mentioned for increasing the contrast.

a) Write a function `contrastadjust0` which instead uses the function from Equation (9.1) in the compendium to increase the contrast. n should be a parameter to the function.

b) Generate the left image in Figure 9.9 on your own by using your code from Exercise 9.2, and instead calling the function `contrastadjust0`.

Exercise 9.4: Adjust contrast in images 3

In this exercise we will look at another function for increasing the contrast of a picture.

a) Show that the function $f : \mathbb{R} \to \mathbb{R}$ given by

$$f_n(x) = x^n,$$

for all n maps the interval $[0, 1] \to [0, 1]$, and that $f'(1) \to \infty$ as $n \to \infty$.

b) The color image `secret.jpg`,shown in Figure 9.10, contains some information that is nearly invisible to the naked eye on most computer monitors. Use the function $f(x)$, to reveal the secret message.

Figure 9.10: Secret message.

Hint. You will first need to convert the image to a greyscale image. You can then use the function `contrastadjust` as a starting point for your own program.

9.3 Filter-based operations on images

The next examples of operations on images we consider will use filters. These examples define what it means to apply a filter to two-dimensional data. We start with the following definition of a computational molecule. This term stems from image processing, and seems at the outset to be unrelated to filters.

Definition 9.13. *Computational molecules.*

We say that an operation S on an image X is given by the *computational molecule*

$$A = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & a_{-1,-1} & a_{-1,0} & a_{-1,1} & \cdots \\ \cdots & a_{0,-1} & \underline{a_{0,0}} & a_{0,1} & \cdots \\ \cdots & a_{1,-1} & a_{1,0} & a_{1,1} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

if we have that

$$(SX)_{i,j} = \sum_{k_1,k_2} a_{k_1,k_2} X_{i-k_1,j-k_2}. \tag{9.3}$$

In the molecule, indices are allowed to be both positive and negative, we underline the element with index $(0,0)$ (the center of the molecule), and assume that $a_{i,j}$ with indices falling outside those listed in the molecule are zero (as for compact filter notation).

In Equation (9.3), it is possible for the indices $i-k_1$ and $j-k_2$ to fall outside the legal range for X. We will solve this case in the same way as we did for filters, namely that we assume that X is extended (either periodically or symmetrically) in both directions. The interpretation of a computational molecule is that we place the center of the molecule on a pixel, multiply the pixel and its neighbors by the corresponding weights $a_{i,j}$ in reverse order, and finally sum up in order to produce the resulting value. This type of operation will turn out to be particularly useful for images. The following result expresses how computational molecules and filters are related. It states that, if we apply one filter to all the columns, and then another filter to all the rows, the end result can be expressed with the help of a computational molecule.

Theorem 9.14. *Filtering and computational molecules.*

Let S_1 and S_2 be filters with compact filter notation $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$, respectively, and consider the operation S where S_1 is first applied to the columns in the image, and then S_2 is applied to the rows in the image. Then S is an operation which can be expressed in terms of the computational molecule $a_{i,j} = (\boldsymbol{t}_1)_i(\boldsymbol{t}_2)_j$.

Proof. Let $X_{i,j}$ be the pixels in the image. When we apply S_1 to the columns of X we get the image Y defined by

$$Y_{i,j} = \sum_{k_1} \sum (t_1)_{k_1} X_{i-k_1,j}.$$

When we apply S_2 to the rows of Y we get the image Z defined by

$$\begin{aligned} Z_{i,j} &= \sum_{k_2} (t_2)_{k_2} Y_{i,j-k_2} = \sum_{k_2} (t_2)_{k_2} \sum_{k_1} (t_1)_{k_1} X_{i-k_1,j-k_2} \\ &= \sum_{k_1} \sum_{k_2} (t_1)_{k_1} (t_2)_{k_2} X_{i-k_1,j-k_2}. \end{aligned}$$

Comparing with Equation (9.3) we see that S is given by the computational molecule with entries $a_{i,j} = (\boldsymbol{t}_1)_i (\boldsymbol{t}_2)_j$. □

Note that, when we filter an image with S_1 and S_2 in this way, the order does not matter: since computing $S_1 X$ is the same as applying S_1 to all columns of X, and computing $Y(S_2)^T$ is the same as applying S_2 to all rows of Y, the combined filtering operation, denoted S, takes the form

$$S(X) = S_1 X (S_2)^T, \tag{9.4}$$

and the fact that the order does not matter simply boils down to the fact that it does not matter which of the left or right multiplications we perform first. Applying S_1 to the columns of X is what we call a *vertical filtering operation*, while applying S_2 to the rows of X is what we call a *horizontal filtering operation*. We can thus state the following.

Observation 9.15. *Order of vertical and horizontal filtering.*
The order of vertical and horizontal filtering of an image does not matter.

Most computational molecules we will consider in the following can be expressed in terms of filters as in this theorem, but clearly there exist also computational molecules which are not on this form, since the matrix A with entries $a_{i,j} = (\boldsymbol{t}_1)_i (\boldsymbol{t}_2)_j$ has rank one, and a general computational molecule can have any rank. In most of the examples the filters are symmetric.

Assume that the image is stored as the matrix `X`. In Exercise 9.5 you will be asked to implement a function `tensor_impl` which computes the transformation $S(X) = S_1 X (S_2)^T$, where `X`, `S1`, and `S2` are input. If the computational molecule is obtained by applying the filter S_1 to the columns, and the filter S_2 to the rows, we can compute it with the following code: (we have assumed that the filter lengths are odd, and that the middle filter coefficient has index 0):

```
def S1(x):
    filterS(S1, x, True)

def S2(x):
    filterS(S2, x, True)

tensor_impl(X, S1, S2)
```

We have here used the function `filterS` to implement the filtering, so that we assume that the image is periodically or symmetrically extended. The above code uses symmetric extension, and can thus be used for symmetric filters. If the filter is non-symmetric, we should use a periodic extension instead, for which the last parameter to `filterS` should be changed.

9.3.1 Tensor product notation for operations on images

Filter-based operations on images can be written compactly using what we will call *tensor product notation*. This is part of a very general tensor product framework, and we will review parts of this framework for the sake of completeness. Let us first define the tensor product of vectors.

Definition 9.16. *Tensor product of vectors.*
If $\boldsymbol{x}, \boldsymbol{y}$ are vectors of length M and N, respectively, their tensor product $\boldsymbol{x} \otimes \boldsymbol{y}$ is defined as the $M \times N$-matrix defined by $(\boldsymbol{x} \otimes \boldsymbol{y})_{i,j} = x_i y_j$. In other words, $\boldsymbol{x} \otimes \boldsymbol{y} = \boldsymbol{x}\boldsymbol{y}^T$.

The tensor product $\boldsymbol{x}\boldsymbol{y}^T$ is also called the *outer product* of $\boldsymbol{x}$ and $\boldsymbol{y}$ (contrary to the inner product $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x}^T \boldsymbol{y}$). In particular $\boldsymbol{x} \otimes \boldsymbol{y}$ is a matrix of rank 1, which means that most matrices cannot be written as a tensor product of two vectors. The special case $\boldsymbol{e}_i \otimes \boldsymbol{e}_j$ is the matrix which is 1 at (i, j) and 0 elsewhere, and the set of all such matrices forms a basis for the set of $M \times N$-matrices.

Observation 9.17. *Standard basis for $L_{M,N}(\mathbb{R})$.*
Let $\mathcal{E}_M = \{\boldsymbol{e}_i\}_{i=0}^{M-1}$ $\mathcal{E}_N = \{\boldsymbol{e}_i\}_{i=0}^{N-1}$ be the standard bases for $\mathbb{R}^M$ and $\mathbb{R}^N$. Then

$$\mathcal{E}_{M,N} = \{\boldsymbol{e}_i \otimes \boldsymbol{e}_j\}_{(i,j)=(0,0)}^{(M-1,N-1)}$$

is a basis for $L_{M,N}(\mathbb{R})$, the set of $M \times N$-matrices. This basis is often referred to as the standard basis for $L_{M,N}(\mathbb{R})$.

The standard basis thus consists of rank 1-matrices. An image can simply be thought of as a matrix in $L_{M,N}(\mathbb{R})$, and a computational molecule is simply a special type of linear transformation from $L_{M,N}(\mathbb{R})$ to itself. Let us also define the tensor product of matrices.

Definition 9.18. *Tensor product of matrices.*
If $S_1 : \mathbb{R}^M \to \mathbb{R}^M$ and $S_2 : \mathbb{R}^N \to \mathbb{R}^N$ are matrices, we define the linear mapping $S_1 \otimes S_2 : L_{M,N}(\mathbb{R}) \to L_{M,N}(\mathbb{R})$ by linear extension of $(S_1 \otimes S_2)(\boldsymbol{e}_i \otimes \boldsymbol{e}_j) = (S_1 \boldsymbol{e}_i) \otimes (S_2 \boldsymbol{e}_j)$. The linear mapping $S_1 \otimes S_2$ is called the tensor product of the matrices S_1 and S_2.

A couple of remarks are in order. First, from linear algebra we know that, when S is linear mapping from V and $S(\boldsymbol{v}_i)$ is known for a basis $\{\boldsymbol{v}_i\}_i$ of V, S is uniquely determined. In particular, since the $\{\boldsymbol{e}_i \otimes \boldsymbol{e}_j\}_{i,j}$ form a basis, there exists a unique linear transformation $S_1 \otimes S_2$ so that $(S_1 \otimes S_2)(\boldsymbol{e}_i \otimes \boldsymbol{e}_j) = (S_1 \boldsymbol{e}_i) \otimes (S_2 \boldsymbol{e}_j)$.

This unique linear transformation is what we call the linear extension from the values in the given basis. Clearly, by linearity, also $(S_1 \otimes S_2)(\boldsymbol{x} \otimes \boldsymbol{y}) = (S_1\boldsymbol{x}) \otimes (S_2\boldsymbol{y})$, since

$$\begin{aligned}(S_1 \otimes S_2)(\boldsymbol{x} \otimes \boldsymbol{y}) &= (S_1 \otimes S_2)((\sum_i x_i\boldsymbol{e}_i) \otimes (\sum_j y_j\boldsymbol{e}_j)) = (S_1 \otimes S_2)(\sum_{i,j} x_iy_j(\boldsymbol{e}_i \otimes \boldsymbol{e}_j)) \\ &= \sum_{i,j} x_iy_j(S_1 \otimes S_2)(\boldsymbol{e}_i \otimes \boldsymbol{e}_j) = \sum_{i,j} x_iy_j(S_1\boldsymbol{e}_i) \otimes (S_2\boldsymbol{e}_j) \\ &= \sum_{i,j} x_iy_jS_1\boldsymbol{e}_i((S_2\boldsymbol{e}_j))^T = S_1(\sum_i x_i\boldsymbol{e}_i)(S_2(\sum_j y_j\boldsymbol{e}_j))^T \\ &= S_1\boldsymbol{x}(S_2\boldsymbol{y})^T = (S_1\boldsymbol{x}) \otimes (S_2\boldsymbol{y}).\end{aligned}$$

Here we used the result from Exercise 9.9. We can now prove the following.

Theorem 9.19. *Comact filter notationa and computational molecules.*

If $S_1 : \mathbb{R}^M \to \mathbb{R}^M$ and $S_2 : \mathbb{R}^N \to \mathbb{R}^N$ are matrices of linear transformations, then $(S_1 \otimes S_2)X = S_1X(S_2)^T$ for any $X \in L_{M,N}(\mathbb{R})$. In particular $S_1 \otimes S_2$ is the operation which applies S_1 to the columns of X, and S_2 to the resulting rows. In other words, if S_1, S_2 have compact filter notations $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$, respectively, then $S_1 \otimes S_2$ has computational molecule $\boldsymbol{t}_1 \otimes \boldsymbol{t}_2$.

We have not formally defined the tensor product of compact filter notations. This is a straightforward extension of the usual tensor product of vectors, where we additionally mark the element at index $(0, 0)$.

Proof. We have that

$$\begin{aligned}(S_1 \otimes S_2)(\boldsymbol{e}_i \otimes \boldsymbol{e}_j) &= (S_1\boldsymbol{e}_i) \otimes (S_2\boldsymbol{e}_j) \\ &= (\text{col}_i(S_1)) \otimes (\text{col}_j(S_2)) = \text{col}_i(S_1)(\text{col}_j(S_2))^T \\ &= \text{col}_i(S_1)\text{row}_j((S_2)^T) = S_1(\boldsymbol{e}_i \otimes \boldsymbol{e}_j)(S_2)^T.\end{aligned}$$

This means that $(S_1 \otimes S_2)X = S_1X(S_2)^T$ for any $X \in L_{M,N}(\mathbb{R})$ also, since equality holds on the basis vectors $\boldsymbol{e}_i \otimes \boldsymbol{e}_j$. Since the matrix A with entries $a_{i,j} = (\boldsymbol{t}_1)_i(\boldsymbol{t}_2)_j$ also can be written as $\boldsymbol{t}_1 \otimes \boldsymbol{t}_2$, the result follows. □

We have thus shown that we alternatively can write $S_1 \otimes S_2$ for the operations we have considered. This notation also makes it easy to combine several two-dimensional filtering operations:

Corollary 9.20. *Composing tensor products.*

We have that $(S_1 \otimes T_1)(S_2 \otimes T_2) = (S_1S_2) \otimes (T_1T_2)$.

Proof. By Theorem 9.19 we have that

$$(S_1 \otimes T_1)(S_2 \otimes T_2)X = S_1(S_2XT_2^T)T_1^T = (S_1S_2)X(T_1T_2)^T = ((S_1S_2) \otimes (T_1T_2))X.$$

for any $X \in L_{M,N}(\mathbb{R})$. This proves the result. □

Suppose that we want to apply the operation $S_1 \otimes S_2$ to an image. We can factorize $S_1 \otimes S_2$ as

$$S_1 \otimes S_2 = (S_1 \otimes I)(I \otimes S_2) = (I \otimes S_2)(S_1 \otimes I). \tag{9.5}$$

Moreover, since

$$(S_1 \otimes I)X = S_1 X \qquad (I \otimes S_2)X = X(S_2)^T = (S_2 X^T)^T,$$

$S_1 \otimes I$ is a vertical filtering operation, and $I \otimes S_2$ is a horizontal filtering operation in this factorization. For filters we have an even stronger result: If S_1, S_2, S_3, S_4 all are filters, we have from Corollary 9.20 that $(S_1 \otimes S_2)(S_3 \otimes S_4) = (S_3 \otimes S_4)(S_1 \otimes S_2)$, since all filters commute. This does not hold in general since general matrices do not commute.

We will now consider two important examples of filtering operations on images: smoothing and edge detection/computing partical derivatives. For all examples we will use the tensor product notation for these operations.

Example 9.21. *Smoothing an image.*

When we considered filtering of digital sound, we observed that replacing each sample of a sound by an average of the sample and its neighbours dampened the high frequencies of the sound. Let us consider the computational molecules where such a filter is applied to both the rows and the columns. For the one-dimensional case on sound, we argued that filter coefficients taken from Pascal's triangle give good smoothing effects. The same can be argued for images. If we use the filter $S = \frac{1}{4}\{1, \underline{2}, 1\}$ (row 2 from Pascal's triangle), Theorem 9.14 says that we obtain the computational molecule

$$A = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & \underline{4} & 2 \\ 1 & 2 & 1 \end{pmatrix}. \tag{9.6}$$

This means that we compute the new pixels by

$$\hat{p}_{i,j} = \frac{1}{16}\big(4p_{i,j} + 2(p_{i,j-1} + p_{i-1,j} + p_{i+1,j} + p_{i,j+1}) + p_{i-1,j-1} + p_{i+1,j-1} + p_{i-1,j+1} + p_{i+1,j+1}\big).$$

If we instead use the filter $S = \frac{1}{64}\{1, 6, 15, \underline{20}, 15, 6, 1\}$ (row 6 from Pascal's triangle), we get the computational molecule

$$\frac{1}{4096}\begin{pmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & \underline{400} & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{pmatrix}. \tag{9.7}$$

For both molecules the weights sum to one, so that the new intensity values $\hat{p}_{i,j}$ are weighted averages of the intensity values on the right. We anticipate that both molecules give a smoothing effect, but that the second molecules provides more smoothing. The result of applying the two molecules in (9.6) and (9.7) to our greyscale-image is shown in the two right images in Figure 9.11. With the help of the function `tensor_impl`, smoothing with the first molecule (9.6) above can be obtained by writing

```
def S(x):
    filterS([1., 2., 1.]/4., x, True);

tensor_impl(X, S, S)
```

To make the smoothing effect visible, we have zoomed in on the face in the image. The smoothing effect is clarly best visible in the second image.

Figure 9.11: The two right images show the effect of smoothing the left image.

Smoothing effects are perhaps more visible if we use a simple image, as the one in the left part of Figure 9.12.

Figure 9.12: The results of smoothing the simple image to the left with the filter $\frac{1}{4}\{1,\underline{2},1\}$ horizontally, vertically, and both, respectively.

Again we have used the filter $S = \frac{1}{4}\{1,\underline{2},1\}$. Here we also have shown what happens if we only smooth the image in one of the directions. The smoothing effects are then only seen in one of the vertical or horizontal directions. In the right image we have smoothed in both directions. We clearly see the union of the two one-dimensional smoothing operations then.

Let us summarize from this example as follows.

Observation 9.22. *Smoothing an image.*
An image P can be smoothed by applying a smoothing filter to the rows, and then to the columns.

Another operation on images which can be expressed in terms of computational molecules is edge detection. An edge in an image is characterised by a large change in intensity values over a small distance in the image. For a continuous function this corresponds to a large derivative. An image is only defined at isolated points, so we cannot compute derivatives, but we have a perfect situation for applying numerical differentiation. Since a grey-level image is a scalar function of two variables, numerical differentiation techniques can be applied.

Partial derivative in x-direction. Let us first consider computation of the partial derivative $\partial P/\partial x$ at all points in the image. Note first that it is the second coordinate in an image which refers to the x-direction used when plotting functions. This means that the familiar symmetric Newton quotient approximation for the partial derivative [23] takes the form

$$\frac{\partial P}{\partial x}(i,j) \approx \frac{p_{i,j+1} - p_{i,j-1}}{2}, \tag{9.8}$$

where we have used the convention $h = 1$ which means that the derivative is measured in terms of 'intensity per pixel'. This corresponds to applying the bass-reducing filter $S = \frac{1}{2}\{1, \underline{0}, -1\}$ to all the rows (alternatively, applying the tensor product $I \otimes S$ to the image). We can thus express this in terms of computational molecules as follows.

