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

Øyvind Ryan

Feb 19, 2016

Preface

Chapter 1

Sound and Fourier series

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.

Solution. Setting p_{ref}=0.00002 Pa and p=100 000 Pa in the decibel expression we get

$$\begin{aligned}
20\log_{10}\left(\frac{p}{p_{\text{ref}}}\right) &= 20\log_{10}\left(\frac{100000}{0.00002}\right) = 20\log_{10}\left(\frac{10^5}{2\times 10^{-5}}\right) \\
&= 20\log_{10}\left(\frac{10^{10}}{2}\right) = 20\left(10-\log_{10}2\right) \approx 194\text{db}.
\end{aligned}$$

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?

Solution. $\sin(2\pi\nu_1 t)$ has period $1/\nu_1$, while $\sin(2\pi\nu_2 t)$ has period $1/\nu_2$. The period is not unique, however. The first one also has period n/ν_1, and the second also n/ν_2, for any n. The sum is periodic if there exist n_1, n_2 so that $n_1/\nu_1 = n_2/\nu_2$, i.e. so that there exists a common period between the two. This common period will also be a period of f. This amounts to that $\nu_1/\nu_2 = n_1/n_2$, i.e. that ν_1/ν_2 is a rational number.

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.

Solution. The function $f(t) = \frac{1}{\sqrt{t}} = t^{-1/2}$ can be used since it has the properties

$$\begin{aligned}
\int_0^T f(t)dt &= \lim_{x\to 0+} \int_x^T t^{-1/2}dt = \lim_{x\to 0+} \left[2t^{1/2}\right]_x^T \\
&= \lim_{x\to 0+} (2T^{1/2} - 2x^{1/2}) = 2T^{1/2} \\
\int_0^T f(t)^2 dt &= \lim_{x\to 0+} \int_x^T t^{-1}dt = \lim_{x\to 0+} \left[\ln t\right]_x^T \\
&= \ln T - \lim_{x\to 0+} \ln x = \infty.
\end{aligned}$$

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 $V_{N_1,T_1} \subset V_{N_2,T_2}$.

Solution. The space V_{N_1,T_1} is spanned by pure tones with frequencies $1/T_1, \ldots, N_1/T_1$, while V_{N_2,T_2} is spanned by pure tones with frequencies $1/T_2, \ldots, N_2/T_2$. We must have that the first set of frequencies is contained in the second. This is achieved if and only if $1/T_1 = k/T_2$ for some integer k, and also $N_1/T_1 \leq N_2/T_2$. In other words, T_2/T_1 must be an integer, and $T_2/T_1 \leq N_2/N_1$.

Exercise 1.5: antisymmetric functions are sine-series

Prove the second part of Theorem 1.20 in the compendium, 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 $[0,T]$.

Solution. For $f(t) = t$ we get that $a_0 = \frac{1}{T}\int_0^T t dt = \frac{T}{2}$. We also get

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

The Fourier series is thus

$$
\frac{T}{2} - \sum_{n\geq 1}\frac{T}{\pi n}\sin(2\pi nt/T).
$$

Note that this is almost a sine series, since it has a constant term, but no other cosine terms. If we had subtracted $T/2$ we would have obtained a function which is antisymmetric, and thus a pure sine series.

For $f(t) = t^2$ we get that $a_0 = \frac{1}{T}\int_0^T t^2 dt = \frac{T^2}{3}$. We also get

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

Here we see that we could use the expressions for the Fourier coefficients of $f(t) = t$ to save some work. The Fourier series is thus

$$
\frac{T^2}{3} + \sum_{n\geq 1}\left(\frac{T^2}{\pi^2 n^2}\cos(2\pi nt/T) - \frac{T^2}{\pi n}\sin(2\pi nt/T)\right).
$$

For $f(t) = t^3$ we get that $a_0 = \frac{1}{T}\int_0^T t^3 dt = \frac{T^3}{4}$. We also get

$$\begin{aligned}
a_n &= \frac{2}{T}\int_0^T t^3\cos(2\pi nt/T)dt \\
&= \frac{2}{T}\left(\left[\frac{T}{2\pi n}t^3\sin(2\pi nt/T)\right]_0^T - \frac{3T}{2\pi n}\int_0^T t^2\sin(2\pi nt/T)dt\right) \\
&= \left(-\frac{3T}{2\pi n}\right)\left(-\frac{T^2}{\pi n}\right) = \frac{3T^3}{2\pi^2 n^2} \\
b_n &= \frac{2}{T}\int_0^T t^3\sin(2\pi nt/T)dt \\
&= \frac{2}{T}\left(\left[-\frac{T}{2\pi n}t^3\cos(2\pi nt/T)\right]_0^T + \frac{3T}{2\pi n}\int_0^T t^2\cos(2\pi nt/T)dt\right) \\
&= -\frac{T^3}{\pi n} + \frac{3T}{2\pi n}\frac{T^2}{\pi^2 n^2} = -\frac{T^3}{\pi n} + \frac{3T^3}{2\pi^3 n^3}.
\end{aligned}$$

Also here we saved some work, by reusing the expressions for the Fourier coefficients of $f(t) = t^2$. The Fourier series is thus

$$\frac{T^3}{4} + \sum_{n\geq 1}\left(\frac{3T^3}{2\pi^2 n^2}\cos(2\pi nt/T) + \left(-\frac{T^3}{\pi n} + \frac{3T^3}{2\pi^3 n^3}\right)\sin(2\pi nt/T)\right).$$

We see that all three Fourier series converge slowly. This is connected to the fact that none of the functions are continuous at the borders of the periods.

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.

Solution. Let us define $a_{n,k}, b_{n,k}$ as the Fourier coefficients of t^k. When $k > 0$ and $n > 0$, integration by parts gives us the following difference equations:

$$
\begin{aligned}
a_{n,k} &= \frac{2}{T}\int_0^T t^k \cos(2\pi nt/T)dt \\
&= \frac{2}{T}\left(\left[\frac{T}{2\pi n}t^k \sin(2\pi nt/T)\right]_0^T - \frac{kT}{2\pi n}\int_0^T t^{k-1}\sin(2\pi nt/T)dt\right) \\
&= -\frac{kT}{2\pi n}b_{n,k-1} \\
b_{n,k} &= \frac{2}{T}\int_0^T t^k \sin(2\pi nt/T)dt \\
&= \frac{2}{T}\left(\left[-\frac{T}{2\pi n}t^k \cos(2\pi nt/T)\right]_0^T + \frac{kT}{2\pi n}\int_0^T t^{k-1}\cos(2\pi nt/T)dt\right) \\
&= -\frac{T^k}{\pi n} + \frac{kT}{2\pi n}a_{n,k-1}.
\end{aligned}
$$

When $n > 0$, these can be used to express $a_{n,k}, b_{n,k}$ in terms of $a_{n,0}, b_{n,0}$, for which we clearly have $a_{n,0} = b_{n,0} = 0$. For $n = 0$ we have that $a_{0,k} = \frac{T^k}{k+1}$ for all k. The following program computes $a_{n,k}, b_{n,k}$ recursively when $n > 0$.

```
def findfouriercoeffs(n, k, T):
    ank, bnk = 0, 0
    if k > 0:
        ankprev, bnkprev = findfouriercoeffs(n,k-1,T)
        ank = -k*T*bnkprev/(2*pi*n)
        bnk = -T**k/(pi*n) + k*T*ankprev/(2*pi*n)
    return ank, bnk
```

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 in the compendium.

Exercise 1.9: Orthonormality of Complex Fourier basis

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

Solution. For $n_1 \neq n_2$ we have that

$$
\begin{aligned}
\langle e^{2\pi in_1t/T}, e^{2\pi in_2t/T}\rangle &= \frac{1}{T}\int_0^T e^{2\pi in_1t/T}e^{-2\pi in_2t/T}dt = \frac{1}{T}\int_0^T e^{2\pi i(n_1-n_2)t/T}dt \\
&= \left[\frac{T}{2\pi i(n_1-n_2)}e^{2\pi i(n_1-n_2)t/T}\right]_0^T \\
&= \frac{T}{2\pi i(n_1-n_2)} - \frac{T}{2\pi i(n_1-n_2)} = 0.
\end{aligned}
$$

When $n_1 = n_2$ the integrand computes to 1, so that $\|e^{2\pi int/T}\| = 1$.

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

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

Solution. We have that

$$
\begin{aligned}
f(t) = \sin^2(2\pi t/T) &= \left(\frac{1}{2i}(e^{2\pi it/T} - e^{-2\pi it/T}\right)^2 \\
&= -\frac{1}{4}(e^{2\pi i2t/T} - 2 + e^{-2\pi i2t/T}) = -\frac{1}{4}e^{2\pi i2t/T} + \frac{1}{2} - \frac{1}{4}e^{-2\pi i2t/T}.
\end{aligned}
$$

This gives the Fourier series of the function (with $y_2 = y_{-2} = -1/4$, $y_0 = 1/2$). This could also have been shown by using the trigonometric identity $\sin^2 x = \frac{1}{2}(1 - \cos(2x))$ first, or by computing the integral $\frac{1}{T}\int_0^T f(t)e^{-2\pi int/T}dt$ (but this is rather cumbersome).

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 $1 \leq n \leq N$.

Solution. We have that

$$
\begin{aligned}
\cos^n(t) &= \left(\frac{1}{2}(e^{it} + e^{-it})\right)^n \\
\sin^n(t) &= \left(\frac{1}{2i}(e^{it} - e^{-it})\right)^n
\end{aligned}
$$

If we multiply out here, we get a sum of terms of the form e^{ikt}, where $-n \leq k \leq n$. As long as $n \leq N$ it is clear that this is in $V_{N,2\pi}$.

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

Solution. We have that

$$\begin{aligned}
\cos(t) &= \frac{1}{2}(e^{it}+e^{-it}) \\
\cos^2(t) &= \frac{1}{4}(e^{it}+e^{-it})^2 = \frac{1}{4}e^{2it}+\frac{1}{2}+\frac{1}{4}e^{-2it} \\
\cos^3(t) &= \frac{1}{8}(e^{it}+e^{-it})^3 = \frac{1}{8}e^{3it}+\frac{3}{8}e^{it}+\frac{3}{8}e^{-it}+\frac{1}{8}e^{-3it}.
\end{aligned}$$

Therefore, for the first function the nonzero Fourier coefficients are $y_{-1} = 1/2$, $y_1 = 1/2$, for the second function $y_{-2} = 1/4$, $y_0 = 1/2$, $y_2 = 1/4$, for the third function $y_{-3} = 1/8$, $y_{-1} = 3/8$, $y_1 = 3/8$, $y_3 = 1/8$.