Observation 9.23. *The partial derivative $\partial P/\partial x$.*
Let $P = (p_{i,j})_{i,j=1}^{m,n}$ be a given image. The partial derivative $\partial P/\partial x$ of the image can be computed with the computational molecule

$$\frac{1}{2}\begin{pmatrix} 0 & 0 & 0 \\ 1 & \underline{0} & -1 \\ 0 & 0 & 0 \end{pmatrix}. \tag{9.9}$$

We have included the two rows of 0s just to make it clear how the computational molecule is to be interpreted when we place it over the pixels. If we apply the `smooth`-function to the same excerpt of the Lena image with this molecule, we obtain the left image in Figure 9.13. It shows many artefacts since the pixel values lie outside the legal range: many of the intensities are in fact negative. More specifically, the intensities turn out to vary in the interval $[-0.424, 0.418]$. We therefore normalise and map all intensities to $[0, 1]$. The result of this is shown in the middle image. The predominant color of this image is an average grey, i.e, an intensity of about 0.5. To get more detail in the image we therefore try to increase the contrast by applying the function f_{50} in equation (9.1) to

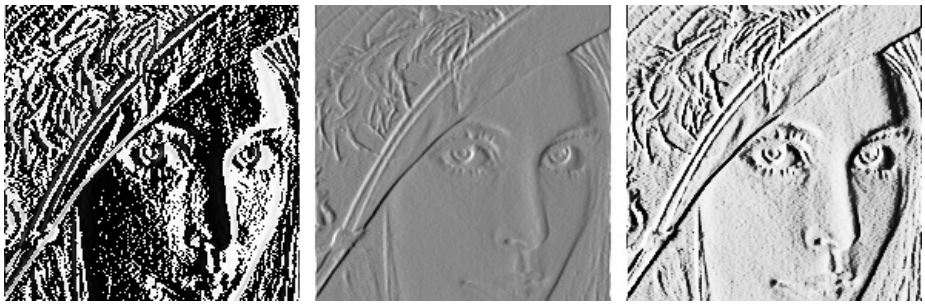

Figure 9.13: Experimenting with the partial derivative in the x-direction for the image in 9.6. The left image has artefacts, since the pixel values are outside the legal range. We therefore normalize the intensities to lie in $[0, 25]$ (middle), before we increase the contrast (right).

each intensity value. The result is shown in the right image in Figure 9.13 which does indeed show more detail.

It is important to understand the colors in these images. We have computed the derivative in the x-direction, and we recall that the computed values varied in the interval $[-0.424, 0.418]$. The negative value corresponds to the largest average decrease in intensity from a pixel $p_{i-1,j}$ to a pixel $p_{i+1,j}$. The positive value on the other hand corresponds to the largest average increase in intensity. A value of 0 in the left image in Figure 9.13 corresponds to no change in intensity between the two pixels.

When the values are mapped to the interval $[0, 1]$ in the middle image in Figure 9.13, the small values are mapped to something close to 0 (almost black), the maximal values are mapped to something close to 1 (almost white), and the values near 0 are mapped to something close to 0.5 (grey). In the right image in Figure 9.13 these values have just been emphasised even more.

The right image in Figure 9.13 tells us that in large parts of the image there is very little variation in the intensity. However, there are some small areas where the intensity changes quite abruptly, and if you look carefully you will notice that in these areas there is typically both black and white pixels close together, like down the vertical front corner of the bus. This will happen when there is a stripe of bright or dark pixels that cut through an area of otherwise quite uniform intensity.

Partial derivative in y-direction. The partial derivative $\partial P/\partial y$ can be computed analogously to $\partial P/\partial x$, i.e. we apply the filter $-S = \frac{1}{2}\{-1, \underline{0}, 1\}$ to all columns of the image (alternatively, apply the tensor product $-S \otimes I$ to the image), where S is the filter which we used for edge detection in the x-direction. Note that the positive direction of this axis in an image is opposite to the direction of the y-axis we use when plotting functions.

Observation 9.24. *The partial derivative $\partial P/\partial y$.*

Let $P = (p_{i,j})_{i,j=1}^{m,n}$ be a given image. The partial derivative $\partial P/\partial y$ of the image can be computed with the computational molecule

$$\frac{1}{2}\begin{pmatrix} 0 & 1 & 0 \\ 0 & \underline{0} & 0 \\ 0 & -1 & 0 \end{pmatrix}. \tag{9.10}$$

The result is shown in Figure 9.15(b). The intensities have been normalised and the contrast enhanced by the function f_{50} from Equation (9.1).

The gradient. The gradient of a scalar function is often used as a measure of the size of the first derivative. The gradient is defined by the vector

$$\nabla P = \left(\frac{\partial P}{\partial x}, \frac{\partial P}{\partial y}\right),$$

so its length is given by

$$|\nabla P| = \sqrt{\left(\frac{\partial P}{\partial x}\right)^2 + \left(\frac{\partial P}{\partial y}\right)^2}.$$

When the two first derivatives have been computed it is a simple matter to compute the gradient vector and its length; the resulting is shown as an image in Figure 9.14c.

Figure 9.14: The computed gradient (left). In the middle the intensities have been normalised to the $[0, 255]$, and to the right the contrast has been increased.

The image of the gradient looks quite different from the images of the two partial derivatives. The reason is that the numbers that represent the length of the gradient are (square roots of) sums of squares of numbers. This means that the parts of the image that have virtually constant intensity (partial derivatives close to 0) are colored black. In the images of the partial derivatives these values ended up in the middle of the range of intensity values, with a final color of grey, since there were both positive and negative values.

The left image in Figure 9.14 shows the computed values of the gradient. Note that it is possible that the length of the gradient could be outside the legal range of values. This would have been seen as artefacts in this image. In the middle image the intensities have been mapped to the legal range. To enhance the contrast further we have to do something different from what was done in the other images since we now have a large number of intensities near 0. The solution is to apply a function like the ones shown in the right plot in Figure 9.8 to the intensities. If we use the function $g_{0.01}$ defined in equation(9.2) we obtain the right image in Figure 9.14.

Figure 9.15: The first-order partial derivatives in the x- and y-direction, respectively. In both images, the computed numbers have been normalised and the contrast enhanced.

9.3.2 Comparing the first derivatives

Figure 9.15 shows the two first-order partial derivatives and the gradient. If we compare the two partial derivatives we see that the x-derivative seems to emphasise vertical edges while the y-derivative seems to emphasise horizontal edges. This is precisely what we must expect. The x-derivative is large when the difference between neighbouring pixels in the x-direction is large, which is the case across a vertical edge. The y-derivative enhances horizontal edges for a similar reason.

The gradient contains information about both derivatives and therefore emphasises edges in all directions. It also gives a simpler image since the sign of the derivatives has been removed.

9.3.3 Second-order derivatives

To compute the three second order derivatives we can combine the two computational molecules which we already have described. For the mixed second order derivative we get $(I \otimes S)((-S) \otimes I) = -S \otimes S$. For the last two second order derivative $\frac{\partial^2 P}{\partial x^2}$, $\frac{\partial^2 P}{\partial y^2}$, we can also use the three point approximation to the second derivative [23]

$$\frac{\partial P}{\partial x^2}(i,j) \approx p_{i,j+1} - 2p_{i,j} + p_{i,j-1} \tag{9.11}$$

to the second derivative (again we have set $h = 1$). This gives a smaller molecule than if we combine the two molecules for order one differentiation (i.e. $(I \otimes S)(I \otimes S) = (I \otimes S^2)$ and $((-S) \otimes I)((-S) \otimes I) = (S^2 \otimes I)$), since $S^2 = \frac{1}{2}\{1, \underline{0}, -1\}\frac{1}{2}\{1, \underline{0}, -1\} = \frac{1}{4}\{1, 0, \underline{-2}, 0, 1\}$.

Observation 9.25. *Second order derivatives of an image.*

The second order derivatives of an image P can be computed by applying the computational molecules

$$\frac{\partial^2 P}{\partial x^2}: \quad \begin{pmatrix} 0 & 0 & 0 \\ 1 & \underline{-2} & 1 \\ 0 & 0 & 0 \end{pmatrix}, \tag{9.12}$$

$$\frac{\partial^2 P}{\partial y \partial x}: \quad \frac{1}{4}\begin{pmatrix} -1 & 0 & 1 \\ 0 & \underline{0} & 0 \\ 1 & 0 & -1 \end{pmatrix}, \tag{9.13}$$

$$\frac{\partial^2 P}{\partial y^2}: \quad \begin{pmatrix} 0 & 1 & 0 \\ 0 & \underline{-2} & 0 \\ 0 & 1 & 0 \end{pmatrix}. \tag{9.14}$$

Figure 9.16: The second-order partial derivatives in the xx-, xy-, and yy-directions, respectively. In all images, the computed numbers have been normalised and the contrast enhanced.

With the information in Observation 9.25 it is quite easy to compute the second-order derivatives, and the results are shown in Figure 9.16. The computed

derivatives were first normalised and then the contrast enhanced with the function f_{100} in each image, see equation (9.1).

As for the first derivatives, the xx-derivative seems to emphasise vertical edges and the yy-derivative horizontal edges. However, we also see that the second derivatives are more sensitive to noise in the image (the areas of grey are less uniform). The mixed derivative behaves a bit differently from the other two, and not surprisingly it seems to pick up both horizontal and vertical edges.

This procedure can be generalized to higher order derivatives also. To apply $\frac{\partial^{k+l}P}{\partial x^k \partial y^l}$ to an image we can compute $S_l \otimes S_k$ where S_r corresponds to any point method for computing the r'th order derivative. We can also compute $(S^l)\otimes(S^k)$, where we iterate the filter $S = \frac{1}{2}\{1, \underline{0}, -1\}$ for the first derivative, but this gives longer filters.

Let us also apply the molecules for differentiation to a chess pattern test image. In Figure 9.17 we have applied $S \otimes I$, $I \otimes S$, and $S \otimes S$, $I \otimes S^2$, and $S^2 \otimes I$ to the example image shown in the upper left.

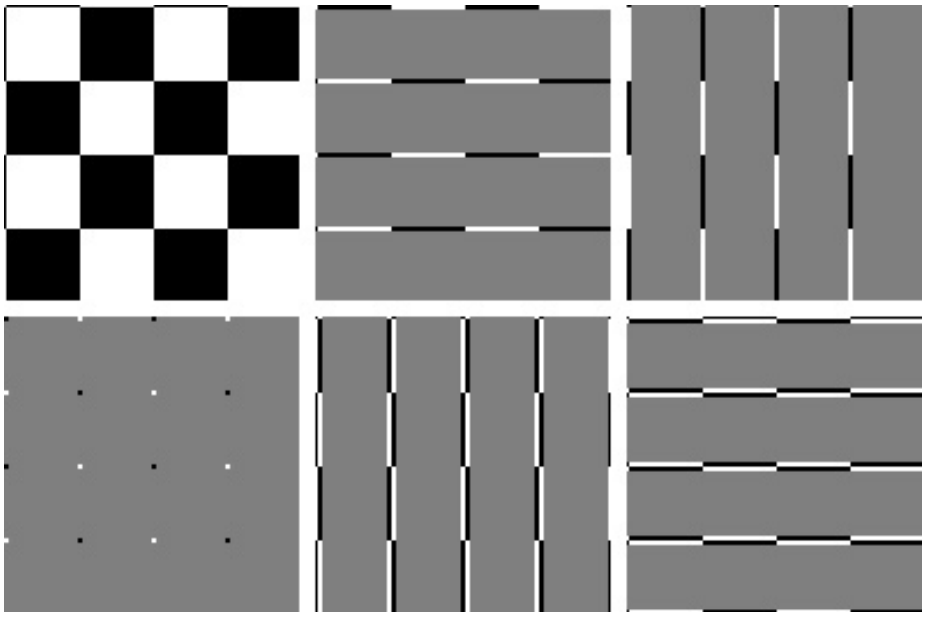

Figure 9.17: Different tensor products representing partial derivatives applied to a simple chess pattern example image (upper left). The tensor products are $S \otimes I$, $I \otimes S$, $S \otimes S$, $I \otimes S^2$, and $S^2 \otimes I$.

These images make it is clear that $S \otimes I$ detects all horizontal edges, that $I \otimes S$ detects all vertical edges, and that $S \otimes S$ detects all points where abrupt changes appear in both directions. We also see that the second order partial derivative detects exactly the same edges which the first order partial derivative found. Note that the edges detected with $I \otimes S^2$ are wider than the ones detected with $I \otimes S$. The reason is that the filter S^2 has more filter coefficients than

S. Also, edges are detected with different colors. This reflects whether the difference between the neighbouring pixels is positive or negative. The values after we have applied the tensor product may thus not lie in the legal range of pixel values (since they may be negative). The figures have taken this into account by mapping the values back to a legal range of values, as we did in Chapter 9. Finally, we also see additional edges at the first and last rows/edges in the images. The reason is that the filter S is defined by assuming that the pixels repeat periodically (i.e. it is a circulant Toeplitz matrix). Due to this, we have additional edges at the first/last rows/edges. This effect can also be seen in Chapter 9, although there we did not assume that the pixels repeat periodically.

Defining a two-dimensional filter by filtering columns and then rows is not the only way we can define a two-dimensional filter. Another possible way is to let the $MN \times MN$-matrix itself be a filter. Unfortunately, this is a bad way to define filtering of an image, since there are some undesirable effects near the boundaries between rows: in the vector we form, the last element of one row is followed by the first element of the next row. These boundary effects are unfortunate when a filter is applied.

What you should have learned in this section.

- The operation $X \to S_1 X (S_2)^T$ can be used to define operations on images, based on one-dimensional operations S_1 and S_2. This amounts to applying S_1 to all columns in the image, and then S_2 to all rows in the result. You should know how this operation can be conveniently expressed with tensor product notation, and that in the typical case when S_1 and S_2 are filters, this can equivalently be expressed in terms of computational molecules.
- The case when the S_i are smoothing filters gives rise to smoothing operations on images.
- A simple highpass filter, corresponding to taking the derivative, gives rise to edge-detection operations on images.

Exercise 9.5: Implement a tensor product

Implement a function `tensor_impl` which takes a matrix `X`, and functions `S1` and `S2` as parameters, and applies `S1` to the columns of `X`, and `S2` to the rows of `X`.

Exercise 9.6: Generate images

Write code which calls the function `tensor_impl` with appropriate filters and which generate the following images:

a) The right image in Figure 9.11.

b) The right image in Figure 9.13.

c) The images in figures 9.14.

d) The images in Figure 9.15.

e) The images in Figure 9.16.

Exercise 9.7: Interpret tensor products

Let the filter S be defined by $S = \{\underline{-1}, 1\}$.

a) Let X be a matrix which represents the pixel values in an image. What can you say about how the new images $(S \otimes I)X$ og $(I \otimes S)X$ look? What are the interpretations of these operations?

b) Write down the $4 \otimes 4$-matrix $X = (1,1,1,1) \otimes (0,0,1,1)$. Compute $(S \otimes I)X$ by applying the filters to the corresponding rows/columns of X as we have learnt, and interpret the result. Do the same for $(I \otimes S)X$.

Exercise 9.8: Computational molecule of moving average filter

Let S be the moving average filter of length $2L+1$, i.e. $T = \frac{1}{L}\{\underbrace{1, \cdots, 1, \underline{1}, 1, \cdots, 1}_{2L+1 \text{ times}}\}$.

What is the computational molecule of $S \otimes S$?

Exercise 9.9: Bilinearity of the tensor product

Show that the mapping $F(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} \otimes \boldsymbol{y}$ is bi-linear, i.e. that $F(\alpha \boldsymbol{x}_1 + \beta \boldsymbol{x}_2, \boldsymbol{y}) = \alpha F(\boldsymbol{x}_1, \boldsymbol{y}) + \beta F(\boldsymbol{x}_2, \boldsymbol{y})$, and $F(\boldsymbol{x}, \alpha \boldsymbol{y}_1 + \beta \boldsymbol{y}_2) = \alpha F(\boldsymbol{x}, \boldsymbol{y}_1) + \beta F(\boldsymbol{x}, \boldsymbol{y}_2)$.

Exercise 9.10: Attempt to write as tensor product

Attempt to find matrices $S_1 : \mathbb{R}^M \to \mathbb{R}^M$ and $S_2 : \mathbb{R}^N \to \mathbb{R}^N$ so that the following mappings from $L_{M,N}(\mathbb{R})$ to $L_{M,N}(\mathbb{R})$ can be written on the form $X \to S_1 X (S_2)^T = (S_1 \otimes S_2)X$. In all the cases, it may be that no such S_1, S_2 can be found. If this is the case, prove it.

a) The mapping which reverses the order of the rows in a matrix.

b) The mapping which reverses the order of the columns in a matrix.

c) The mapping which transposes a matrix.

Exercise 9.11: Computational molecules

Let the filter S be defined by $S = \{1, \underline{2}, 1\}$.

a) Write down the computational molecule of $S \otimes S$.

b) Let us define $\boldsymbol{x} = (1, 2, 3)$, $\boldsymbol{y} = (3, 2, 1)$, $\boldsymbol{z} = (2, 2, 2)$, and $\boldsymbol{w} = (1, 4, 2)$. Compute the matrix $A = \boldsymbol{x} \otimes \boldsymbol{y} + \boldsymbol{z} \otimes \boldsymbol{w}$.

c) Compute $(S \otimes S)A$ by applying the filter S to every row and column in the matrix the way we have learnt. If the matrix A was more generally an image, what can you say about how the new image will look?

Exercise 9.12: Computational molecules

Let $S = \frac{1}{4}\{1, \underline{2}, 1\}$ be a filter.

a) What is the effect of applying the tensor products $S \otimes I$, $I \otimes S$, and $S \otimes S$ on an image represented by the matrix X?

b) Compute $(S \otimes S)(\boldsymbol{x} \otimes \boldsymbol{y})$, where $\boldsymbol{x} = (4, 8, 8, 4)$, $\boldsymbol{y} = (8, 4, 8, 4)$ (i.e. both $\boldsymbol{x}$ and $\boldsymbol{y}$ are column vectors).

Exercise 9.13: Comment on code

Suppose that we have an image given by the $M \times N$-matrix `X`, and consider the following code:

```
for n in range(N):
    X[0, n] = 0.25*X[N-1, n] + 0.5*X[0, n] + 0.25*X[1, n]
    X[1:(N-1), n] = 0.25*X[0:(N-2), n] + 0.5*X[1:(N-1), n] \
    + 0.25*X[2:N, n]
    X[N-1, n] = 0.25*X[N-2, n] + 0.5*X[N-1, n] + 0.25*X[0, n]
for m in range(m):
    X[m, 0] = 0.25*X[m, M-1] + 0.5*X[m, 0] + 0.25*X[m, 1]
    X[m, 1:(M-1)] = 0.25*X[m, 0:(M-2)] + 0.5*X[m, 1:(M-1)] \
    + 0.25*X[m, 2:M]
    X[m, M-1] = 0.25*X[m, M-2] + 0.5*X[m, M-1] + 0.25*X[m, 0]
```

Which tensor product is applied to the image? Comment what the code does, in particular the first and third line in the inner `for`-loop. What effect does the code have on the image?

Exercise 9.14: Eigenvectors of tensor products

Let $\boldsymbol{v}_A$ be an eigenvector of A with eigenvalue λ_A, and $\boldsymbol{v}_B$ an eigenvector of B with eigenvalue λ_B. Show that $\boldsymbol{v}_A \otimes \boldsymbol{v}_B$ is an eigenvector of $A \otimes B$ with eigenvalue $\lambda_A \lambda_B$. Explain from this why $\|A \otimes B\| = \|A\|\|B\|$, where $\|\cdot\|$ denotes the operator norm of a matrix.