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 $f_n(t) = \cos^n t$.

Solution. In order to find the Fourier coefficients of $\cos^n(t)$ we have to multiply out the expression $\frac{1}{2^n}(e^{it}+e^{-it})^n$. The coefficients we get after this can alos be obtained from Pascal's triangle.

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 in the compendium for the complex case. Use Theorem 1.26 in the compendium to verify your result.

Solution. We obtain that

$$\begin{aligned}
y_n &= \frac{1}{T}\int_0^{T/2} e^{-2\pi int/T}dt - \frac{1}{T}\int_{T/2}^{T} e^{-2\pi int/T}dt \\
&= -\frac{1}{T}\left[\frac{T}{2\pi in}e^{-2\pi int/T}\right]_0^{T/2} + \frac{1}{T}\left[\frac{T}{2\pi in}e^{-2\pi int/T}\right]_{T/2}^{T} \\
&= \frac{1}{2\pi in}\left(-e^{-\pi in}+1+1-e^{-\pi in}+\right) \\
&= \frac{1}{\pi in}\left(1-e^{-\pi in}\right) = \begin{cases} 0, & \text{if } n \text{ is even;} \\ 2/(\pi in), & \text{if } n \text{ is odd.} \end{cases}.
\end{aligned}$$

Instead using Theorem 1.26 in the compendium together with the coefficients $b_n = \frac{2(1-\cos(n\pi))}{n\pi}$ we computed in Example 1.17 in the compendium, we obtain

$$y_n = \frac{1}{2}(a_n - ib_n) = -\frac{1}{2}i \begin{cases} 0, & \text{if } n \text{ is even;} \\ 4/(n\pi), & \text{if } n \text{ is odd.} \end{cases} = \begin{cases} 0, & \text{if } n \text{ is even;} \\ 2/(\pi i n), & \text{if } n \text{ is odd.} \end{cases}$$

when $n > 0$. The case $n < 0$ follows similarly.

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 in the compendium to verify your calculations from Exercise 1.6.

Solution. For $f(t) = t$ we get

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

From Exercise 1.6 we had $b_n = -\frac{T}{\pi n}$, for which Theorem 1.26 in the compendium gives $y_n = \frac{T}{2\pi n}i$ for $n > 0$, which coincides with the expression we obtained. The case $n < 0$ follows similarly.
For $f(t) = t^2$ we get

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

From Exercise 1.6 we had $a_n = \frac{T^2}{\pi^2n^2}$ and $b_n = -\frac{T^2}{\pi n}$, for which Theorem 1.26 in the compendium gives $y_n = \frac{1}{2}\left(\frac{T^2}{\pi^2n^2} + i\frac{T^2}{\pi n}\right)$ for $n > 0$, which also is seen to coincide with what we obtained. The case $n < 0$ follows similarly.
For $f(t) = t^3$ we get

$$y_n = \frac{1}{T}\int_0^T t^3 e^{-2\pi int/T} dt = \frac{1}{T}\left(\left[-\frac{T}{2\pi in}t^3 e^{-2\pi int/T}\right]_0^T + 3\int_0^T \frac{T}{2\pi in}t^2 e^{-2\pi int/T} dt\right)$$
$$= -\frac{T^3}{2\pi in} + 3\frac{T}{2\pi in}(\frac{T^2}{2\pi^2 n^2} + \frac{T^2}{2\pi n}i) = 3\frac{T^3}{4\pi^2 n^2} + \left(\frac{T^3}{2\pi n} - 3\frac{T^3}{4\pi^3 n^3}\right)i =$$

From Exercise 1.6 we had $a_n = \frac{3T^3}{2\pi^2 n^2}$ and $b_n = -\frac{T^3}{\pi n} + \frac{3T^3}{2\pi^3 n^3}$ for which Theorem 1.26 in the compendium gives

$$y_n = \frac{1}{2}\left(\frac{3T^3}{2\pi^2 n^2} + i\left(\frac{T^3}{\pi n} - \frac{3T^3}{2\pi^3 n^3}\right)\right) = \frac{3T^3}{4\pi^2 n^2} + \left(\frac{T^3}{2\pi n} - \frac{3T^3}{4\pi^3 n^3}\right)i$$

for $n > 0$, which also is seen to coincide with what we obtained. The case $n < 0$ follows similarly.

Exercise 1.16: Complex Fourier coefficients for symmetric and antisymmetric functions

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

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

Solution. If f is symmetric about 0 we have that $b_n = 0$. Theorem 1.26 in the compendium then gives that $y_n = \frac{1}{2}a_n$, which is real. The same theorem gives that $y_{-n} = \frac{1}{2}a_n = y_n$.

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

Solution. If f is antisymmetric about 0 we have that $a_n = 0$. Theorem 1.26 in the compendium then gives that $y_n = -\frac{1}{2}b_n$, which is purely imaginary. The same theorem gives that $y_{-n} = \frac{1}{2}b_n = -y_n$.

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

Solution. When $y_n = y_{-n}$ we can write

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

This is clearly symmetric, but then also $\sum_{n=-N}^{N} y_n e^{2\pi int/T}$ is symmetric since it is a sum of symmetric functions.

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

Solution. When $y_n = -y_{-n}$ we can write

$$y_{-n}e^{2\pi i(-n)t/T} + y_n e^{2\pi int/T} = y_n(-e^{2\pi int/T} + e^{2\pi int/T}) = 2iy_n \sin(2\pi nt/T)$$

This is clearly antisymmetric, but then also $\sum_{n=-N}^{N} y_n e^{2\pi int/T}$ is antisymmetric since it is a sum of antisymmetric functions, and since $y_0 = 0$.