Exercise 9.15: The Kronecker product

The *Kronecker tensor product* of two matrices A and B, written $A \otimes^k B$, is defined as

$$A \otimes^k B = \begin{pmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,M}B \\ a_{2,1}B & a_{2,2}B & \cdots & a_{2,M}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{p,1}B & a_{p,2}B & \cdots & a_{p,M}B \end{pmatrix},$$

where the entries of A are $a_{i,j}$. The tensor product of a $p \times M$-matrix, and a $q \times N$-matrix is thus a $(pq) \times (MN)$-matrix. Note that this tensor product in particular gives meaning for vectors: if $\boldsymbol{x} \in \mathbb{R}^M$, $\boldsymbol{y} \in \mathbb{R}^N$ are column vectors, then $\boldsymbol{x} \otimes^k \boldsymbol{y} \in \mathbb{R}^{MN}$ is also a column vector. In this exercise we will investigate how the Kronecker tensor product is related to tensor products as we have defined them in this section.

a) Explain that, if $\boldsymbol{x} \in \mathbb{R}^M$, $\boldsymbol{y} \in \mathbb{R}^N$ are column vectors, then $\boldsymbol{x} \otimes^k \boldsymbol{y}$ is the column vector where the rows of $\boldsymbol{x} \otimes \boldsymbol{y}$ have first been stacked into one large row vector, and this vector transposed. The linear extension of the operation defined by

$$\boldsymbol{x} \otimes \boldsymbol{y} \in \mathbb{R}^{M,N} \to \boldsymbol{x} \otimes^k \boldsymbol{y} \in \mathbb{R}^{MN}$$

thus stacks the rows of the input matrix into one large row vector, and transposes the result.

b) Show that $(A \otimes^k B)(\boldsymbol{x} \otimes^k \boldsymbol{y}) = (A\boldsymbol{x}) \otimes^k (B\boldsymbol{y})$. We can thus use any of the defined tensor products $\otimes$, $\otimes_k$ to produce the same result, i.e. we have the commutative diagram shown in Figure 9.18, where the vertical arrows represent stacking the rows in the matrix, and transposing, and the horizontal arrows represent the two tensor product linear transformations we have defined. In particular, we can compute the tensor product in terms of vectors, or in terms of matrices, and it is clear that the Kronecker tensor product gives the matrix of tensor product operations.

$$\begin{array}{ccc} \boldsymbol{x} \otimes \boldsymbol{y} & \xrightarrow{A \otimes B} & (A\boldsymbol{x}) \otimes (B\boldsymbol{y}) \\ \downarrow & & \downarrow \\ \boldsymbol{x} \otimes^k \boldsymbol{y} & \xrightarrow{A \otimes^k B} & (A\boldsymbol{x}) \otimes^k (B\boldsymbol{y}), \end{array}$$

Figure 9.18: Tensor products

c) Using the Euclidean inner product on $L(M, N) = \mathbb{R}^{MN}$, i.e.

$$\langle X, Y \rangle = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} X_{i,j} \overline{Y_{i,j}}.$$

and the correspondence in a) we can define the inner product of $\boldsymbol{x}_1 \otimes \boldsymbol{y}_1$ and $\boldsymbol{x}_2 \otimes \boldsymbol{y}_2$ by

$$\langle \boldsymbol{x}_1 \otimes \boldsymbol{y}_1, \boldsymbol{x}_2 \otimes \boldsymbol{y}_2 \rangle = \langle \boldsymbol{x}_1 \otimes^k \boldsymbol{y}_1, \boldsymbol{x}_2 \otimes^k \boldsymbol{y}_2 \rangle.$$

Show that

$$\langle \boldsymbol{x}_1 \otimes \boldsymbol{y}_1, \boldsymbol{x}_2 \otimes \boldsymbol{y}_2 \rangle = \langle \boldsymbol{x}_1, \boldsymbol{x}_2 \rangle \langle \boldsymbol{y}_1, \boldsymbol{y}_2 \rangle.$$

Clearly this extends linearly to an inner product on $L_{M,N}$.

d) Show that the FFT factorization can be written as

$$\begin{pmatrix} F_{N/2} & D_{N/2}F_{N/2} \\ F_{N/2} & -D_{N/2}F_{N/2} \end{pmatrix} = \begin{pmatrix} I_{N/2} & D_{N/2} \\ I_{N/2} & -D_{N/2} \end{pmatrix} (I_2 \otimes_k F_{N/2}).$$

Also rewrite the sparse matrix factorization for the FFT from Equation (2.18) in the compendium in terms of tensor products.

9.4 Change of coordinates in tensor products

Filter-based operations were not the only operations we considered for sound. We also considered the DFT, the DCT, and the wavelet transform, which were changes of coordinates which gave us useful frequency- or time-frequency information. We would like to define similar changes of coordinates for images, which also give useful such information. Tensor product notation will also be useful in this respect, and we start with the following result.

Theorem 9.26. *The basis $\mathcal{B}_1 \otimes \mathcal{B}_2$.*

If $\mathcal{B}_1 = \{\boldsymbol{v}_i\}_{i=0}^{M-1}$ is a basis for $\mathbb{R}^M$, and $\mathcal{B}_2 = \{\boldsymbol{w}_j\}_{j=0}^{N-1}$ is a basis for $\mathbb{R}^N$, then $\{\boldsymbol{v}_i \otimes \boldsymbol{w}_j\}_{(i,j)=(0,0)}^{(M-1,N-1)}$ is a basis for $L_{M,N}(\mathbb{R})$. We denote this basis by $\mathcal{B}_1 \otimes \mathcal{B}_2$.

Proof. Suppose that $\sum_{(i,j)=(0,0)}^{(M-1,N-1)} \alpha_{i,j}(\boldsymbol{v}_i \otimes \boldsymbol{w}_j) = 0$. Setting $\boldsymbol{h}_i = \sum_{j=0}^{N-1} \alpha_{i,j}\boldsymbol{w}_j$ we get

$$\sum_{j=0}^{N-1} \alpha_{i,j}(\boldsymbol{v}_i \otimes \boldsymbol{w}_j) = \boldsymbol{v}_i \otimes (\sum_{j=0}^{N-1} \alpha_{i,j}\boldsymbol{w}_j) = \boldsymbol{v}_i \otimes \boldsymbol{h}_i.$$

where we have used the bi-linearity of the tensor product mapping $(\boldsymbol{x}, \boldsymbol{y}) \to \boldsymbol{x} \otimes \boldsymbol{y}$ (Exercise 9.9). This means that

$$\mathbf{0} = \sum_{(i,j)=(0,0)}^{(M-1,N-1)} \alpha_{i,j}(\boldsymbol{v}_i \otimes \boldsymbol{w}_j) = \sum_{i=0}^{M-1} \boldsymbol{v}_i \otimes \boldsymbol{h}_i = \sum_{i=0}^{M-1} \boldsymbol{v}_i \boldsymbol{h}_i^T.$$

Column k in this matrix equation says $\mathbf{0} = \sum_{i=0}^{M-1} h_{i,k}\boldsymbol{v}_i$, where $h_{i,k}$ are the components in $\boldsymbol{h}_i$. By linear independence of the $\boldsymbol{v}_i$ we must have that $h_{0,k} = h_{1,k} = \cdots = h_{M-1,k} = 0$. Since this applies for all k, we must have that all $\boldsymbol{h}_i = 0$. This means that $\sum_{j=0}^{N-1} \alpha_{i,j}\boldsymbol{w}_j = \mathbf{0}$ for all i, from which it follows by linear independence of the $\boldsymbol{w}_j$ that $\alpha_{i,j} = 0$ for all j, and for all i. This means that $\mathcal{B}_1 \otimes \mathcal{B}_2$ is a basis. □

In particular, as we have already seen, the standard basis for $L_{M,N}(\mathbb{R})$ can be written $\mathcal{E}_{M,N} = \mathcal{E}_M \otimes \mathcal{E}_N$. This is the basis for a useful convention: For a tensor product the bases are most naturally indexed in two dimensions, rather than the usual sequential indexing. This difference translates also to the meaning of coordinate vectors, which now are more naturally thought of as coordinate matrices:

Definition 9.27. *Coordinate matrix.*

Let $\mathcal{B} = \{\boldsymbol{b}_i\}_{i=0}^{M-1}$, $\mathcal{C} = \{\boldsymbol{c}_j\}_{j=0}^{N-1}$ be bases for $\mathbb{R}^M$ and $\mathbb{R}^N$, and let $A \in L_{M,N}(\mathbb{R})$. By the coordinate matrix of A in $\mathcal{B} \otimes \mathcal{C}$ we mean the $M \times N$-matrix X (with components X_{kl}) such that $A = \sum_{k,l} X_{k,l}(\boldsymbol{b}_k \otimes \boldsymbol{c}_l)$.

We will have use for the following theorem, which shows how change of coordinates in $\mathbb{R}^M$ and $\mathbb{R}^N$ translate to a change of coordinates in the tensor product:

Theorem 9.28. *Change of coordinates in tensor products.*

Assume that

- $\mathcal{B}_1, \mathcal{C}_1$ are bases for $\mathbb{R}^M$, and that S_1 is the change of coordinates matrix from $\mathcal{B}_1$ to $\mathcal{C}_1$,
- $\mathcal{B}_2, \mathcal{C}_2$ are bases for $\mathbb{R}^N$, and that S_2 is the change of coordinates matrix from $\mathcal{B}_2$ to $\mathcal{C}_2$.

Both $\mathcal{B}_1 \otimes \mathcal{B}_2$ and $\mathcal{C}_1 \otimes \mathcal{C}_2$ are bases for $L_{M,N}(\mathbb{R})$, and if X is the coordinate matrix in $\mathcal{B}_1 \otimes \mathcal{B}_2$, and Y the coordinate matrix in $\mathcal{C}_1 \otimes \mathcal{C}_2$, then the change of coordinates from $\mathcal{B}_1 \otimes \mathcal{B}_2$ to $\mathcal{C}_1 \otimes \mathcal{C}_2$ can be computed as

$$Y = S_1 X (S_2)^T. \tag{9.15}$$

Proof. Let $\boldsymbol{c}_{ki}$ be the i'th basis vector in $\mathcal{C}_k$, $\boldsymbol{b}_{ki}$ the i'th basis vector in $\mathcal{B}_k$, $k = 1, 2$. Since any change of coordinates is linear, it is enough to show that it coincides with $X \to S_1 X (S_2)^T$ for any basis vector in $\mathcal{B}_1 \otimes \mathcal{B}_2$. The basis vector $\boldsymbol{b}_{1i} \otimes \boldsymbol{b}_{2j}$ has coordinate vector $X = \boldsymbol{e}_i \otimes \boldsymbol{e}_j$ in $\mathcal{B}_1 \otimes \mathcal{B}_2$. With the mapping $X \to S_1 X (S_2)^T$ this is sent to

$$S_1 X (S_2)^T = S_1(\boldsymbol{e}_i \otimes \boldsymbol{e}_j)(S_2)^T = S_1 \boldsymbol{e}_i (\boldsymbol{e}_j)^T (S_2)^T = S_1 \boldsymbol{e}_i (S_2 \boldsymbol{e}_j)^T = \mathrm{col}_i(S_1)(\mathrm{col}_j(S_2))^T.$$

On the other hand, since column i in S_1 is the coordinates of $\boldsymbol{b}_{1i}$ in the basis $\mathcal{C}_1$, and column j in S_2 is the coordinates of $\boldsymbol{b}_{2j}$ in the basis $\mathcal{C}_2$, we can write

$$\begin{aligned}
\boldsymbol{b}_{1i} \otimes \boldsymbol{b}_{2j} &= \left(\sum_k (S_1)_{k,i} \boldsymbol{c}_{1k} \right) \otimes \left(\sum_l (S_2)_{l,j} \boldsymbol{c}_{2l} \right) = \sum_{k,l} (S_1)_{k,i} (S_2)_{l,j} (\boldsymbol{c}_{1k} \otimes \boldsymbol{c}_{2l}) \\
&= \sum_{k,l} (\mathrm{col}_i(S_1))_k (\mathrm{col}_j(S_2))_l (\boldsymbol{c}_{1k} \otimes \boldsymbol{c}_{2l}) \\
&= \sum_{k,l} (\mathrm{col}_i(S_1)(\mathrm{col}_j(S_2))^T)_{k,l} (\boldsymbol{c}_{1k} \otimes \boldsymbol{c}_{2l}).
\end{aligned}$$

This shows that the coordinate matrix of $\boldsymbol{b}_{1i} \otimes \boldsymbol{b}_{2j}$ in $\mathcal{C}_1 \otimes \mathcal{C}_2$ is $\text{col}_i(S_1)(\text{col}_j(S_2))^T$. This means that the change of coordinates coincides with the mapping $X \to S_1 X (S_2)^T$ for any vector in $\mathcal{B}_1 \otimes \mathcal{B}_2$, so that the change of coordinates is given by $X \to S_1 X (S_2)^T$ for all vectors also. □

In both cases of filtering and change of coordinates in tensor products, we see that we need to compute the mapping $X \to S_1 X (S_2)^T$. As we have seen, this amounts to a row/column-wise operation, which we restate as follows:

Observation 9.29. *Change of coordinates in tensor products.*

The change of coordinates from $\mathcal{B}_1 \otimes \mathcal{B}_2$ to $\mathcal{C}_1 \otimes \mathcal{C}_2$ can be implemented as follows:

- For every column in the coordinate matrix in $\mathcal{B}_1 \otimes \mathcal{B}_2$, perform a change of coordinates from $\mathcal{B}_1$ to $\mathcal{C}_1$.
- For every row in the resulting matrix, perform a change of coordinates from $\mathcal{B}_2$ to $\mathcal{C}_2$.

We can again use the funtion `tensor_impl` in order to implement change of coordinates for a tensor product. We just need to replace the filters with the functions `S1` and `S2` for computing the corresponding changes of coordinates:

```
tensor_impl(X, S1, S2)
```

The operation $X \to (S_1)X(S_2)^T$, which we now have encountered in two different ways, is one particular type of linear transformation from $\mathbb{R}^{N^2}$ to itself (see Exercise 9.15 for how the matrix of this linear transformation can be constructed). While a general such linear transformation requires N^4 multiplications (i.e. when we perform a full matrix multiplication), $X \to (S_1)X(S_2)^T$ can be implemented generally with only $2N^3$ multiplications (since multiplication of two $N \times N$-matrices require N^3 multiplications in general). The operation $X \to (S_1)X(S_2)^T$ is thus computationally simpler than linear transformations in general. In practice the operations S_1 and S_2 are also computationally simpler, since they can be filters, FFT's, or wavelet transformations, so that the complexity in $X \to (S_1)X(S_2)^T$ can be even lower.

In the following examples, we will interpret the pixel values in an image as coordinates in the standard basis, and perform a change of coordinates.

Example 9.30. *Change of coordinates with the DFT.*

The DFT is one particular change of coordinates which we have considered. It was the change of coordinates from the standard basis to the Fourier basis. A corresponding change of coordinates in a tensor product is obtained by substituting the DFT as the functions S_1, S_2 for implementing the changes of coordinates above. The change of coordinates in the opposite direction is obtained by using the IDFT instead of the DFT.

Modern image standards do typically not apply a change of coordinates to the entire image. Rather the image is split into smaller squares of appropriate size, called blocks, and a change of coordinates is performed independently for each block. In this example we have split the image into blocks of size 8×8.

Recall that the DFT values express frequency components. The same applies for the two-dimensional DFT and thus for images, but frequencies are now represented in two different directions. Let us introduce a neglection threshold in the same way as in Example 2.28, to view the image after we set certain frequencies to zero. As for sound, this has little effect on the human perception of the image, if we use a suitable neglection threshold. After we have performed the two-dimensional DFT on an image, we can neglect DFT-coefficients below a threshold on the resulting matrix `X` with the following code:

```
X *= (abs(X) >= threshold)
```

`abs(X)>=threshold` now instead returns a *threshold matrix* with 1 and 0 of the same size as `X`.

In Figure 9.19 we have applied the two-dimensional DFT to our test image. We have then neglected DFT coefficients which are below certain thresholds, and transformed the samples back to reconstruct the image. When increasing the threshold, the image becomes more and more unclear, but the image is quite clear in the first case, where as much as more than 76.6% of the samples have been zeroed out. A blocking effect at the block boundaries is clearly visible.

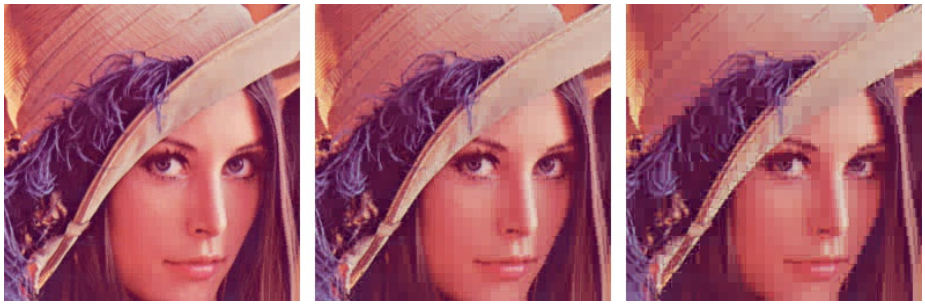

Figure 9.19: The effect on an image when it is transformed with the DFT, and the DFT-coefficients below a certain threshold are zeroed out. The threshold has been increased from left to right, from 100, to 200, and 400. The percentage of pixel values that were zeroed out are 76.6, 89.3, and 95.3, respectively.

Example 9.31. *Change of coordinates with the DCT.*

Similarly to the DFT, the DCT was the change of coordinates from the standard basis to what we called the DCT basis. Change of coordinates in tensor products between the standard basis and the DCT basis is obtained by substituting with the DCT and the IDCT for the changes of coordinates S_1, S_2 above.

The DCT is used more than the DFT in image processing. In particular, the JPEG standard applies a two-dimensional DCT, rather than a two-dimensional DFT. With the JPEG standard, the blocks are always 8×8, as in the previous example. It is of course not a coincidence that a power of 2 is chosen here: the DCT, as the DFT, has an efficient implementation for powers of 2.

If we follow the same strategy for the DCT as for the DFT example, so that we zero out DCT-coefficients which are below a given threshold [1], and use the same block sizes, we get the images shown in Figure 9.20. We see similar effects as with the DFT.

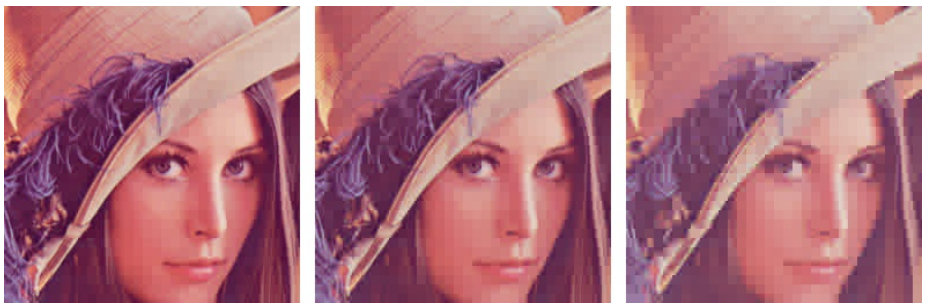

Figure 9.20: The effect on an image when it is transformed with the DCT, and the DCT-coefficients below a certain threshold are zeroed out. The threshold has been increased from left to right, from 30, to 50, and 100. The percentage of pixel values that were zeroed out are 93.2, 95.8, and 97.7, respectively.