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 in the compendium to verify your result.

Solution. We obtain that

$$\begin{aligned}
y_n &= \frac{1}{T}\int_{-T/4}^{T/4} e^{-2\pi int/T}dt - \frac{1}{T}\int_{-T/2}^{-T/4} e^{-2\pi int/T}dt - \frac{1}{T}\int_{T/4}^{T/2} e^{-2\pi int/T}dt \\
&= -\left[\frac{1}{2\pi in}e^{-2\pi int/T}\right]_{-T/4}^{T/4} + \left[\frac{1}{2\pi in}e^{-2\pi int/T}\right]_{-T/2}^{-T/4} + \left[\frac{1}{2\pi in}e^{-2\pi int/T}\right]_{T/4}^{T/2} \\
&= \frac{1}{2\pi in}\left(-e^{-\pi in/2} + e^{\pi in/2} + e^{\pi in/2} - e^{\pi in} + e^{-\pi in} - e^{-\pi in/2}\right) \\
&= \frac{1}{\pi n}\left(2\sin(\pi n/2) - \sin(\pi n)\right) = \frac{2}{\pi n}\sin(\pi n/2).
\end{aligned}$$

The square wave defined in this exercise can be obtained by delaying our original square wave with $-T/4$. Using Property 3 in Theorem 1.28 in the compendium with $d = -T/4$ on the complex Fourier coefficients

$$y_n = \begin{cases} 0, & \text{if } n \text{ is even;} \\ 2/(\pi in), & \text{if } n \text{ is odd,} \end{cases}$$

which we obtained for the square wave in Exercise 1.13, we obtain the Fourier coefficients

$$e^{2\pi in(T/4)/T}\begin{cases}0, & \text{if } n \text{ is even;}\\ 2/(\pi in), & \text{if } n \text{ is odd.}\end{cases} = \begin{cases}0, & \text{if } n \text{ is even;}\\ \frac{2i\sin(\pi n/2)}{\pi in}, & \text{if } n \text{ is odd.}\end{cases}$$
$$= \begin{cases}0, & \text{if } n \text{ is even;}\\ \frac{2}{\pi n}\sin(\pi n/2), & \text{if } n \text{ is odd.}\end{cases}.$$

This verifies the result.

Exercise 1.18: Find function from its Fourier series

Find a function f which has the complex Fourier series

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

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

Solution. Since the real Fourier series of the square wave is

$$\sum_{n\geq 1, n \text{ odd}} \frac{4}{\pi n}\sin(2\pi nt/T),$$

Theorem 1.26 in the compendium gives us that the complex Fourier coefficients are $y_n = -\frac{1}{2}i\frac{4}{\pi n} = -\frac{2i}{\pi n}$, and $y_{-n} = \frac{1}{2}i\frac{4}{\pi n} = \frac{2i}{\pi n}$ for $n > 0$. This means that $y_n = -\frac{2i}{\pi n}$ for all n, so that the complex Fourier series of the square wave is

$$-\sum_{n \text{ odd}} \frac{2i}{\pi n} e^{2\pi int/T}.$$

Using Property 4 in Theorem 1.28 in the compendium we get that the $e^{-2\pi i4t/T}$ (i.e. set $d = -4$) times the square wave has its n'th Fourier coefficient equal to $-\frac{2i}{\pi(n+4)}$. Using linearity, this means that $2ie^{-2\pi i4t/T}$ times the square wave has its n'th Fourier coefficient equal to $\frac{4}{\pi(n+4)}$. We thus have that the function

$$f(t) = \begin{cases} 2ie^{-2\pi i4t/T} & ,0 \leq t < T/2\\ -2ie^{-2\pi i4t/T} & ,T/2 \leq t < T\end{cases}$$

has the desired Fourier series.

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.

Solution. The $2n$th complex Fourier coefficient of $\breve{f}$ is

$$\begin{aligned}
&\frac{1}{2T}\int_0^{2T} \breve{f}(t)e^{-2\pi i 2nt/(2T)}dt \\
&= \frac{1}{2T}\int_0^{T} f(t)e^{-2\pi int/T}dt + \frac{1}{2T}\int_T^{2T} f(2T-t)e^{-2\pi int/T}dt.
\end{aligned}$$

Substituting $u = 2T - t$ in the second integral we see that this is

$$\begin{aligned}
&= \frac{1}{2T}\int_0^{T} f(t)e^{-2\pi int/T}dt - \frac{1}{2T}\int_T^{0} f(u)e^{2\pi inu/T}du \\
&= \frac{1}{2T}\int_0^{T} f(t)e^{-2\pi int/T}dt + \frac{1}{2T}\int_0^{T} f(t)e^{2\pi int/T}dt \\
&= \frac{1}{2}y_n + \frac{1}{2}y_{-n}.
\end{aligned}$$

Therefore we have $a_{2n} = y_n - y_{-n}$.

Chapter 2

Digital sound and Discrete Fourier analysis

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 in the compendium, 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.

Solution. The code for playing the sound can look like this:

```
t1 = arange(0, 4/440.0, 1/float(fs))
t2 = arange(4/440.0, 12/440.0, 1/float(fs))
t3 = arange(12/440.0, 20/440.0, 1/float(fs))

f1 = 0*t1                                  # The first part of f
f2 = 2*((440*t2-4)/8)*sin(2*pi*440*t2)    # The second part of f
f3 = 2*sin(2*pi*440*t3)                    # The third part of f
x = hstack([f1, f2, f3])
x /= abs(x).max()
play(x, fs)
```

Note that the sound has duration less than 0.05s, so you should only hear a very short beep. You also need to scale the values to be within -1 and 1, since some of the listed values are outside this range.

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 in the compendium as closely as possible. Generate the samples of this sound, and listen to it.

Solution. The important thing to note here is that there are two oscillations present in Figure 1.1(b) in the compendium: One slow oscillation with a higher amplitude, and one faster oscillation, with a lower amplitude. We see that there are 10 periods of the smaller oscillation within one period of the larger oscillation, so that we should be able to reconstruct the figure by using frequencies where one is 10 times the other, such as 440Hz and 4400Hz. Also, we see from the figure that the amplitude of the larger oscillation is close to 1, and close to 0.3 for the smaller oscillation. A good choice therefore seems to be $a = 1, b = 0.3$. The code can look this:

```
t = arange(0,3,1/float(fs))
x = sin(2*pi*440*t) + 0.3*sin(2*pi*4400*t)
x /= abs(x).max()
play(x, fs)
```

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).

Solution. The code can look like this:

```
def play_pure_sound(f):
    """
    Play a pure sound with a given frequency over three seconds.
    """
    fs = 44100
    t = linspace(0, 3, 3*fs)
    x = sin(2*pi*f*t)
    play(x, fs)
```

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 in the compendium and the triangle wave of Example 1.11 in the compendium, 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 $T = \frac{1}{440}$.

Solution. The code can look like this:

```
def play_square(T):
    length=3
    fs = 44100
    samplesperperiod = fs*T
    oneperiod = hstack([ones((samplesperperiod/2),dtype=float),\
                        -ones((samplesperperiod/2),dtype=float)])
    x = tile(oneperiod,length/T)
    play(x, fs)
```

```
def play_triangle(T):
    length = 3
    fs = 44100
    samplesperperiod = fs*T
    oneperiod = hstack([linspace(-1, 1, samplesperperiod/2), \
                       linspace(1, -1, samplesperperiod/2)])
    x = tile(oneperiod, length/T)
    play(x, fs)
```

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 1.11 in the compendium and 1.18 in the compendium.

Solution. The code can look like this:

```
def play_square_fourier(T, N):
    length = 3
    fs = 44100
    t = linspace(0, length, fs*length)
    x = zeros(t.size)
    for n in range(1,N+1,2):
        x += sin(2*pi*n*t/T)/n
    x *= 4/pi
```

```
    x /= abs(x).max()
    play(x, fs)
```

```
def play_triangle_fourier(T, N):
    length = 3
    fs = 44100
    t = linspace(0, length, fs*length)
    x = zeros(t.size)
    for n in range(1,N+1,2):
        x -= cos(2*pi*n*t/T)/n**2
    x *= 8/(pi**2)
    x /= abs(x).max()
    play(x,fs)
```

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 in the compendium?

Solution. The code can look like this:

```
def play_with_different_fs( x, fs):
    play(x, fs)
    raw_input('PRESS ENTER TO CONTINUE.')
    play(x, 2*fs)
    raw_input('PRESS ENTER TO CONTINUE.')
    play(x, fs/2)
```

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 in the compendium?

Solution. The code can look like this:

```
def play_reverse(x, fs):
    """
    Play the sound backwards.
    """
    N = shape(x)[0]
    play(x[(N-1)::(-1)], fs)
```

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.

Solution. The code can look like this:

```
def play_with_noise(x, fs, c=0.1):
    """
    Play the sound with noise added. c represents the noise level,
    a number between 0 and 1.
    """
    z = x + c*(2*random.random(shape(x))-1)
    z /= abs(z).max()
    play(z, fs)
```

b) With your program, generate the two sounds played in Example 2.7 in the compendium, 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?

Exercise 2.9: Computing the DFT by hand

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

Solution. As in Example 2.16 in the compendium we get

$$F_4 \begin{pmatrix} 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$$

$$= \frac{1}{2} \begin{pmatrix} 2+3+4+5 \\ 2-3i-4+5i \\ 2-3+4-5 \\ 2+3i-4-5i \end{pmatrix} = \begin{pmatrix} 7 \\ -1+i \\ -1 \\ -1-i \end{pmatrix}.$$

Exercise 2.10: Exact form of low-order DFT matrix

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

Solution. For $N = 6$ the entries are on the form $\frac{1}{\sqrt{6}} e^{-2\pi ink/6} = \frac{1}{\sqrt{6}} e^{-\pi ink/3}$. This means that the entries in the Fourier matrix are the numbers $\frac{1}{\sqrt{6}} e^{-\pi i/3} = \frac{1}{\sqrt{6}}(1/2 - i\sqrt{3}/2)$, $\frac{1}{\sqrt{6}} e^{-2\pi i/3} = \frac{1}{\sqrt{6}}(-1/2 - i\sqrt{3}/2)$, and so on. The matrix is thus

$$F_6 = \frac{1}{\sqrt{6}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1/2 - i\sqrt{3}/2 & -1/2 - i\sqrt{3}/2 & -1 & -1/2 + i\sqrt{3}/2 & 1/2 + i\sqrt{2}/2 \\ 1 & -1/2 - i\sqrt{3}/2 & -1/2 + i\sqrt{3}/2 & 1 & -1/2 - i\sqrt{3}/2 & +1/2 - i\sqrt{3}/2 \\ 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -1/2 + i\sqrt{3}/2 & -1/2 - i\sqrt{3}/2 & 1 & -1/2 + i\sqrt{3}/2 & -1/2 - i\sqrt{3}/2 \\ 1 & 1/2 + i\sqrt{2}/2 & -1/2 + i\sqrt{3}/2 & -1 & -1/2 - i\sqrt{3}/2 & 1/2 - i\sqrt{3}/2 \end{pmatrix}$$