It is also interesting to compare with what happens when we drop splitting the image into blocks. Of course, when we neglect many of the DCT-coefficients, we should see some artifacts, but there is no reason to believe that these should be at the old block boundaries. The new artifacts can be seen in Figure 9.21, where the same thresholds as before have been used. Clearly, the new artifacts take a completely different shape.

In the exercises you will be asked to implement functions which generate the images shown in these examples.

What you should have learned in this section.

- The operation $X \to S_1 X (S_2)^T$ can also be used to facilitate change of coordinates in images, in addition to filtering images. In other words, change of coordinates is done first column by column, then row by row. The DCT and the DFT are particular changes of coordinates used for images.

[1] The JPEG standard does not do exactly the kind of thresholding described here. Rather it performs what is called a quantization.

Figure 9.21: The effect on an image when it is transformed with the DCT, and the DCT-coefficients below a certain threshold are zeroed out. The image has not been split into blocks here, and the same thresholds as in Figure 9.20 were used. The percentage of pixel values that were zeroed out are 93.2, 96.6, and 98.8, respectively.

Exercise 9.16: Implement DFT and DCT on blocks

In this section we have used functions which apply the DCT and the DFT either to subblocks of size 8×8, or to the full image. Implement functions which applies the DFT, IDFT, DCT, and IDCT, to consecutive segments of length 8.

Exercise 9.17: Implement two-dimensional FFT and DCT

Write down code for running FFT2, IFFT2, DCT2, and IDCT2 on an image, using the function `tensor_impl`.

Exercise 9.18: Zeroing out DCT coefficients

The following function `showDCThigher` applies the DCT to an image in the same way as the JPEG standard does. The function takes a threshold parameter, and sets DCT coefficients below this value to zero:

```
def showDCThigher(threshold):
    img = imread('images/lena.png', 'png').astype(float)
    zeroedout = 0
    tensor_impl(img, DCTImpl8, DCTImpl8)
    thresholdmatr = (abs(img) >= threshold)
    zeroedout += prod(shape(img)) - sum(thresholdmatr)
    img *= thresholdmatr
    tensor_impl(img, IDCTImpl8, IDCTImpl8)
    mapto01(img)
    imshow(uint8(255*img))
    print '%i percent of samples zeroed out\n' \
    % 100*zeroedout/prod(shape(img))
```

a) Explain this code line by line.

b) Run `showDCThigher` for different threshold parameters, and check that this reproduces the test images of this section, and prints the correct numbers of values which have been neglected (i.e. which are below the threshold) on screen.

Exercise 9.19: Comment code

Suppose that we have given an image by the matrix `X`. Consider the following code:

```
threshold = 30
[M, N] = shape(X)[0:2]
for n in range(N):
    FFTImpl(X[:, n], FFTKernelStandard)
for m in range(M):
    FFTImpl(X[m, :], FFTKernelStandard)

X = X.*(abs(X) >= threshold)

for n in range(N):
    FFTImpl(X[:, n], FFTKernelStandard, 0)
for m in range(M):
    FFTImpl((X[m, :], FFTKernelStandard, 0)
```

Comment what the code does. Comment in particular on the meaning of the parameter `threshold`, and what effect this has on the image.

9.5 Summary

We started by discussing the basic question what an image is, and took a closer look at digital images. We then went through several operations which give meaning for digital images. Many of these operations could be described in terms of a row/column-wise application of filters, and more generally in term of what we called computational molecules. We defined the tensor product, and saw how our operations could be expressed within this framework. The tensor product framework could also be used to state change of coordinates for images, so that we could consider changes of coordinates such as the DFT and the DCT also for images. The algorithm for computing filtering operations or changes of coordinates for images turned out to be similar, in the sense that the one-dimensional counterparts were simply assplied to the rows and the columns in the image.

In introductory image processing textbooks, many other image processing methods are presented. We have limited to the techniques presented here, since our interest in images is mainly for transformation operations which are useful for compression. An excellent textbook on image processing which uses Matlab is [15]. This contains important topics such as image restoration and reconstruction, geometric transformations, morphology, and object recognition. None of these are considered in this book.

In much literature, one only mentions that filtering can be extended to images by performing one-dimensional filtering for the rows, followed by one-dimensional

filtering for the columns, without properly explaining why this is the natural thing to do. The tensor product may be the most natural concept to explain this, and a concept which is firmly established in mathematical literature. Tensor products are usually not part of beginning courses in linear algebra. We have limited the focus here to an introduction to tensor products, and the theory needed to explain filtering an image, and computing the two-dimensional wavelet transform. Some linear algebra books (such as [22]) present tensor products in exercise form only, and often only mentions the Kronecker tensor product, as we defined it.

Many international standards exist for compression of images, and we will take a closer look at two of them in this book. The JPEG standard, perhaps the most popular format for images on the Internet, applies a change of coordinates with a two-dimensional DCT, as described in this chapter. The compression level in JPEG images is selected by the user and may result in conspicuous artefacts if set too high. JPEG is especially prone to artefacts in areas where the intensity changes quickly from pixel to pixel. JPEG is usually lossy, but may also be lossless and has become. The standard defines both the algorithms for encoding and decoding and the storage format. The extension of a JPEG-file is `.jpg` or `.jpeg`. JPEG is short for *Joint Photographic Experts Group*, and was approved as an international standard in 1994. A more detailed description of the standard can be found in [27].

The second standard we will consider is JPEG2000. It was developed to address some of the shortcomings of JPEG, and is based on wavelets. The standard document for this [17] does not focus on explaining the theory behind the standard. As the MP3 standard document, it rather states step-by-step procedures for implementing the standard.

The theory we present related to these image standards concentrate on transforming the image (either with a DWT or a DCT) to obtain something which is more suitable for (lossless or lossy) compression. However, many other steps are also needed in order to obtain a full image compression system. One of these is *quantization*. In the simplest form of quantization, every resulting sample from the transformation is rounded to a fixed number of bits. Quantization can also be done in more advanced ways than this: We have already mentioned that the MP3 standard may use different number of bits for values in the different subbands, depending on the importance of the samples for the human perception. The JPEG2000 standard quantizes in such a way that there is bigger interval around 0 which is quantized to 0, i.e. the rounding error is allowed to be bigger in an interval around 0. Standards which are lossless do not apply quantization, since this always leads to loss.

Somewhere in the image processing or sound processing pipeline, we also need a step which actually achieves compression of the data. Different standards use different lossless coding techniques for this. JPEG2000 uses an advances type of arithmetic coding for this. JPEG can also use arithmetic coding, but also Huffman coding.

Besides transformation, quantization, and coding, many other steps are used, which have different tasks. Many standards preprocess the pixel values

before a transform is applied. Preprocessing may mean to center the pixel values around a certain value (JPEG2000 does this), or extracting the different image components before they are processed separately. Also, the image is often split into smaller parts (often called *tiles*), which are processed separately. For big images this is very important, since it allows users to zoom in on a small part of the image, without processing larger uninteresting parts of the image. Independent processing of the separate tiles makes the image compression what we call *error-resilient*, to errors such as transmission errors, since errors in one tile does not propagate to errors in the other tiles. It is also much more memory-friendly to process the image in several smaller parts, since it is not required to have the entire image in memory at any time. It also gives possibilities for parallel computing. For standards such as JPEG and JPEG2000, tiles are split into even smaller parts, called *blocks*, where parts of the processing within each block also is performed independently. This makes the possibilities for parallel computing even bigger.

An image standard also defines how to store metadata about an image, and what metadata is accepted, like resolution, time when the image was taken, where the image was taken (such as GPS coordinates), and similar information. Metadata can also tell us how the color in the image are represented. As we have already seen, in most color images the color of a pixel is represented in terms of the amount of red, green and blue or (r, g, b). But there are other possibilities as well: Instead of storing all 24 bits of color information in cases where each of the three color components needs 8 bits, it is common to create a table of up to 256 colors with which a given image could be represented quite well. Instead of storing the 24 bits, one then just stores a color table in the metadata, and at each pixel, the eight bits corresponding to the correct entry in the table. This is usually referred to as *eight-bit color*, and the table is called a *look-up* table or *palette*. For large photographs, however, 256 colors is far from sufficient to obtain reasonable colour reproduction. Metadata is usually stored in the beginning of the file, formatted in a very specific way.

Chapter 10

Using tensor products to apply wavelets to images

Previously we have used the theory of wavelets to analyze sound. We would also like to use wavelets in a similar way to analyze images. Since the tensor product concept constructs two dimensional objects (matrices) from one-dimensional objects (vectors), we are lead to believe that tensor products can also be used to apply wavelets to images. In this chapter we will see that this can indeed be done. The vector spaces we V_m encountered for wavelets were function spaces, however. What we therefore need first is to establish a general definition of tensor products of function spaces. This will be done in the first section of this chapter. In the second section we will then specialize the function spaces to the spaces V_m we use for wavelets, and interpret the tensor product of these and the wavelet transform applied to images more carefully. Finally we will look at some examples on this theory applied to some example images.

The examples in this chapter can be run from the notebook `applinalgnbchap10.ipynb`.

10.1 Tensor product of function spaces

In the setting of functions, it will turn out that the tensor product of two univariate functions can be most intiutively defined as a function in two variables. This seems somewhat different from the strategy of Chapter 9, but we will see that the results we obtain will be very similar.

Definition 10.1. *Tensor product of function spaces.*

Let U_1 and U_2 be vector spaces of functions, defined on the intervals $[0, M)$ and $[0, N)$, respectively, and suppose that $f_1 \in U_1$ and $f_2 \in U_2$. The tensor product of f_1 and f_2, denoted $f_1 \otimes f_2$, is the function in two variables defined on $[0, M) \times [0, N)$ by

$$(f_1 \otimes f_2)(t_1, t_2) = f_1(t_1) f_2(t_2).$$

$f_1 \otimes f_2$ is also called the separable extension of f_1 and f_2 to two variables. The tensor product of the spaces $U_1 \otimes U_2$ is the vector space spanned by the two-variable functions $\{f_1 \otimes f_2\}_{f_1 \in U_1, f_2 \in U_2}$.

We will always assume that the spaces U_1 and U_2 consist of functions which are at least integrable. In this case $U_1 \otimes U_2$ is also an inner product space, with the inner product given by a double integral,

$$\langle f, g \rangle = \int_0^N \int_0^M f(t_1, t_2) g(t_1, t_2) dt_1 dt_2. \tag{10.1}$$

In particular, this says that

$$\begin{aligned} \langle f_1 \otimes f_2, g_1 \otimes g_2 \rangle &= \int_0^N \int_0^M f_1(t_1) f_2(t_2) g_1(t_1) g_2(t_2) dt_1 dt_2 \\ &= \int_0^M f_1(t_1) g_1(t_1) dt_1 \int_0^N f_2(t_2) g_2(t_2) dt_2 = \langle f_1, g_1 \rangle \langle f_2, g_2 \rangle. \end{aligned} \tag{10.2}$$

This means that for tensor products, a double integral can be computed as the product of two one-dimensional integrals. This formula also ensures that inner products of tensor products of functions obey the same rule as we found for tensor products of vectors in Exercise 9.15.

The tensor product space defined in Definition 10.1 is useful for approximation of functions of two variables if each of the two spaces of univariate functions have good approximation properties.

Idea 10.2. *Using tensor products for approximation.*

If the spaces U_1 and U_2 can be used to approximate functions in one variable, then $U_1 \otimes U_2$ can be used to approximate functions in two variables.

We will not state this precisely, but just consider some important examples.

Example 10.3. *Tensor products of polynomials.*

Let $U_1 = U_2$ be the space of all polynomials of finite degree. We know that U_1 can be used for approximating many kinds of functions, such as continuous functions, for example by Taylor series. The tensor product $U_1 \otimes U_1$ consists of all functions on the form $\sum_{i,j} \alpha_{i,j} t_1^i t_2^j$. It turns out that polynomials in several variables have approximation properties analogous to univariate polynomials.

Example 10.4. *Tensor products of Fourier spaces.*

Let $U_1 = U_2 = V_{N,T}$ be the Nth order Fourier space which is spanned by the functions

$$e^{-2\pi i N t/T}, \ldots, e^{-2\pi i t/T}, 1, e^{2\pi i t/T}, \ldots, e^{2\pi i N t/T}$$

The tensor product space $U_1 \otimes U_1$ now consists of all functions on the form

$$\sum_{k,l=-N}^{N} \alpha_{k,l} e^{2\pi i k t_1/T} e^{2\pi i l t_2/T}.$$

One can show that this space has approximation properties similar to $V_{N,T}$ for functions in two variables. This is the basis for the theory of Fourier series in two variables.

In the following we think of $U_1 \otimes U_2$ as a space which can be used for approximating a general class of functions. By associating a function with the vector of coordinates relative to some basis, and a matrix with a function in two variables, we have the following parallel to Theorem 9.26:

Theorem 10.5. *Bases for tensor products of function spaces.*
If $\{f_i\}_{i=0}^{M-1}$ is a basis for U_1 and $\{g_j\}_{j=0}^{N-1}$ is a basis for U_2, then $\{f_i \otimes g_j\}_{(i,j)=(0,0)}^{(M-1,N-1)}$ is a basis for $U_1 \otimes U_2$. Moreover, if the bases for U_1 and U_2 are orthogonal/orthonormal, then the basis for $U_1 \otimes U_2$ is orthogonal/orthonormal.

Proof. The proof is similar to that of Theorem 9.26: if

$$\sum_{(i,j)=(0,0)}^{(M-1,N-1)} \alpha_{i,j}(f_i \otimes g_j) = 0,$$

we define $h_i(t_2) = \sum_{j=0}^{N-1} \alpha_{i,j} g_j(t_2)$. It follows as before that $\sum_{i=0}^{M-1} h_i(t_2) f_i = 0$ for any t_2, so that $h_i(t_2) = 0$ for any t_2 due to linear independence of the f_i. But then $\alpha_{i,j} = 0$ also, due to linear independene of the g_j. The statement about orthogonality follows from Equation (10.2). □

We can now define the tensor product of two bases of functions as before, and coordinate matrices as before:

Definition 10.6. *Coordinate matrix.*
if $\mathcal{B} = \{f_i\}_{i=0}^{M-1}$ and $\mathcal{C} = \{g_j\}_{j=0}^{N-1}$, we define $\mathcal{B} \otimes \mathcal{C}$ as the basis $\{f_i \otimes g_j\}_{(i,j)=(0,0)}^{(M-1,N-1)}$ for $U_1 \otimes U_2$. We say that X is the coordinate matrix of f if $f(t_1, t_2) = \sum_{i,j} X_{i,j}(f_i \otimes g_j)(t_1, t_2)$, where $X_{i,j}$ are the elements of X.

Theorem 9.28 can also be proved in the same way in the context of function spaces. We state this as follows:

Theorem 10.7. *Change of coordinates in tensor products of function spaces.*
Assume that U_1 and U_2 are function spaces, and that

- $\mathcal{B}_1, \mathcal{C}_1$ are bases for U_1, and that S_1 is the change of coordinates matrix from $\mathcal{B}_1$ to $\mathcal{C}_1$,
- $\mathcal{B}_2, \mathcal{C}_2$ are bases for U_2, and that S_2 is the change of coordinates matrix from $\mathcal{B}_2$ to $\mathcal{C}_2$.

Both $\mathcal{B}_1 \otimes \mathcal{B}_2$ and $\mathcal{C}_1 \otimes \mathcal{C}_2$ are bases for $U_1 \otimes U_2$, and if X is the coordinate matrix in $\mathcal{B}_1 \otimes \mathcal{B}_2$, Y the coordinate matrix in $\mathcal{C}_1 \otimes \mathcal{C}_2$, then the change of coordinates from $\mathcal{B}_1 \otimes \mathcal{B}_2$ to $\mathcal{C}_1 \otimes \mathcal{C}_2$ can be computed as

$$Y = S_1 X (S_2)^T. \tag{10.3}$$

10.2 Tensor product of function spaces in a wavelet setting

We will now specialize the spaces U_1, U_2 from Definition 10.1 to the resolution spaces V_m and the detail spaces W_m, arising from a given wavelet. We can in particular form the tensor products $\phi_{0,n_1} \otimes \phi_{0,n_2}$. We will assume that

- the first component ϕ_{0,n_1} has period M (so that $\{\phi_{0,n_1}\}_{n_1=0}^{M-1}$ is a basis for the first component space),
- the second component ϕ_{0,n_2} has period N (so that $\{\phi_{0,n_2}\}_{n_2=0}^{N-1}$ is a basis for the second component space).

When we speak of $V_0 \otimes V_0$ we thus mean an MN-dimensional space with basis $\{\phi_{0,n_1} \otimes \phi_{0,n_2}\}_{(n_1,n_2)=(0,0)}^{(M-1,N-1)}$, where the coordinate matrices are $M \times N$. This difference in the dimension of the two components is done to allow for images where the number of rows and columns may be different. In the following we will implicitly assume that the component spaces have dimension M and N, to ease notation. If we use that $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$ also is a basis for V_m, we get the following corollary to Theorem 10.5:

Corollary 10.8. *Bases for tensor products.*

Let ϕ, ψ be a scaling function and a mother wavelet. Then the two sets of tensor products given by

$$\boldsymbol{\phi}_m \otimes \boldsymbol{\phi}_m = \{\phi_{m,n_1} \otimes \phi_{m,n_2}\}_{n_1,n_2}$$

and

$$\begin{aligned}
&(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1}) \otimes (\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1}) \\
&= \{\phi_{m-1,n_1} \otimes \phi_{m-1,n_2}, \\
&\qquad \phi_{m-1,n_1} \otimes \psi_{m-1,n_2}, \\
&\qquad \psi_{m-1,n_1} \otimes \phi_{m-1,n_2}, \\
&\qquad \psi_{m-1,n_1} \otimes \psi_{m-1,n_2}\}_{n_1,n_2}
\end{aligned}$$

are both bases for $V_m \otimes V_m$. This second basis is orthogonal/orthonormal whenever the first basis is.

From this we observe that, while the one-dimensional wavelet decomposition splits V_m into a direct sum of the two vector spaces V_{m-1} and W_{m-1}, the corresponding two-dimensional decomposition splits $V_m \otimes V_m$ into a direct sum of four tensor product vector spaces. These vector spaces deserve individual names:

Definition 10.9. *Tensor product spaces.*
We define the following tensor product spaces:

- The space $W_m^{(0,1)}$ spanned by $\{\phi_{m,n_1} \otimes \psi_{m,n_2}\}_{n_1,n_2}$,
- The space $W_m^{(1,0)}$ spanned by $\{\psi_{m,n_1} \otimes \phi_{m,n_2}\}_{n_1,n_2}$,
- The space $W_m^{(1,1)}$ spanned by $\{\psi_{m,n_1} \otimes \psi_{m,n_2}\}_{n_1,n_2}$.