The cases $N = 8$ and $N = 12$ follow similarly, but are even more tedious. For $N = 8$ the entries are $\frac{1}{\sqrt{8}} e^{\pi ink/4}$, which can be expressed exactly since we can express exactly any sines and cosines of a multiple of $\pi/4$. For $N = 12$ we get the base angle $\pi/6$, for which we also have exact values for sines and cosines for all multiples.

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. $z_5 = x_0$, $z_6 = x_1, \ldots, z_{N-1} = x_{N-6}$, and $z_0 = x_{N-5}, \ldots, z_4 = x_{N-1}$. For a given n, if $|(F_N \boldsymbol{x})_n| = 2$, what is then $|(F_N \boldsymbol{z})_n|$? Justify the answer.

Solution. $\boldsymbol{z}$ is the vector $\boldsymbol{x}$ delayed with $d = 5$ samples, and then Property 3 of Theorem 2.18 in the compendium gives us that $(F_N \boldsymbol{z})_n = e^{-2\pi i 5k/N} (F_N \boldsymbol{x})_n$. In particular $|(F_N \boldsymbol{z})_n| = |(F_N \boldsymbol{x})_n| = 2$, since $|e^{-2\pi i 5k/N}| = 1$.

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 $(F_8(\boldsymbol{x}))_6$?

Solution. By Theorem 2.18 in the compendium we know that $(F_N(\boldsymbol{x}))_{N-n} = \overline{(F_N(\boldsymbol{x}))_n}$ when $\boldsymbol{x}$ is a real vector. If we set $N = 8$ and $n = 2$ we get that $(F_8(\boldsymbol{x}))_6 = \overline{(F_8(\boldsymbol{x}))_2} = \overline{2 - i} = 2 + i$.

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 $F_N\boldsymbol{x}$?

Solution. The idea is to express $\boldsymbol{x}$ as a linear combination of the Fourier basis vectors ϕ_n, and use that $F_N\phi_n = \boldsymbol{e}_n$. We have that

$$\begin{aligned}
\cos^2(2\pi k/N) &= \left(\frac{1}{2}\left(e^{2\pi ik/N} + e^{-2\pi ikn/N}\right)\right)^2 \\
&= \frac{1}{4}e^{2\pi i2k/N} + \frac{1}{2} + \frac{1}{4}e^{-2\pi i2k/N} = \frac{1}{4}e^{2\pi i2k/N} + \frac{1}{2} + \frac{1}{4}e^{2\pi i(N-2)k/N} \\
&= \sqrt{N}\left(\frac{1}{4}\phi_2 + \frac{1}{2}\phi_0 + \frac{1}{4}\phi_{N-2}\right).
\end{aligned}$$

We here used the periodicity of $e^{2\pi ikn/N}$, i.e. that $e^{-2\pi i2k/N} = e^{2\pi i(N-2)k/N}$. Since F_N is linear and $F_N(\phi_n) = \boldsymbol{e}_n$, we have that

$$F_N(\boldsymbol{x}) = \sqrt{N}\left(\frac{1}{4}\boldsymbol{e}_2 + \frac{1}{2}\boldsymbol{e}_0 + \frac{1}{4}\boldsymbol{e}_{N-2}\right) = \sqrt{N}\,(1/2, 0, 1/4, 0, \ldots, 0, 1/4, 0)\,.$$

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

Solution. We get

$$\begin{aligned}
y_n &= \sum_{k=0}^{N-1} c^k e^{-2\pi ink/N} = \sum_{k=0}^{N-1} (ce^{-2\pi in/N})^k \\
&= \frac{1 - (ce^{-2\pi in/N})^N}{1 - ce^{-2\pi in/N}} = \frac{1 - c^N}{1 - ce^{-2\pi in/N}}.
\end{aligned}$$

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 in the compendium 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.

Solution. The code can look like this:

```
def DFTImpl(x, forward=True):
    """
    Compute the DFT of the vector x using standard matrix
    multiplication. To avoid out of memory situations, we do not
    allocate the entire DFT matrix, only one row of it at a time.
    Note that this function differs from the FFT in that it includes
    the normalizing factor 1/sqrt(N). The DFT is computed along axis
    0. If there is another axis, the DFT is computed for each element
    in this as well.

    x: a vector
    forward: Whether or not this is forward (i.e. DFT)
    or reverse (i.e. IDFT)
    """
    y = zeros_like(x).astype(complex)
    N = len(x)
    sign = -(2*forward - 1)
    if ndim(x) == 1:
        for n in xrange(N):
            D = exp(sign*2*pi*n*1j*arange(float(N))/N)
            y[n] = dot(D, x)
    else:
        for n in range(N):
            D = exp(sign*2*pi*n*1j*arange(float(N))/N)
            for s2 in xrange(shape(x)[1]):
                y[n,s2] = dot(D,x[:, s2])
    if sign == 1:
        y /= float(N)
    return y
```

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.