Since these spaces are linearly independent, we can write

$$V_m \otimes V_m = (V_{m-1} \otimes V_{m-1}) \oplus W_{m-1}^{(0,1)} \oplus W_{m-1}^{(1,0)} \oplus W_{m-1}^{(1,1)}. \tag{10.4}$$

Also in the setting of tensor products we refer to $V_{m-1} \otimes V_{m-1}$ as the space of low-resolution approximations. The remaining parts, $W_{m-1}^{(0,1)}$, $W_{m-1}^{(1,0)}$, and $W_{m-1}^{(1,1)}$, are refered to as detail spaces. The coordinate matrix of

$$\begin{aligned}\sum_{n_1,n_2=0}^{2^{m-1}N} &(c_{m-1,n_1,n_2}(\phi_{m-1,n_1} \otimes \phi_{m-1,n_2}) + w_{m-1,n_1,n_2}^{(0,1)}(\phi_{m-1,n_1} \otimes \psi_{m-1,n_2}) + \\ &w_{m-1,n_1,n_2}^{(1,0)}(\psi_{m-1,n_1} \otimes \phi_{m-1,n_2}) + w_{m-1,n_1,n_2}^{(1,1)}(\psi_{m-1,n_1} \otimes \psi_{m-1,n_2}))\end{aligned} \tag{10.5}$$

in the basis $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1}) \otimes (\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$ is

$$\left(\begin{array}{ccc|ccc} c_{m-1,0,0} & \cdots & & w_{m-1,0,0}^{(0,1)} & \cdots & \\ \vdots & \vdots & & \vdots & \vdots & \\ \hline w_{m-1,0,0}^{(1,0)} & \cdots & & w_{m-1,0,0}^{(1,1)} & \cdots & \\ \vdots & \vdots & & \vdots & \vdots & \end{array}\right). \tag{10.6}$$

The coordinate matrix is thus split into four submatrices:

- The c_{m-1}-values, i.e. the coordinates for $V_{m-1} \oplus V_{m-1}$. This is the upper left corner in Equation (10.6).
- The $w_{m-1}^{(0,1)}$-values, i.e. the coordinates for $W_{m-1}^{(0,1)}$. This is the upper right corner in Equation (10.6).
- The $w_{m-1}^{(1,0)}$-values, i.e. the coordinates for $W_{m-1}^{(1,0)}$. This is the lower left corner in Equation (10.6).

- The $w_{m-1}^{(1,1)}$-values, i.e. the coordinates for $W_{m-1}^{(1,1)}$. This is the lower right corner in Equation (10.6).

The $w_{m-1}^{(i,j)}$-values are as in the one-dimensional situation often refered to as wavelet coefficients. Let us consider the Haar wavelet as an example.

Example 10.10. *Piecewise constant functions.*

If V_m is the vector space of piecewise constant functions on any interval of the form $[k2^{-m}, (k+1)2^{-m})$ (as in the piecewise constant wavelet), $V_m \otimes V_m$ is the vector space of functions in two variables which are constant on any square of the form $[k_1 2^{-m}, (k_1+1)2^{-m}) \times [k_2 2^{-m}, (k_2+1)2^{-m})$. Clearly $\phi_{m,k_1} \otimes \phi_{m,k_2}$ is constant on such a square and 0 elsewhere, and these functions are a basis for $V_m \otimes V_m$.

Let us compute the orthogonal projection of $\phi_{1,k_1} \otimes \phi_{1,k_2}$ onto $V_0 \otimes V_0$. Since the Haar wavelet is orthonormal, the basis functions in (10.4) are orthonormal, and we can thus use the orthogonal decomposition formula to find this projection. Clearly $\phi_{1,k_1} \otimes \phi_{1,k_2}$ has different support from all except one of $\phi_{0,n_1} \otimes \phi_{0,n_2}$. Since

$$\langle \phi_{1,k_1} \otimes \phi_{1,k_2}, \phi_{0,n_1} \otimes \phi_{0,n_2} \rangle = \langle \phi_{1,k_1}, \phi_{0,n_1} \rangle \langle \phi_{1,k_2}, \phi_{0,n_2} \rangle = \frac{\sqrt{2}}{2}\frac{\sqrt{2}}{2} = \frac{1}{2}$$

when the supports intersect, we obtain

$$\mathrm{proj}_{V_0 \otimes V_0}(\phi_{1,k_1} \otimes \phi_{1,k_2}) = \begin{cases} \frac{1}{2}(\phi_{0,k_1/2} \otimes \phi_{0,k_2/2}) & \text{when } k_1, k_2 \text{ are even} \\ \frac{1}{2}(\phi_{0,k_1/2} \otimes \phi_{0,(k_2-1)/2}) & \text{when } k_1 \text{ is even, } k_2 \text{ is odd} \\ \frac{1}{2}(\phi_{0,(k_1-1)/2} \otimes \phi_{0,k_2/2}) & \text{when } k_1 \text{ is odd, } k_2 \text{ is even} \\ \frac{1}{2}(\phi_{0,(k_1-1)/2} \otimes \phi_{0,(k_2-1)/2}) & \text{when } k_1, k_2 \text{ are odd} \end{cases}$$

So, in this case there were 4 different formulas, since there were 4 different combinations of even/odd. Let us also compute the projection onto the orthogonal complement of $V_0 \otimes V_0$ in $V_1 \otimes V_1$, and let us express this in terms of the $\phi_{0,n}, \psi_{0,n}$, like we did in the one-variable case. Also here there are 4 different formulas. When k_1, k_2 are both even we obtain

$$\begin{aligned} &\phi_{1,k_1} \otimes \phi_{1,k_2} - \mathrm{proj}_{V_0 \otimes V_0}(\phi_{1,k_1} \otimes \phi_{1,k_2}) \\ &= \phi_{1,k_1} \otimes \phi_{1,k_2} - \frac{1}{2}(\phi_{0,k_1/2} \otimes \phi_{0,k_2/2}) \\ &= \left(\frac{1}{\sqrt{2}}(\phi_{0,k_1/2} + \psi_{0,k_1/2})\right) \otimes \left(\frac{1}{\sqrt{2}}(\phi_{0,k_2/2} + \psi_{0,k_2/2})\right) - \frac{1}{2}(\phi_{0,k_1/2} \otimes \phi_{0,k_2/2}) \\ &= \frac{1}{2}(\phi_{0,k_1/2} \otimes \phi_{0,k_2/2}) + \frac{1}{2}(\phi_{0,k_1/2} \otimes \psi_{0,k_2/2}) \\ &+ \frac{1}{2}(\psi_{0,k_1/2} \otimes \phi_{0,k_2/2}) + \frac{1}{2}(\psi_{0,k_1/2} \otimes \psi_{0,k_2/2}) - \frac{1}{2}(\phi_{0,k_1/2} \otimes \phi_{0,k_2/2}) \\ &= \frac{1}{2}(\phi_{0,k_1/2} \otimes \psi_{0,k_2/2}) + \frac{1}{2}(\psi_{0,k_1/2} \otimes \phi_{0,k_2/2}) + \frac{1}{2}(\psi_{0,k_1/2} \otimes \psi_{0,k_2/2}). \end{aligned}$$

Here we have used the relation $\phi_{1,k_i} = \frac{1}{\sqrt{2}}(\phi_{0,k_i/2} + \psi_{0,k_i/2})$, which we have from our first analysis of the Haar wavelet. Checking the other possibilities we find similar formulas for the projection onto the orthogonal complement of $V_0 \otimes V_0$ in $V_1 \otimes V_1$ when either k_1 or k_2 is odd. In all cases, the formulas use the basis functions for $W_0^{(0,1)}$, $W_0^{(1,0)}$, $W_0^{(1,1)}$. These functions are shown in Figure 10.1, together with the function $\phi \otimes \phi \in V_0 \otimes V_0$.

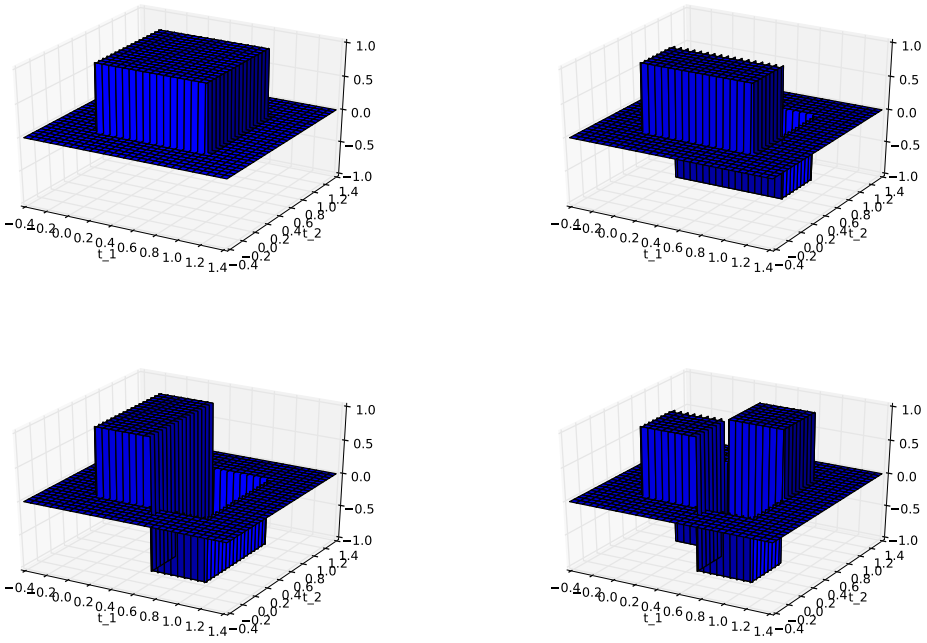

Figure 10.1: The functions $\phi \otimes \phi$, $\phi \otimes \psi$, $\psi \otimes \phi$, $\psi \otimes \psi$, which are bases for $(V_0 \otimes V_0) \oplus W_0^{(0,1)} \oplus W_0^{(1,0)} \oplus W_0^{(1,1)}$ for the Haar wavelet.

Example 10.11. *Piecewise linear functions.*

If we instead use any of the wavelets for piecewise linear functions, the wavelet basis functions are not orthogonal anymore, just as in the one-dimensional case. The new basis functions are shown in Figure 10.2 for the alternative piecewise linear wavelet.

An immediate corollary of Theorem 10.7 is the following:

Corollary 10.12. *Implementing tensor product.*

Let

$$A_m = P_{(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1}) \leftarrow \boldsymbol{\phi}_m}$$
$$B_m = P_{\boldsymbol{\phi}_m \leftarrow (\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})}$$

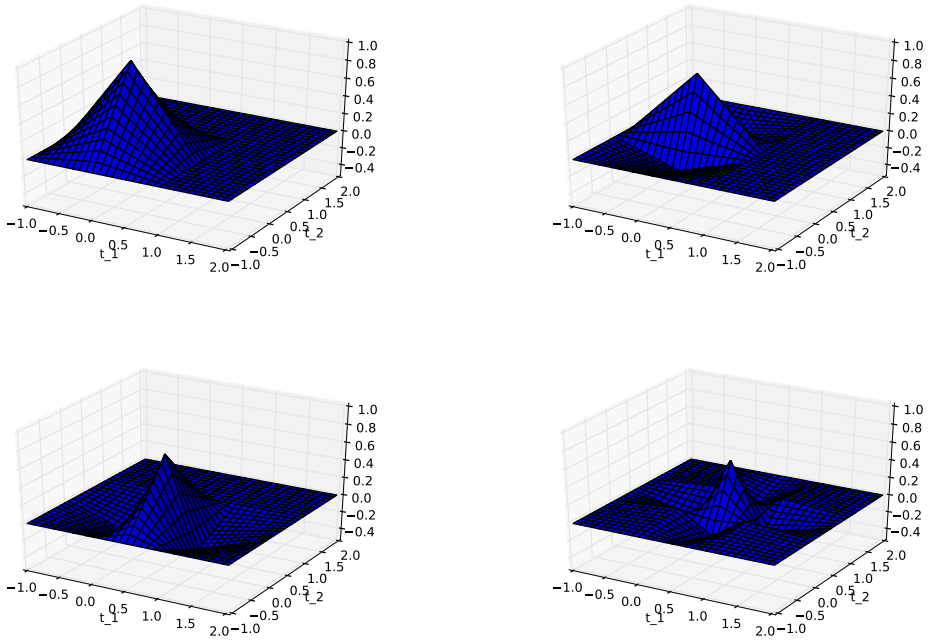

Figure 10.2: The functions $\phi \otimes \phi$, $\phi \otimes \psi$, $\psi \otimes \phi$, $\psi \otimes \psi$, which are bases for $(V_0 \otimes V_0) \oplus W_0^{(0,1)} \oplus W_0^{(1,0)} \oplus W_0^{(1,1)}$ for the alternative piecewise linear wavelet.

be the stages in the DWT and the IDWT, and let

$$X = (c_{m,i,j})_{i,j} \qquad Y = \begin{pmatrix} (c_{m-1,i,j})_{i,j} & (w^{(0,1)}_{m-1,i,j})_{i,j} \\ (w^{(1,0)}_{m-1,i,j})_{i,j} & (w^{(1,1)}_{m-1,i,j})_{i,j} \end{pmatrix} \tag{10.7}$$

be the coordinate matrices in $\boldsymbol{\phi}_m \otimes \boldsymbol{\phi}_m$, and $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1}) \otimes (\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$, respectively. Then

$$Y = A_m X A_m^T \tag{10.8}$$
$$X = B_m Y B_m^T \tag{10.9}$$

By the m-level two-dimensional DWT/IDWT (or DWT2/IDWT2) we mean the change of coordinates where this is repeated m times as in a DWT/IDWT.

It is straightforward to make implementations of DWT2 and IDWT2, in the same way we implemented `DWTImpl` and `IDWTImpl`. In Exercise 10.1 you will be asked to program functions `DW2TImpl` and `IDW2TImpl` for this. Each stage in DWT2 and IDWT2 can now be implemented by substituting the matrices A_m, B_m above into the code following Theorem 9.28. When using many levels of the DWT2, the next stage is applied only to the upper left corner of the

matrix, just as the DWT at the next stage only is applied to the first part of the coordinates. At each stage, the upper left corner of the coordinate matrix (which gets smaller at each iteration), is split into four equally big parts. This is illustrated in Figure 10.3, where the different types of coordinates which appear in the first two stages in a DWT2 are indicated.

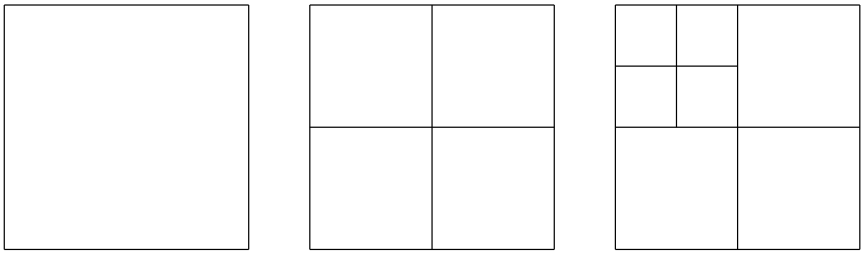

Figure 10.3: Illustration of the different coordinates in a two level DWT2 before the first stage is performed (left), after the first stage (middle), and after the second stage (right).

It is instructive to see what information the different types of coordinates in an image represent. In the following examples we will discard some types of coordinates, and view the resulting image. Discarding a type of coordinates will be illustrated by coloring the corresponding regions from Figure 10.3 black. As an example, if we perform a two-level DWT2 (i.e. we start with a coordinate matrix in the basis $\boldsymbol{\phi}_2 \otimes \boldsymbol{\phi}_2$), Figure 10.4 illustrates first the collection of all coordinates, and then the resulting collection of coordinates after removing subbands at the first level successively.

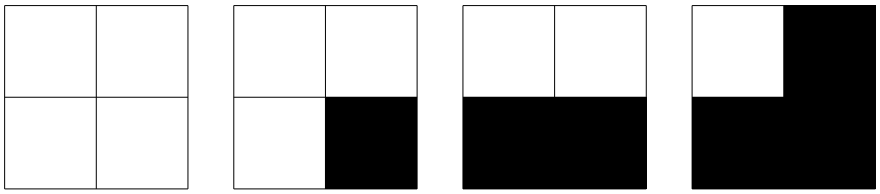

Figure 10.4: Graphical representation of neglecting the wavelet coefficients at the first level. After applying DWT2, the wavelet coefficients are split into four parts, as shown in the left figure. In the following figures we have removed coefficients from $W_1^{(1,1)}$, $W_1^{(1,0)}$, and $W_1^{(0,1)}$, in that order.

Figure 10.5 illustrates in the same way incremental removal of the subbands at the second level.

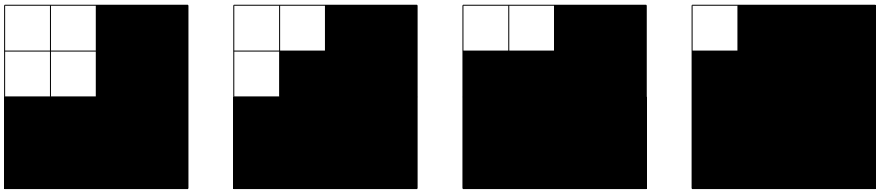

Figure 10.5: Graphical representation of neglecting the wavelet coefficients at the second level. After applying the second stage in DWT2, the wavelet coefficients from the upper left corner are also split into four parts, as shown in the left figure. In the following figures we have removed coefficients from $W_2^{(1,1)}$, $W_2^{(1,0)}$, and $W_2^{(0,1)}$, in that order.

Before we turn to experiments on images using wavelets, we would like to make another interpretation on the corners in the matrices after the DWT2, which correspond to the different coordinates $(c_{m-1,i,j})_{i,j}$, $(w^{(0,1)})_{m-1,i,j}$, $(w^{(1,0)})_{m-1,i,j}$, and $(w^{(1,1)})_{m-1,i,j}$. It turns out that these corners have natural interpretations in terms of the filter characterization of wavelets, as given in Chapter 6. Recall again that in a DWT2, the DWT is first applied to the columns in the image, then to the rows in the image. Recall first that the DWT2 applies first the DWT to all columns, and then to all rows in the resulting matrix.

First the DWT is applied to all columns in the image. Since the first half of the coordinates in a DWT are outputs from a lowpass filter H_0 (Theorem 6.3), the upper half after the DWT has now been subject to a lowpass filter to the columns. Similarly, the second half of the coordinates in a DWT are outputs from a highpass filter H_1 (Theorem 6.3 again), so that the bottom half after the DWT has been subject to a highpass filter to the columns.

Then the DWT is applied to all rows in the image. Similarly as when we applied the DWT to the columns, the left half after the DWT has been subject to the same lowpass filter to the rows, and the right half after the DWT has been subject to the same highpass filter to the rows.

These observations split the resulting matrix after DWT2 into four blocks, with each block corresponding to a combination of lowpass and highpass filters. The following names are thus given to these blocks:

- The upper left corner is called the LL-subband,
- The upper right corner is called the LH-subband,
- The lower left corner is called the HL-subband,
- The lower right corner is called the HH-subband.

The two letters indicate the type of filters which have been applied (L=lowpass, H=highpass). The first letter indicates the type of filter which is applied to the

columns, the second indicates which is applied to the rows. The order is therefore important. The name *subband* comes from the interpretation of these filters as being selective on a certain frequency band. In conclusion, a block in the matrix after the DWT2 corresponds to applying a combination of lowpass/higpass filters to the rows of the columns of the image. Due to this, and since lowpass filters extract slow variations, highpass filters abrupt changes, the following holds:

Observation 10.13. *Visual interpretation of the DWT2.*
After the DWT2 has been applied to an image, we expect to see the following:

- In the upper left corner, slow variations in both the vertical and horizontal directions are captured, i.e. this is a low-resolution version of the image.
- In the upper right corner, slow variations in the vertical direction are captured, together with abrupt changes in the horizontal direction.
- In the lower left corner, slow variations in the horizontal direction are captured, together with abrupt changes in the vertical direction.
- In the lower right corner, abrupt changes in both directions appear are captured.

These effects will be studied through examples in the next section.

What you should have learned in this section.

- The special interpretation of DWT2 applied to an image as splitting into four types of coordinates (each being one corner of the image), which represent lowpass/highpass combinations in the horizontal/vertical directions.

10.3 Experiments with images using wavelets

In this section we will make some experiments with images using the wavelets we have considered The wavelet theory is applied to images in the following way: We first visualize the pixels in the image as coordinates in the basis $\boldsymbol{\phi}_m \otimes \boldsymbol{\phi}_m$ (so that the image has size $(2^m M) \times (2^m N)$). As in the case for sound, this will represent a good approximation wehn m is large. We then perform a change of coordinates with the DWT2. As we did for sound, we can then either set the detail components from the $W_k^{(i,j)}$-spaces to zero, or the low-resolution approximation from $V_0 \otimes V_0$ to zero, depending on whether we want to inspect the detail components or the low-resolution approximation. Finally we apply the IDWT2 to end up with coordinates in $\boldsymbol{\phi}_m \otimes \boldsymbol{\phi}_m$ again, and display the new image with pixel values equal to these coordinates.

Example 10.14. *Applying the Haar wavelet to a very simple example image.*

Let us apply the Haar wavelet to the sample chess pattern example image from Figure 9.17. The lowpass filter of the Haar wavelet was essentially a smoothing filter with two elements. Also, as we have seen, the highpass filter essentially computes an approximation to the partial derivative. Clearly, abrupt changes in the vertical and horizontal directions appear here only at the edges in the chess pattern, and abrupt changes in both directions appear only at the grid points in the chess pattern. Due to Observation 10.13, after a DWT2 we expect to see the following:

- In the upper left corner, we should see a low-resolution version of the image.
- In the upper right corner, only the vertical edges in the chess pattern should be visible.
- In the lower left corner, only the horizontal edges in the chess pattern should be visible.
- In the lower right corner, only the grid points in the chess pattern should be visible.

In Figure 10.6 we have applied one level of the DWT2 to the chess pattern example image, and all these effects are seen clearly here.

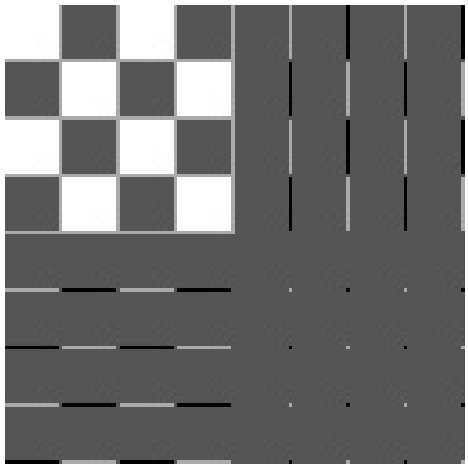

Figure 10.6: The chess pattern example image after application of the DWT2. The Haar wavelet was used.

Example 10.15. *Creating thumbnail images.*

Let us apply the Haar wavelet to our sample image. After the DWT2, the upper left submatrices represent the low-resolution approximations from

$V_{m-1} \otimes V_{m-1}$, $V_{m-2} \otimes V_{m-2}$, and so on. We can now use the following code to store the low-resolution approximation for $m = 1$:

```
DWT2Impl(X, 1, DWTKernelHaar)
X = X[0:(shape(X)[0]/2), 0:(shape(X)[1]/2)]
mapto01(X); X *= 255
```

Note that here it is necessary to map the result back to $[0, 255]$.

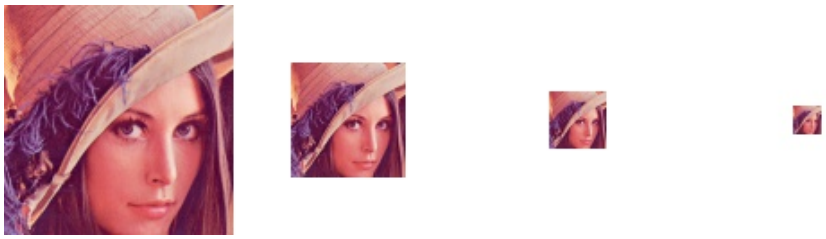

Figure 10.7: The corresponding thumbnail images for the Image of Lena, obtained with a DWT of 1, 2, 3, and 4 levels.

In Figure 10.7 the results are shown up to 4 resolutions. In Figure 10.8 we have also shown the entire result after a 1- and 2-stage DWT2 on the image. The first two thumbnail images can be seen as the the upper left corners of the first two images. The other corners represent detail.

Figure 10.8: The corresponding image resulting from a wavelet transform with the Haar-wavelet for $m = 1$ and $m = 2$.

Example 10.16. *Detail and low-resolution approximations with the Haar wavelet.*

In Exercise 10.4 you will be asked to implement a function `showDWT` which displays the low-resolution approximations or the detail components for our test

image for any wavelet, using functions we have previously implemented. Let us take a closer look at the images generated when the Haar wavelet is used. Above we viewed the low-resolution approximation as a smaller image. Let us compare with the image resulting from setting the wavelet detail coefficients to zero, and viewing the result as an image of the same size. In particular, let us neglect the wavelet coefficients as pictured in Figure 10.4 and Figure 10.5. Since the Haar wavelet has few vanishing moments, we should expect that the lower order resolution approximations from V_0 are worse when m increase. Figure 10.9 confirms this for the lower order resolution approximations.

Figure 10.9: Image of Lena, with detail at the first 1, 2, 3, and 4 levels zeroed out, respectively, for the Haar wavelet.

Alternatively, we should see that the higher order detail spaces contain more information. The new images when `showDWT` is used to display the detail components for the Haar wavelet are shown in Figure 10.10.

Figure 10.10: The corresponding detail for the images in Figure 10.9, with the Haar wavelet.

The black color indicates values which are close to 0. In other words, most of the coefficients are close to 0, which reflects one of the properties of the wavelet.

Example 10.17. *Experimenting with different wavelets.*

Using the function `showDWT`, we can display the low-resolution approximations at a given resolution of our image test file `lena.png`, for the Spline 5/3 and CDF 9/7 wavelets in addition to the Haar wavelet, with the following code:

```
showDWT(m, DWTKernelHaar, IDWTKernelHaar, True)
showDWT(m, DWTKernel53, IDWTKernel53, True)
showDWT(m, DWTKernel97, IDWTKernel97, True)
```

The first call to `showDWT` displays the result using the Haar wavelet. The second call to `showDWT` moves to the Spline 5/3 wavelet, and the third call uses the CDF

9/7 wavelet. We can repeat this for various number of levels m, and compare the different images.

Example 10.18. *The Spline 5/3 wavelet and removing bands in the detail spaces.*

Since the detail components now are split into three bands, another thing we can try is to neglect only parts of the detail components (i.e.e some of $W_m^{(1,1)}$, $W_m^{(1,0)}$, $W_m^{(0,1)}$), contrary to the one-dimensional case. Let us use the Spline 5/3 wavelet. The resulting images when the bands on the first level indicated in Figure 10.4 are removed are shown in Figure 10.11.

Figure 10.11: Image of Lena, with various bands of detail at the first level zeroed out. From left to right, the detail at $W_1^{(1,1)}$, $W_1^{(1,0)}$, $W_1^{(0,1)}$, as illustrated in Figure 10.4. The Spline 5/3 wavelet was used.

The resulting images when the bands on the second level indicated in Figure 10.5 are removed are shown in Figure 10.12.

Figure 10.12: Image of Lena, with various bands of detail at the second level zeroed out. From left to right, the detail at $W_2^{(1,1)}$, $W_2^{(1,0)}$, $W_2^{(0,1)}$, as illustrated in Figure 10.5. The Spline 5/3 wavelet was used.

The image is seen still to resemble the original one, even after two levels of wavelets coefficients have been neglected. This in itself is good for compression purposes, since we may achieve compression simply by dropping the given

coefficients. However, if we continue to neglect more levels of coefficients, the result will look poorer. In Figure 10.13 we have also shown the resulting image after the third and fourth level of detail have been neglected. Although we still can see details in the image, the quality in the image is definitely poorer.

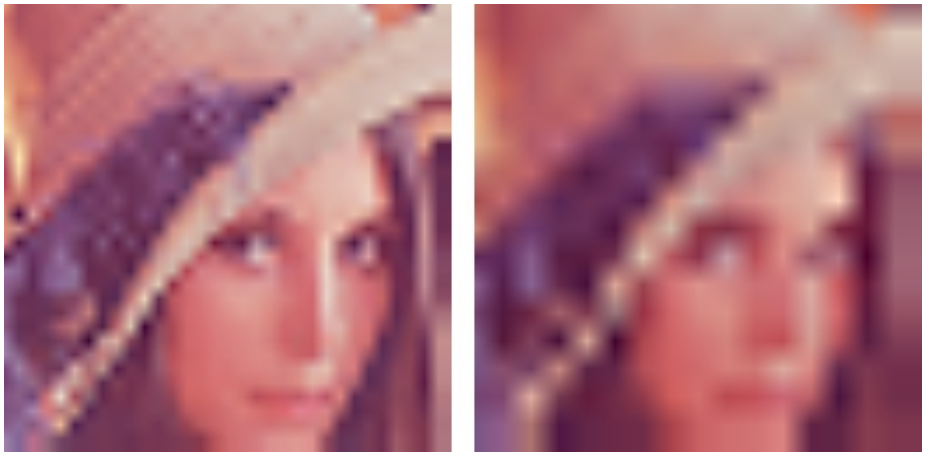

Figure 10.13: Image of Lena, with detail including level 3 and 4 zeroed out. The Spline 5/3 wavelet was used.

Although the quality is poorer when we neglect levels of wavelet coefficients, all information is kept if we additionally include the detail/bands. In Figure 10.14, we have shown the corresponding detail for the two right images in Figure 10.11, and Figure 10.13. Clearly, more detail can be seen in the image when more of the detail is included.

Example 10.19. *the CDF 9/7 wavelet.*

Let us repeat the previous example for the CDF 9/7 wavelet, using the function `showDWT` you implemented in Exercise 10.4. We should now see improved images when we discard the detail in the images. Figure 10.15 confirms this for the lower resolution spaces,

while Figure 10.16 confirms this for the higher order detail spaces.

As mentioned, the procedure developed in this section for applying a wavelet transform to an image with the help of the tensor product construction, is adopted in the JPEG2000 standard. This lossy (can also be used as lossless) image format was developed by the Joint Photographic Experts Group and published in 2000. After significant processing of the wavelet coefficients, the final coding with JPEG2000 uses an advanced version of arithmetic coding. At the cost of increased encoding and decoding times, JPEG2000 leads to as much as 20 % improvement in compression ratios for medium compression rates, possibly more for high or low compression rates. The artefacts are less visible than in JPEG and appear at higher compression rates. Although a number of components in JPEG2000 are patented, the patent holders have agreed that the

Figure 10.14: The corresponding detail for the image of Lena. The Spline 5/3 wavelet was used.

core software should be available free of charge, and JPEG2000 is part of most Linux distributions. However, there appear to be some further, rather obscure, patents that have not been licensed, and this may be the reason why JPEG2000 is not used more. The extension of JPEG2000 files is `.jp2`.

What you should have learned in this section.

- How to call functions which perform different wavelet transformations on an image.
- Be able to interpret the detail components and low-resolution approximations in what you see.

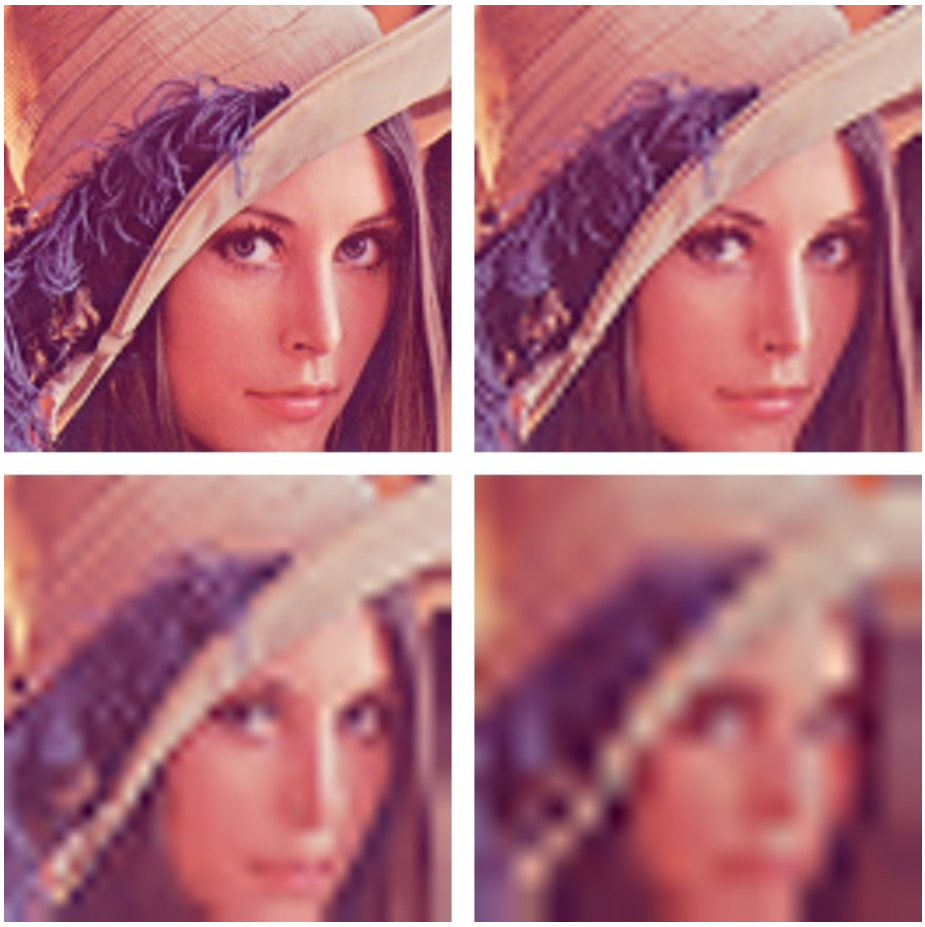

Figure 10.15: Image of Lena, with higher levels of detail neglected. The CDF 9/7 wavelet was used.

Exercise 10.1: Implement two-dimensional DWT

Implement functions `DW2TImpl` and `IDW2TImpl` which perform the m-level DWT2 and the IDWT2, respectively, on an image. The functions should take the same input as `DWTImpl` and `IDWTImpl`, with the input vector replaced with a two-dimensional object. The functions should at each stage call `DWTImpl` and `IDWTImpl` with $m = 1$, and each call to these functions should alter the appropriate upper left submatrix in the coordinate matrix. If the image has several color components, the functions should be applied to each color component. There are three color components in the test image 'lena.png'.

Figure 10.16: The corresponding detail for the image of Lena. The CDF 9/7 wavelet was used.

Exercise 10.2: Comment code

Assume that we have an image represented by the $M \times N$-matrix `X`, and consider the following code:

```
for n in range(N):
    c = (X[0:M:2, n] + X[0:M:2, n])/sqrt(2)
    w = (X[0:M:2, n] - X[0:M:2, n])/sqrt(2)
    X[:, n] = vstack([c, w])

for m in range(M):
    c = (X[m, 0:N:2] + X[m, 0:N:2])/sqrt(2)
    w = (X[m, 0:N:2] - X[m, 0:N:2])/sqrt(2)
    X[m, :] = hstack([c,w])
```

a) Comment what the code does, and explain what you will see if you display `X` as an image after the code has run.

b) The code above has an inverse transformation, which reproduce the original image from the transformed values which we obtained. Assume that you zero out the values in the lower left and the upper right corner of the matrix `X` after the code above has run, and that you then reproduce the image by applying this inverse transformation. What changes can you then expect in the image?

Exercise 10.3: Comment code

In this exercise we will use the filters $G_0 = \{\underline{1}, 1\}$, $G_1 = \{1, \underline{-1}\}$.

a) Let X be a matrix which represents the pixel values in an image. Define $\boldsymbol{x} = (1, 0, 1, 0)$ and $\boldsymbol{y} = (0, 1, 0, 1)$. Compute $(G_0 \otimes G_0)(\boldsymbol{x} \otimes \boldsymbol{y})$.

b) For a general image X, describe how the images $(G_0 \otimes G_0)X$, $(G_0 \otimes G_1)X$, $(G_1 \otimes G_0)X$, and $(G_1 \otimes G_1)X$ may look.

c) Assume that we run the following code on an image represented by the matrix `X`:

```
M, N = shape(X)
for n in range(N):
    c = X[0:M:2, n] + X[1:M:2, n]
    w = X[0:M:2, n] - X[1:M:2, n]
    X[:, n] = vstack([c,w])

for m in range(M):
    c = X[m, 0:N:2] + X[m, 1:N:2]
    w = X[m, 0:N:2] - X[m, 1:N:2]
    X[m, :] = hstack([c,w])
```

Comment the code. Describe what will be shown in the upper left corner of `X` after the code has run. Do the same for the lower left corner of the matrix. What is the connection with the images $(G_0 \otimes G_0)X$, $(G_0 \otimes G_1)X$, $(G_1 \otimes G_0)X$, and $(G_1 \otimes G_1)X$?

Exercise 10.4: Zeroint out DWT coefficients

In this exercise we will experiment with applying the m-level DWT2 to an image.

a) Write a function `showDWT`, which takes m, a DWT kernel `f`, an IDWT kernel `invf`, and a variable `lowres` as input, and

- reads the image file `lena.png`,
- performs an m-level DWT2 on the image samples using the function `DW2TImpl`, with DWT kernel `f`
- sets all wavelet coefficients representing detail to zero if `lowres` is true (i.e. keep only the low-resolution coordinates from $\boldsymbol{\phi}_0 \otimes \boldsymbol{\phi}_0$),

- sets all low-resolution coordinates to zero if `lowres` is false (i.e. keep only the detail coordinates),
- performs an IDWT2 on the resulting coefficients using the function `IDW2TImpl`, with IDWT kernel `invf`,
- displays the resulting image.

b) Do the image samples returned by `showDWT` lie in $[0, 255]$?

c) Run the function `showDWT` for different values of m for the Haar wavelet, with `lowres` set to true. Describe what you see for different m. For which m can you see that the image gets degraded? How does it get degraded? Compare with what you saw with the function `showDCThigher` in Exercise 9.18, where you performed a DCT on the image samples instead, and set DCT coefficients below a given threshold to zero.

d) Repeat what you did in c., but this time with `lowres` set to false instead. What kind of image do you see now? Can you recognize the original image in what you see? Try to explain why the images seem to get clearer when you increase m.

e) In the code in Example 10.17, set `lowres` to false in the call to `showDWT` also for the other wavelets. and repeat what you did in d..

Exercise 10.5: Experiments on a test image

In Figure 10.17 we have applied the DWT2 with the Haar wavelet to an image very similar to the one you see in Figure 10.6. You see here, however, that there seems to be no detail components, which is very different from Figure 10.6, even though the images are very similar. Attempt to explain what causes this to happen.

Hint. Compare with Exercise 5.17.