Solution. We have that

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

If n is odd, we see that $y_n = 0$.

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.

Solution. The proof is the same as in a), except for a sign change.

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

Solution. Clearly the set of vectors which satisfies $x_{k+N/2} = \pm x_k$ is a vector space V of dimension $N/2$. The set of vectors where every second component is zero is also a vector space of dimension $N/2$, let us denote this by W. We have shown that $F_N(V) \subset W$, but since F_N is unitary, $F_N(V)$ also has dimension $N/2$, so that $F_N(V) = W$. This shows that when every second y_n is 0, we must have that $x_{k+N/2} = \pm x_k$, and the proof is done.

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 $0 \leq k < N/2$.

Solution. In the proofs above, compute the IDFT instead.

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 in the compendium to show that

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

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

Solution. We have that

$$
\begin{aligned}
(F_N(\boldsymbol{x}))_k &= (F_N(\boldsymbol{x}_1 + i\boldsymbol{x}_2))_k = (F_N(\boldsymbol{x}_1))_k + i(F_N(\boldsymbol{x}_2))_k \\
(F_N(\boldsymbol{x}))_{N-k} &= (F_N(\boldsymbol{x}_1))_{N-k} + i(F_N(\boldsymbol{x}_2))_{N-k} = \overline{(F_N(\boldsymbol{x}_1))_k} + i\overline{(F_N(\boldsymbol{x}_2))_k},
\end{aligned}
$$

where we have used Property 1 of Theorem 2.18 in the compendium. If we take the complex conjugate in the last equation, we are left with the two equations

$$
\begin{aligned}
(F_N(\boldsymbol{x}))_k &= (F_N(\boldsymbol{x}_1))_k + i(F_N(\boldsymbol{x}_2))_k \\
\overline{(F_N(\boldsymbol{x}))_{N-k}} &= (F_N(\boldsymbol{x}_1))_k - i(F_N(\boldsymbol{x}_2))_k.
\end{aligned}
$$

If we add these we get

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

which is the first equation. If we instead subtract the equations we get

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

which is the second equation

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?

Solution. First a sound file is read. We then restrict to the first 2^{12} sound samples, perform a DFT, zero out the frequencies which correspond to DFT-indices between 2^{10} and $2^{12} - 2^{10} - 1$, and perform an IDFT. Finally we scale the sound samples so that these lie between -1 and 1, which is the range we demand for the sound samples, and play the new sound.

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?

Solution. As we have seen, DFT index n corresponds to frequency $\nu = nf_s/N$. Above $N = 2^{17}$, so that we get the connection $\nu = nf_s/N = n \times 44100/2^{17}$. We zeroed the DFT indices above $n = 2^{15}$, so that frequencies above $\nu = 2^{15} \times 44100/2^{17} = 11025Hz$ are affected.

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 in the compendium,
- 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?

Solution. The code can look as follows.

```
import matplotlib.pyplot as plt
from numpy import *

f = lambda t:cos(t)**6
M = 5
T = 2*pi
N = 2*M + 1
t = linspace(0, T, 100)
x = f(linspace(0, T - T/float(N), N))
y = fft.fft(x, None, 0)/N
s = real(y[0])*ones(len(t))
for k in range(1,(N+1)/2):
    s += 2*real(y[k]*exp(2*pi*1j*k*t/float(T)))
plt.plot(t, s, 'r', t, f(t), 'g')
plt.legend(['Interpolant from V_{M,T}','f'])
plt.show()
```

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 in the compendium. 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.

Solution. The functions can be implemented as follows:

```
def FFTImpl(x, FFTKernel, forward = True):
    """
    Compute the FFT or IFFT of the vector x. Note that this function
    differs from the DFT in that the normalizing factor 1/sqrt(N) is
    not included. The FFT is computed along axis 0. If there is
    another axis, the FFT is computed for each element in this as
    well. This function calls a kernel for computing the FFT. The
    kernel assumes that the input has been bit-reversed, and contains
    only one axis. This function is where the actual bit reversal and
    the splitting of the axes take place.

    x: a vector
    FFTKernel: can be any of FFTKernelStandard, FFTKernelNonrec, and
    FFTKernelSplitradix. The kernel assumes that the input has been
    bit-reversed, and contains only one axis.
    forward: Whether the FFT or the IFFT is applied
    """
    if ndim(x) == 1:
        bitreverse(x)
        FFTKernel(x, forward)
    else:
        bitreversearr(x)
        for s2 in xrange(shape(x)[1]):
            FFTKernel(x[:, s2], forward)
    if not forward:
        x /= len(x)
```

```
def FFTKernelStandard(x, forward):
    """
    Compute the FFT of x, using a standard FFT algorithm.

    x: a bit-reversed version of the input. Should have only one axis
    forward: Whether the FFT or the IFFT is applied
    """
    N = len(x)
    sign = -1
    if not forward:
        sign = 1
    if N > 1:
        xe, xo = x[0:(N/2)], x[(N/2):]
        FFTKernelStandard(xe, forward)
        FFTKernelStandard(xo, forward)
        D = exp(sign*2*pi*1j*arange(float(N/2))/N)
        xo *= D
        x[:] = concatenate([xe + xo, xe - xo])
```

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 in the compendium), 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?

Solution. The two different curves you see should have a derivative approximately equal to one and two, respectively.