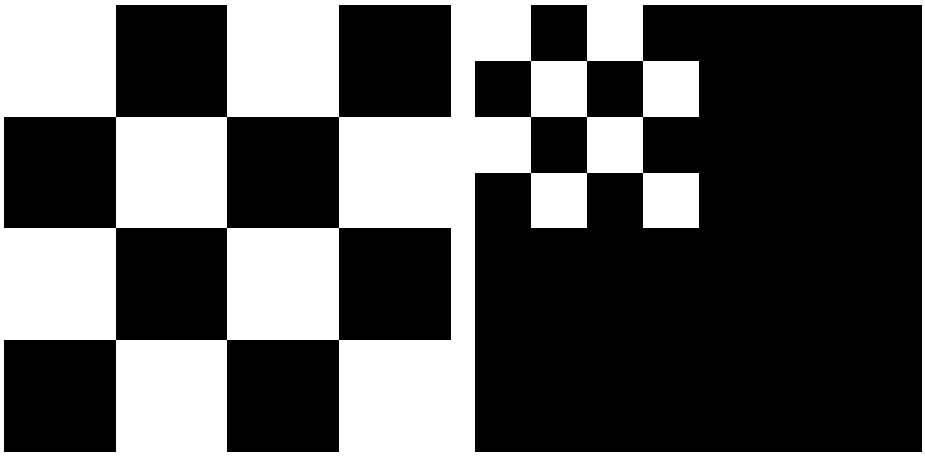

Figure 10.17: A simple image before and after one level of the DWT2. The Haar wavelet was used.

10.4 An application to the FBI standard for compression of fingerprint images

In the beginning of the 1990s, the FBI had a major problem when it came to their archive of fingerprint images. With more than 200 million fingerprint records, their digital storage exploded in size, so that some compression strategy needed to be employed. Several strategies were tried, for instance the widely adopted JPEG standard. The problem with JPEG had to do with the blocking artefacts, which we saw in Section 9.4. Among other strategies, FBI chose a wavelet-based strategy due to its nice properties. The particular way wavelets are applied in this strategy is called *Wavelet transform/scalar quantization* (WSQ).

Fingerprint images are a very specific type of images, as seen in Figure 10.18. They differ from natural images by having a large number of abrupt changes. One may ask whether other wavelets than the ones we have used up to now are more suitable for compressing such images. After all, the technique of vanishing moments we have used for constructing wavelets are most suitable when the images display some regularity (as many natural images do). Extensive tests were undertaken to compare different wavelets, and the CDF 9/7 wavelet used by JPEG2000 turned out to perform very well, also for fingerprint images. One advantage with the choice of this wavelet for the FBI standard is that one then can exploit existing wavelet transformations from the JPEG2000 standard.

Besides the choice of wavelet, one can also ask other questions in the quest to compress fingerprint images: What number of levels is optimal in the application of the DWT2? And, while the levels in a DWT2 (see Figure 10.3) have an interpretation as change of coordinates, one can apply a DWT2 to the other subbands as well. This can not be interpreted as a change of coordinates, but if we assume that these subbands have the same characteristics as the original

Figure 10.18: A typical fingerprint image.

image, the DWT2 will also help us with compression when applied to them. Let us illustrate how the FBI standard applies the DWT2 to the different subbands. We will split this process into five stages. The subband structures and the resulting images after stage 1-4 are illustrated in Figure 10.19 and in Figure 10.20, respectively.

1. First apply the first stage in a DWT2. This gives the upper left corners in the two figures.

2. Then apply a DWT2 to all four resulting subbands. This is different from the DWT2, which only continues on the upper left corner. This gives the upper right corners in the two figures.

3. Then apply a DWT2 in three of the four resulting subbands. This gives the lower left corners.

4. In all remaining subbands, the DWT2 is again applied. This gives the lower right corners.

Now for the last stage. A DWT2 is again applied, but this time only to the upper left corner. The subbands are illustrated in Figure 10.21, and in Figure 10.22 the resulting image is shown.

When establishing the standard for compression of fingerprint images, the FBI chose this subband decomposition. In Figure 10.23 we also show the corresponding low resolution approximation and detail.

As can be seen from the subband decomposition, the low-resolution approximation is simply the approximation after a five stage DWT2.

The original JPEG2000 standard did not give the possibility for this type of subband decomposition. This has been added to a later extension of the

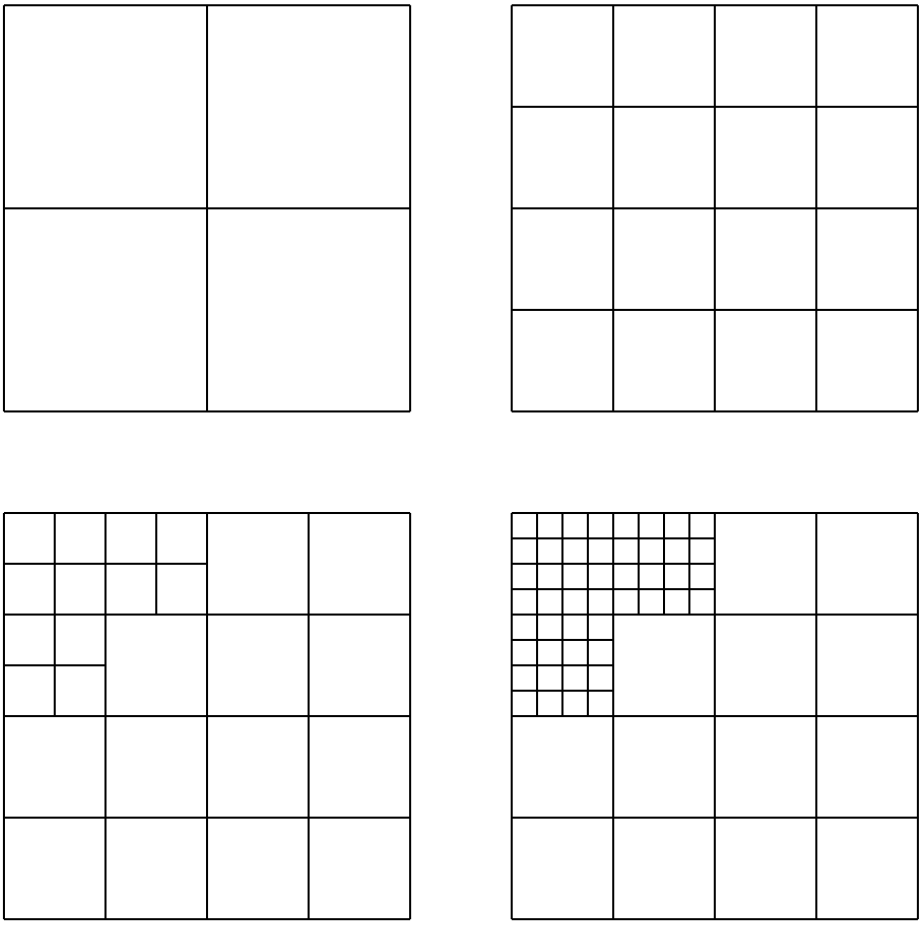

Figure 10.19: Subband structure after the different stages of the wavelet applications in the FBI fingerprint compression scheme.

standard, which makes the two standards more compatible. IN FBI's system, there are also other important parts besides the actual compression strategy, such as *fingerprint pattern matching*: In order to match a fingerprint quickly with the records in the database, several characteristics of the fingerprints are stored, such as the number of lines in the fingerprint, and points where the lines split or join. When the database is indexed with this information, one may not need to decompress all images in the database to perform matching. We will not go into details on this here.

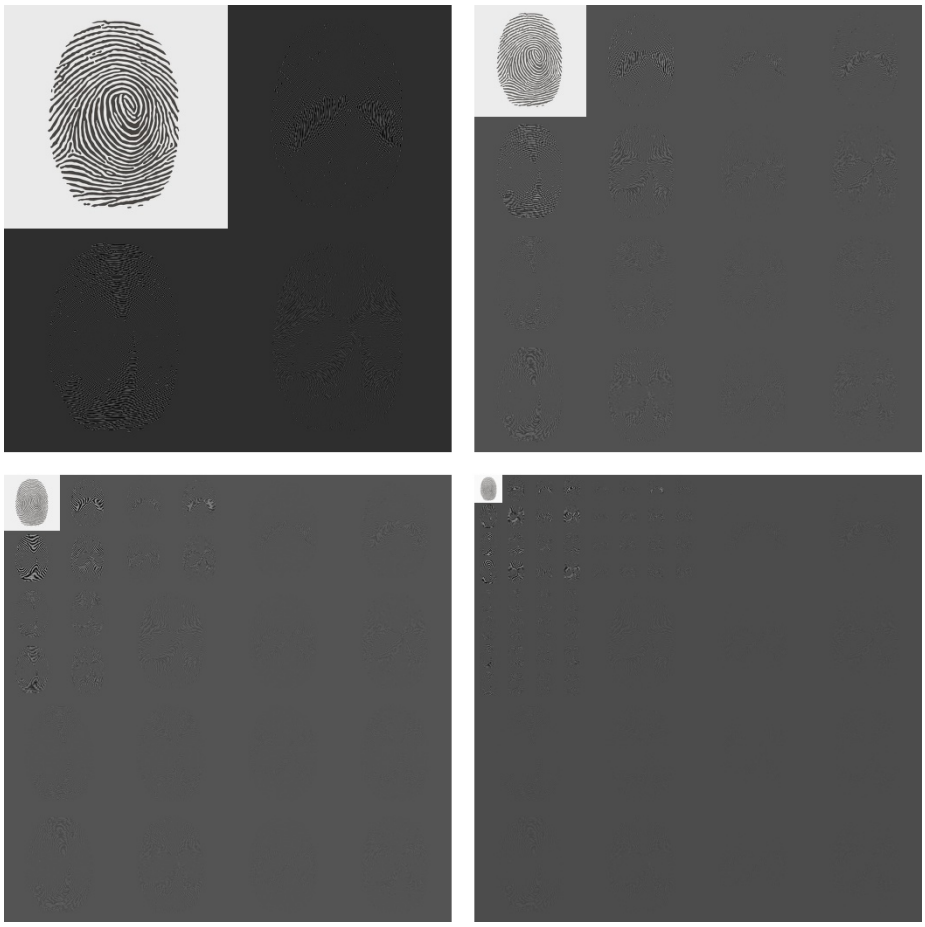

Figure 10.20: The fingerprint image after several DWT's.

Exercise 10.6: Implement the fingerprint compression scheme

Write code which generates the images shown in figures 10.20, 10.22, and 10.23. Use the functions `DW2TImpl` and `IDW2TImpl` with the CDF 9/7 wavelet kernel functions as input.

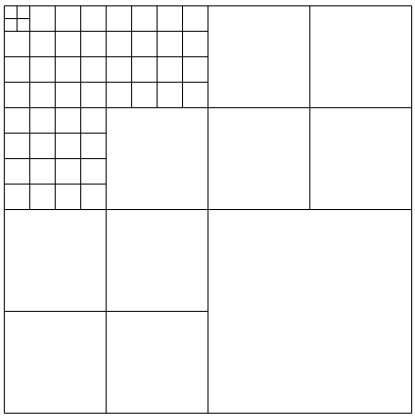

Figure 10.21: Subbands structure after all stages.

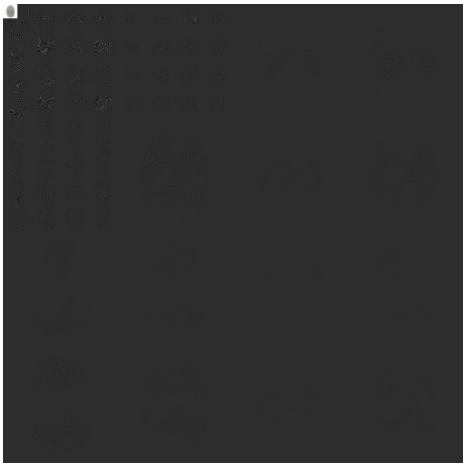

Figure 10.22: The resulting image obtained with the subband decomposition employed by the FBI.

Figure 10.23: The low-resolution approximation and the detail obtained by the FBI standard for compression of fingerprint images, when applied to our sample fingerprint image.

10.5 Summary

We extended the tensor product construction to functions by defining the tensor product of functions as a function in two variables. We explained with some examples that this made the tensor product formalism useful for approximation of functions in several variables. We extended the wavelet transform to the tensor product setting, so that it too could be applied to images. We also performed several experiments on our test image, such as creating low-resolution images and neglecting wavelet coefficients. We also used different wavelets, such as the Haar wavelet, the Spline 5/3 wavelet, and the CDF 9/7 wavelet. The experiments confirmed what we previously have proved, that wavelets with many vanishing moments are better suited for compression purposes.

The specification of the JPGE2000 standard can be found in [17]. In [36], most details of this theory is covered, in particular details on how the wavelet coefficients are coded (which is not covered here).

One particular application of wavelets in image processing is the compression of fingerprint images. The standard which describes how this should be performed can be found in [11]. In [2], the theory is described. The book [13] uses the application to compression of fingerprint images as an example of the usefulness of recent developments in wavelet theory.

Appendix A

Basic Linear Algebra

This book assumes that the student has taken a beginning course in linear algebra at university level. In this appendix we summarize the most important concepts one needs to know from linear algebra. Note that what is listed here should not be considered as a substitute for such a course: It is important for the student to go through a full course in linear algebra, in order to get good intuition for these concepts through extensive exercises. Such exercises are omitted here.

A.1 Matrices

An $m \times n$-matrix is simply a set of mn numbers, stored in m rows and n columns. We write a_{kn} for the entry in row k and column n of the matrix A. The zero matrix, denoted $\mathbf{0}$ is the matrix with all zeroes. A square matrix (i.e. where $m = n$) is said to be diagonal if $a_{kn} = 0$ whenever $k \neq n$. The identity matrix, denoted I, or I_n to make the dimension of the matrix clear, is the diagonal matrix where the entries on the diagonal are 1, the rest zeroes. If A is a matrix we will denote the transpose of A by A^T. If A is invertible we denote its inverse by A^{-1}. We say that a matrix A is *orthogonal* if $A^T A = AA^T = I$. A matrix is called sparse if most of the entries in the matrix are zero.

A.2 Vector spaces

A set of vectors V is called a vector space if ... We say that the vectors $\{\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_{n-1}\}$ are *linearly independent* if, whenever $\sum_{i=0}^{n-1} c_i \boldsymbol{v}_i = \mathbf{0}$, we must have that all $c_i = 0$. We will say that a set of vectors $\mathcal{B} = \{\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_{n-1}\}$ from V is a *basis* for V if the vectors are linearly independent, and span V.

Subspaces of $\mathbb{R}^N$, and function spaces.

A.3 Inner products and orthogonality

Most vector spaces in this book are inner product spaces. A (real) inner product on a vector space is a binary operation, written as $(\boldsymbol{u}, \boldsymbol{v}) \to \langle \boldsymbol{u}, \boldsymbol{v} \rangle$, which fulfills the following properties for any vectors $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$:

- $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \langle \boldsymbol{v}, \boldsymbol{u} \rangle$
- $\langle \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w} \rangle = \langle \boldsymbol{u}, \boldsymbol{w} \rangle + \langle \boldsymbol{v}, \boldsymbol{w} \rangle$
- $\langle c\boldsymbol{u}, \boldsymbol{v} \rangle = c\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ for any scalar c
- $\langle \boldsymbol{u}, \boldsymbol{u} \rangle \geq 0$, and $\langle \boldsymbol{u}, \boldsymbol{u} \rangle = 0$ if and only if $\boldsymbol{u} = 0$.

$\boldsymbol{u}$ and $\boldsymbol{v}$ are said to be *orthogonal* if $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \mathbf{0}$. In this book we have seen two important examples of inner product spaces. First of all the Euclidean inner product, which is defined by

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \sum_{i=0}^{n-1} u_i v_i \tag{A.1}$$

for any $\boldsymbol{u}$, $\boldsymbol{v}$ in $\mathbb{R}^n$. For functions we have seen examples which are variants of the following form:

$$\langle f, g \rangle = \int f(t)g(t)dt. \tag{A.2}$$

Any set of mutually orthogonal elements are also linearly independent. A basis where all basis vectors are mutually orthogonal is called an *orthogonal basis*. If additionally the vectors all have length 1, we say that the basis is *orthonormal*. If $\boldsymbol{x}$ is in a vector space with an orthogonal basis $\mathcal{B} = \{\boldsymbol{v}_k\}_{k=0}^{n-1}$, we can express $\boldsymbol{x}$ as

$$\frac{\langle \boldsymbol{x}, \boldsymbol{v}_0 \rangle}{\langle \boldsymbol{v}_0, \boldsymbol{v}_0 \rangle} \boldsymbol{v}_0 + \frac{\langle \boldsymbol{x}, \boldsymbol{v}_1 \rangle}{\langle \boldsymbol{v}_1, \boldsymbol{v}_1 \rangle} \boldsymbol{v}_1 + \cdots + \frac{\langle \boldsymbol{x}, \boldsymbol{v}_{n-1} \rangle}{\langle \boldsymbol{v}_{n-1}, \boldsymbol{v}_{n-1} \rangle} \boldsymbol{v}_{n-1}. \tag{A.3}$$

In other words, the weights in linear combinations are easily found when the basis is orthogonal. This is also called the *orthogonal decomposition theorem*.

By the *projection* of a vector $\boldsymbol{x}$ onto a subspace U we mean the vector $\boldsymbol{y} = \text{proj}_U \boldsymbol{x}$ which minimizes the distance $\|\boldsymbol{y} - \boldsymbol{x}\|$. If $\boldsymbol{v}_i$ is an orthogonal basis for U, we have that $\text{proj}_U \boldsymbol{x}$ can be written by Equation (A.3).

A.4 Coordinates and change of coordinates

If $\mathcal{B} = \{\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_{n-1}\}$ is a basis for a vector space, and $\boldsymbol{x} = \sum_{i=0}^{n-1} x_i \boldsymbol{v}_i$, we say that $(x_0, x_1, \ldots, x_{n-1})$ is the *coordinate vector* of $\boldsymbol{x}$ w.r.t. the basis $\mathcal{B}$. We also write $[\boldsymbol{x}]_{\mathcal{B}}$ for this coordinate vector.

If $\mathcal{B}$ and $\mathcal{C}$ are two different bases for the same vector space, we can write down the two coordinate vectors $[\boldsymbol{x}]_{\mathcal{B}}$ and $[\boldsymbol{x}]_{\mathcal{C}}$. A useful operation is to transform the coordinates in $\mathcal{B}$ to those in $\mathcal{C}$, i.e. apply the transformation which sends $[\boldsymbol{x}]_{\mathcal{B}}$ to $[\boldsymbol{x}]_{\mathcal{C}}$. This is a linear transformation, and we will denote the $n \times n$-matrix of this linear transformation by $P_{\mathcal{C}\leftarrow\mathcal{B}}$, and call this the *change of coordinate matrix* from $\mathcal{B}$ to $\mathcal{C}$ In other words, the change of coordinate matrix is defined by requiring that

$$P_{\mathcal{C}\leftarrow\mathcal{B}}[\boldsymbol{x}]_{\mathcal{B}} = [\boldsymbol{x}]_{\mathcal{C}}. \tag{A.4}$$

It is straightforward to show that $P_{\mathcal{C}\leftarrow\mathcal{B}} = (P_{\mathcal{B}\leftarrow\mathcal{C}})^{-1}$, so that matrix inversion can be used to compute the change of coordinate matrix the opposite way. It is also straightforward to show that the columns in the change of coordinate matrix can be obtained by expressing the old basis in terms of the new basis, i.e. finding the vectors $[P_{\mathcal{B}\leftarrow\mathcal{C}}(\boldsymbol{v}_i)]_{\mathcal{C}}$.

If L is a linear transformation between the spaces V and W, and $\mathcal{B}$ is a basis for V, $\mathcal{C}$ a basis for W, we can consider the operation which sends the coordinates of $\boldsymbol{v} \in V$ in the basis $\mathcal{B}$ to the coordinates of $L\boldsymbol{v} \in W$ in the basis $\mathcal{C}$. This is represented by a matrix, called *the matrix of L relative to the bases $\mathcal{B}$ and $\mathcal{C}$*. Similarly to change of coordinate matrices, the columns of the matrix of L relative to the bases $\mathcal{B}$ and $\mathcal{C}$ are given by $[L(\boldsymbol{v}_i)]_{\mathcal{C}}$.

A.5 Eigenvectors and eigenvalues

If A is a linear transformation from a vector space to itself, a vector $\boldsymbol{v}$ is called an *eigenvector* if there exists a scalar λ so that $A\boldsymbol{v} = \lambda\boldsymbol{v}$. λ is called the corresponding eigenvalue.

If the matrix A is symmetric, the following hold:

- The eigenvalues of A are real,
- the eigenspaces of A are orthonormal,
- any vector can be decomposed as a sum of eigenvectors from A.

Fo non-symmetric matrices, these results do not hold in general. But for filters, clearly the second and third property always hold, regardless of whether the filter is symmetric or not.

A.6 Diagonalization

One can show that, for a symmetric matrix, $A = PDP^T$ where D is a digonal matrix and the eigenvalues of A are the values on the diagonal of D, and P is a matrix where the columns are the eigenvectors of A, with corresponding eigenvalue appearing in the same column in D.

Appendix B

Signal processing and linear algebra: a translation guide

This book should not be considered as a standard signal processing textbook. There are several reasons for this. First of all, much signal processing literature is written for people with an engineering background. This book is written for people with a basic linear algebra background. Secondly, the book does not give a comprehensive treatment of all basic signal processing concepts. Signal processing concepts are introduced whenever they are needed to encompass the mathematical exposition. In order to learn more about the different signal processing concepts, the reader can consult many excellent textbooks, such as [28, 1, 25, 32]. The translation guide of this chapter may be of some help in this respect, when one tries to unify material presented here with material from these signal processing textbooks. The translation guide handles both differences in notation between this book and signal processing literature, and topical differences. Most topical differences are also elaborated further in the summaries of the different chapters. The book has adopted most of its notation and concepts from mathematical literature.

B.1 Complex numbers

There are several differences between engineering literature and mathematics. In mathematics literature, i is used for the imaginary complex number which satisfies $i^2 = -1$. In engineering literature, the name j is used instead.

B.2 Functions

What in signal processing are refered to as continuous-time signals, are here refered to as functions. Usually we refer to a function by the letter f, according

to the mathematical tradition. The variable is mostly time, represented by the symbol t.

In signal processing, one often uses capital letters to denote a function which is the Fourier transform of another function, so that the Fourier ransform of x would be denoted by X. Here we simply denote a periodic function by its Fourier coefficients y_n, and we avoid the CTFT. We use analog filters, however, which also work in continuous time. Analog filters preserve frequencies, and we have used ν to denote frequency (variations per second), and not used angular frequency ω. In signal processing literature it is common to junp between the two.

B.3 Vectors

Discrete-time signals, as they are used in signal processing, are here mostly refered to as vectors. To as big extent as possible, we have attempted to keep vectors finite-dimensional. Vectors are in boldface (i.e. $\boldsymbol{x}$), but its elements are not in boldface, and with subscripts (i.e. x_n). Superscripts are also used to differ between vectors with the same base name (i.e. $\boldsymbol{x}^{(1)}$, $\boldsymbol{x}^{(2)}$ etc.), so that this does not interfer with the vector indices. In signal processing literature the corresponding notation would be x for the signal, and $x[n]$ for its elements, and signals with equal base names could be named like $x_1[n], x_2[n]$.

We have sometimes denoted the Fourier transform of $\boldsymbol{x}$ by $\widehat{\boldsymbol{x}}$, according to the mathematical tradition. More often we have distuinguished between a vector and its Discrete Fourier transform by using $\boldsymbol{x}$ for the first, and $\boldsymbol{y}$ for the latter. This also makes us distuinguish between the input and output to a filter, where we instead use $\boldsymbol{z}$ for the latter. Much signal processing literature write (capital) X for the DFT of the vector x.

B.4 Inner products and orthogonality

Throughout the book we have defined inner products for functions (for Fourier analysis and wavelets), and we have also used the standard inner product of $\mathbb{R}^N$. from this we have deduced the orthogonality of several basis functions used in signal processing theory. That the functions are orthogonal, as well as the inner product itself are, however, often not commented on in signal processing literature. As an unfortunate consequence, one has to explain the expression for the Fourier series using other means than the orthogonal decomposition formula and the least squares method. Also, one does not mention that the DFT is a unitary transformation.

B.5 Matrices and filters

Boldface notation is not used for matrices, according to the mathematical tradition. In signal processing, it is not common to formulate matrix equations,

such as for the DFT and DCT, or matrix factorizations. Instead one typically writes down each equation, one equation for each row in $\boldsymbol{y} = A\boldsymbol{x}$, i.e. not recognizing matrix/vector multiplication. We have sticked to the name filtering operations, but made it clear that this is nothing but a linear transformation with a Toeplitz matrix as its matrix. In particular, we alternately use the terms filtering and multiplication with a Toeplitz matrix. The characterization of filters as circulant Toeplitz matrices is usually not done in signal processing literature (but see [13]). In this text we allow for matrices also to be of infinite dimensions, expanding on the common use in linear algebra. When infinite dimensions are assumed, infinite in both directions is assumed. Matrices are scaled if necessary to make them unitary, in particular the DCT and the DFT. This scaling is usually not done in signal processing literature.

Representing a filter in terms of a finite matrix and restriction of a filter to a finite signal. This is usually omitted in signal processing literature.

One of the most important statements in signal processing is that convolution in time is equivalent to multiplication in frequency. We have presented a compelling interpretation of this in linear algebra terms. Since the frequency response simply are eigenvalues of the filter, and convolution simply is matrix factorization, multiplication in frequency simply means to multiply two diagonal matrices to obtain the frequency response of the product. Moreover, the Fourier basis vectors can be interpreted as eigenvectors.

B.6 Convolution

While we have defined the concept of convolution, readers familiar with signal processing may have noticed that this concept has not been used much. The reason is that we have wanted to present convolution as a matrix multiplication (to adapt to mathematical tradition), and that we have used the concept of filtering often instead. In signal processing literature one defines convolution in terms of vectors of infinite length. We have avoided this, since in practice vectors always need to be truncated to finite lengths. Due to this, we also have analyzed how a finite vector may be turned into a periodic vector (periodic or symmetric extension), and how this affects our analysis. Also we have concentrated on FIR-filters, and this makes us avoid convergence issues. Note that we do not present matrix multiplication as a method of implementing filtering, due to the special structure of this operation. We do not suggest other methods for implementation than applying the convolution formula in a brute-force way, or factoring the filter in simpler components.

B.7 Polyphase factorizations and lifting

In signal processing literature, it is not common to associate polyphase components with matrices, but rather with Laurent polynomials generated from the corresponding filter. The Laurent polynomial is nothing else than the Z-

transform of the associated filter. Associating polyphase components with blocks in a block matrix makes this book fit with block matrix methods in linear algebra textbooks.

The polyphase factorization serves two purposes in this book. Firstly, the lifting factorization (as used for wavelets) is derived from it, and put in a linear algebra framework as a factorization into sparse matrices, similarly to the FFT factorization. Thereby it fits together with many of the matrix factorization results from classical linear algebra, where also sparsity is what makes the factorization good for computation.

Secondly, the polyphase factorization of the filter bank transforms in the MP3 standard are derived (also as a sparse matrix factorization), and from this it is apparent what properties to put on the prototype filters in order to obtain useful transforms. In fact, from this factorization it became apparent that the MP3 filter bank transforms could be expressed in terms of alternative QMF filter banks (i.e. $M = 2$).

These two topics (lifting and the MP3 filter bank transform polyphase factorization) are usually not presented in a unified way in textbooks. we see here that there is a big advantage of doing this, since the second can build on theory from the first.

B.8 Transforms in general

In signal processing, one often refers to the forward and reverse filter bank transforms as analysis and synthesis, respectively, and for obvious reasons. In mathematical literature, one instead often use the term change of coordinates in a wavelet setting. These terms are not normally used in mathematical literature, where the term basis vectors/change of coordinate matrices would be used instead. Also, the output from a forward filter bank transform is often refered to as the *transformed vector*, and the result we get when we apply the reverse filter bank transform to this is called the *reconstructed vector*.

This exposition takes extra care in presenting how the DCT is derived naturally from the DFT. In particular both the DFT and the DCT are derived as matrices of eigenvectors for finite-dimensional filters. The DCT is derived from the DFT in that one restricts to a certain subset of vectors. The orthogonality of these matrices follows from the orthogonality of distinct eigenspaces.

B.9 Perfect reconstruction systems

The term biorthogonality is not used to describe a mutual property of the filters of wavelets. Borthogonality corresponds simply to two matrices being inverses of oneanother. For the same reason, the term perfect reconstruction is not used much. Much wavelet theory refer to a property called delay normalization. This terms has been avoided by mostly considering wavelets with symmetric filters,

for which delay-normalization is automatic. There are, however, many examples of wavelets where this term is important.

B.10 Z-transform and frequency response

The Z-transform and the frequency response are much used in signal processing literaure, and are important concepts for filter design. We have deliberately dropped the Z-transform. Due to this, much signal processing has of course been left out, since placements of poles and zeroes are not performed outside or inside the unit circle, since the frequency response only captures the values on the unit circle. Placement of poles and circles is perhaps the most-used design feature in filter design. The focus here is on implementing filters, not designing them, however.

In signal processing literature, the DTFT and the Z-transform is used, assuming that the inputs and outputs are vectors of infinite length. In practice of course, some truncation is needed, since only finite-dimensional arithmetic is performed by the computer. How this truncation is to be done without affecting the computations is thus never mentioned in signal processing, although it is always performed somehow. This exposition shows that this truncation can be taken as part of the theory, without seriously affecting the results.

Nomenclature

symbol	definition
T	Period of a function
ν	Frequency
f_N	Nth order Fourier series of f
$V_{N,T}$	Nth order Fourier space
$\mathcal{D}_{N,T}$	Order N real Fourier basis for $V_{N,T}$
$\mathcal{F}_{N,T}$	Order N complex Fourier basis for $V_{N,T}$
$\breve{f}$	Symmetric extension of the function f
$\lambda_s(\nu)$	Frequency response of an analog filter
f_s	Sampling frequency
T_s	Sampling period
N	Number of points in a DFT/DCT
$\mathcal{F}_N = \{\boldsymbol{\phi}_0, \boldsymbol{\phi}_1, \cdots, \boldsymbol{\phi}_{N-1}\}$	Fourier basis for $\mathbb{R}^N$
F_N	$Nimes N$-Fourier matrix
$\hat{\boldsymbol{x}}$	DFT of the vector $\boldsymbol{x}$
$\overline{A}$	Conjugate of a matrix
A^H	Conjugate transpose of a matrix
$\boldsymbol{x}^{(e)}$	Vector of even samples
$\boldsymbol{x}^{(o)}$	Vector of odd samples
$O(N)$	Order of an algorithm
$l(S)$	Length of a filter
$\boldsymbol{x} * \boldsymbol{y}$	Convolution of vectors
$\lambda_{S,n}$	Vector frequency response of a digital filter
E_d	Filter which delays with d samples
ω	Angular frequency
$\lambda_S(\omega)$	Continuous frequency response of a digital filter
$\breve{\boldsymbol{x}}$	Symmetric extension of a vector
S_r	Symmetric restriction of S
S^f	Matrix with the columns reversed
$\mathcal{D}_N = \{\boldsymbol{d}_0, \boldsymbol{d}_1, \cdots, \boldsymbol{d}_{N-1}\}$	N-point DCT basis for $\mathbb{R}^N$
DCT_N	$N \times N$-DCT matrix

symbol	definition
ϕ	Scaling function
V_m	Resolution space
r$\boldsymbol{\phi}_m$	Basis for V_m
$c_{m,n}$	Coordinates in $\boldsymbol{\phi}_m$
W_m	Detail space
r$U \oplus V$	Direct sum of vector spaces
$\boldsymbol{\psi}_m$	Basis for W_m
$w_{m,n}$	Coordinates in $\boldsymbol{\psi}_m$
$\mathcal{C}_m$	Reordering of $(\boldsymbol{\phi}_{m-1}, \boldsymbol{\psi}_{m-1})$
$\tilde{\phi}$	Dual scaling function
$\tilde{\psi}$	Dual mother wavelet
$\tilde{V}_m$	Dual resolution space
$\tilde{W}_m$	Dual detail space
$\mathcal{D}_m$	Reordering of $\boldsymbol{\phi}_m$
$\mathcal{E}_N = \{\boldsymbol{e}_0, \boldsymbol{e}_1, \cdots, \boldsymbol{e}_{N-1}\}$	Standard basis for $\mathbb{R}^N$
$\otimes$	Tensor product
$W_m^{(0,1)}$	Resolution m Complementary wavelet space, LH
$W_m^{(1,0)}$	Resolution m Complementary wavelet space, HL
$W_m^{(1,1)}$	Resolution m Complementary wavelet space, HH
A^T	Transpose of a matrix
A^{-1}	Inverse of a matrix
$\langle \boldsymbol{u}, \boldsymbol{v} \rangle$	Inner product
$[\boldsymbol{x}]_{\mathcal{B}}$	Coordinate vector of $\boldsymbol{x}$ relative to the basis $\mathcal{B}$
$P_{\mathcal{C} \leftarrow \mathcal{B}}$	Change of coordinate matrix from $\mathcal{B}$ to $\mathcal{C}$

Bibliography

[1] A. Ambardar. *Digital Signal Processing: a Modern Introduction*. Cengage Learning, 2006.

[2] C. M. Brislawn. Fingerprints go digital. *Notices of the AMS*, 42(11):1278–1283, 1995.

[3] B. A. Cipra. The best of the 20th century: Editors name top 10 algorithms. *SIAM News*, 33(4), 2000. http://www.uta.edu/faculty/rcli/TopTen/topten.pdf.

[4] A. Cohen and I. Daubechies. Wavelets on the interval and fast wavelet transforms. *Applied and computational harmonic analysis*, 1:54–81, 1993.

[5] A. Cohen, I. Daubechies, and J-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Appl. Math.*, 45(5):485–560, June 1992.

[6] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comp.*, 19:297–301, 1965.

[7] A. Croisier, D. Esteban, and C. Galand. Perfect channel splitting by use of interpolation/decimation/tree decomposition techniques. *Int. Conf. on Information Sciences and Systems*, pages 443–446, August 1976.

[8] I. Daubechies. Orthonormal bases of compactly supported wavelets. *Communications on Pure and Appl. Math.*, 41(7):909–996, October 1988.

[9] I. Daubechies. *Ten Lectures on Wavelets*. CBMS-NSF conference series in applied mathematics. SIAM Ed., 1992.

[10] P. Duhamel and H. Hollmann. 'split-radix' FFT-algorithm. *Electronic letters*, 20(1):14–16, 1984.

[11] FBI. WSQ gray-scale fingerprint image compression specification. Technical report, IAFIS-IC, 1993.

[12] G. B. Folland. *Real Analysis. Modern Techniques and Their Applications*. John Wiley and sons, 1984.

[13] M. W. Frazier. *An Introduction to Wavelets Through Linear Algebra*. Springer, 1999.

[14] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[15] R. C. Gonzalez, R. E. Woods, and S. L. Eddins. *Digital Image Processing Using MATLAB*. Gatesmark publishing, 2009.

[16] ISI/IEC. Information technology - coding of moving pictures and associated audio for digital storage media at up to about 1.5 mbit/s. Technical report, ISO/IEC, 1993.

[17] ISO/IEC. Jpeg2000 part 1 final draft international standard. iso/iec fdis 15444-1. Technical report, ISO/IEC, 2000.

[18] S.G Johnson and M. Frigo. A modified split-radix FFT with fewer arithmetic operations. *IEEE Transactions on Signal Processing*, 54, 2006.

[19] J.D. Johnston. A filter family designed for use in quadrature mirror filter banks. *Proc. Int. Conf. Acoust. Speech and Sig. Proc.*, pages 291–294, 1980.

[20] D. C. Lay. *Linear Algebra and Its Applications (4th Edition)*. Addison-Wesley, 2011.

[21] S. Mallat. *A Wavelet Tour of Signal Processing*. Tapir Academic Press, 1998.

[22] C. D. Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, 2000.

[23] Knut Mørken. *Numerical Algorithms and Digital Representation*. UIO, 2013.

[24] P. Noll. MPEG digital audio coding. *IEEE Signal processing magazine*, pages 59–81, September 1997.

[25] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1989.

[26] D. Pan. A tutorial on MPEG/audio compression. *IEEE Multimedia*, pages 60–74, Summer 1995.

[27] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reihnold, 1993.

[28] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing. Principles, Algorithms, and Applications. Fourth Edition*. Pearson, 2007.

[29] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56:1107–1108, June 1968.

[30] T. A. Ramstad, S. O. Aase, and J. H. Husøy. *Subband Compression of Images: Principles and Examples: Principles and Examples*, volume 6. Elsevier Science, 1995.

[31] C. E. Shannon. Communication in the presence of noise. *Proc. Institute of Radio Engineers*, 37(1):10–21, Jan. 1949.

[32] P. Stoica and R. Moses. *Spectral Analysis of Signals*. Prentice Hall, 2005.

[33] G. Strang and T. Nguyen. *Wavelets and Filter Banks*. Wellesley - Cambridge Press, 1996.

[34] W. Sweldens. The lifting scheme: a new philosophy in biorthogonal wavelet constructions. *Wavelet Applications in Signal and Image Processing III*, pages 68–79, 1995.

[35] W. Sweldens. The lifting scheme: a custom-design construction of biorthogonal wavelets. *Applied and computational harmonic analysis*, 3:186–200, 1996.

[36] D. S. Taubman and M. W. Marcellin. *JPEG2000. Image Compression. Fundamentals, Standards and Practice.* Kluwer Academic Publishers, 2002.

[37] M. Vetterli and J. Kovacevic. *Wavelets and Subband Coding*. Prentice Hall, 1995.

[38] M. Vetterli and H. J. Nussbaumer. Simple FFT and DCT algorithms with reduced number of operations. *Signal Processing*, 6:267–278, 1984.

[39] S. Winograd. On computing the discrete Fourier transform. *Math. Comp.*, 32:175–199, 1978.

[40] R. Yavne. An economical method for calculating the discrete Fourier transform. *Proc. AFIPS Fall Joint Computer Conf.*, 33:115–125, 1968.

Index