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.

Solution. There may be several reasons for this. One is that Python code runs slowly when compared to native code, which is much used in the built-in FFT. Also, the built-in `fft` has been subject to much more optimization than we have covered here.

Solution. The code can look as follows.

```
x0, fs = audioread('sounds/castanets.wav')

kvals = arange(3,16)
slowtime = zeros(len(kvals))
fasttime = zeros(len(kvals))
fastesttime = zeros(len(kvals))
N = 2**kvals
for k in kvals:
    x = x0[0:2**k].astype(complex)

    start = time()
    DFTImpl(x)
    slowtime[k - kvals[0]] = time() - start

    start = time()
    FFTImpl(x, FFTKernelStandard)
    fasttime[k - kvals[0]] = time() - start

    start = time()
    fft.fft(x, None, 0)
    fastesttime[k - kvals[0]] = time() - start

# a.
plt.plot(kvals, slowtime, 'ro-', \
     kvals, fasttime, 'go-', \
     kvals, fastesttime, 'bo-')
plt.grid('on')
plt.title('time usage of the DFT methods')
plt.legend(['DFT', 'Standard FFT', 'Built-in FFT'])
plt.xlabel('log_2 N')
plt.ylabel('time used [s]')
plt.show()

plt.figure()

# b.
plt.loglog(N, slowtime, 'ro-', N, fasttime, 'go-', N, fastesttime, 'bo-')
plt.axis('equal')
plt.legend(['DFT', 'Standard FFT', 'Built-in FFT'])
plt.show()
```

Exercise 2.24: Combine two FFT's

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

Solution. We get

$$\mathrm{DFT}_4\boldsymbol{x}_1 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 5 \\ 7 \end{pmatrix} = \begin{pmatrix} 16 \\ -4+4i \\ -4 \\ -4-4i \end{pmatrix}$$

$$\mathrm{DFT}_4\boldsymbol{x}_2 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 2 \\ 4 \\ 6 \\ 8 \end{pmatrix} = \begin{pmatrix} 20 \\ -4+4i \\ -4 \\ -4-4i \end{pmatrix}$$

In the FFT-algorithm we split the computation of $\mathrm{DFT}_4(\boldsymbol{x})$ into the computation of $\mathrm{DFT}_2(\boldsymbol{x}^{(e)})$ and $\mathrm{DFT}_2(\boldsymbol{x}^{(o)})$, where $x^{(e)}$ and $x^{(o)}$ are vectors of length 4 with even-indexed and odd-indexed components, respectively. In this case we have $\boldsymbol{x}^{(e)} = (1, 3, 5, 7)$ and $\boldsymbol{x}^{(o)} = (2, 4, 6, 8)$. In other words, the FFT-algorithm uses the FFT-computations we first made, so that we can save computation. The benefit of using the FFT-algorithm is that we save computations, so that we end up with $O(5N \log_2 N)$ real arithmetic operations.

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.

Solution. We have that $\boldsymbol{x}^{(p)}$ is a constant vector of length N_2 for $0 \leq p < N_1$. But then the DFT of all the $\boldsymbol{x}^{(p)}$ has zero outside entry zero. Multiplying with $e^{-2\pi ikn/N}$ does not affect this. The last $N_2 - 1$ rows are thus zero before the final DFT is applied, so that these rows are zero also after this final DFT. After assembling the polyphase components again we have that yrN_2 are the only nonzero DFT-coefficients.

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?

Solution. When we compute $e^{-2\pi in/N}$, we do some multiplications/divisions in the exponent. These are not counted because they do not depend on $\boldsymbol{x}$, and may therefore be precomputed.

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 $e^{2\pi in/N}(y_n - \overline{y_{N/2-n}})$. 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.

Solution. The algorithm for the non-recursive FFT can look as follows

```
def FFTKernelNonrec(x, forward):
    """
    Compute the FFT of x, using a non-recursive FFT algorithm.

    x: a bit-reversed version of the input. Should have only one axis
    forward: Whether the FFT or the IFFT is applied
    """
    N = len(x)
    sign = -1
    if not forward:
        sign = 1
    D = exp(sign*2*pi*1j*arange(float(N/2))/N)
    nextN = 1
    while nextN < N:
        k = 0
        while k < N:
            xe, xo = x[k:(k + nextN)], x[(k + nextN):(k + 2*nextN)]
            xo *= D[0::(N/(2*nextN))]
            x[k:(k+2*nextN)] = concatenate([xe + xo, xe - xo])
            k += 2*nextN
        nextN *= 2
```

If you add the non-recursive algorithm to the code from Exercise 2.23, you will see that the non-recursive algorithm performs much better. There may be several reasons for this. First of all, there are no recursive function calls. Secondly, the values in the matrices $D_{N/2}$ are constructed once and for all with the non-recursive algorithm. Code which compares execution times for the original FFT algorithm, our non-recursive implementation, and the split-radix algorithm of the next exercise, can look as follows:

```
x0, fs = audioread('sounds/castanets.wav')

kvals = arange(3,16)
slowtime = zeros(len(kvals))
fasttime = zeros(len(kvals))
fastesttime = zeros(len(kvals))
N = 2**kvals
for k in kvals:
    x = x0[0:2**k].astype(complex)

    start = time()
    FFTImpl(x, FFTKernelStandard)
    slowtime[k - kvals[0]] = time() - start
```

```
        start = time()
        FFTImpl(x, FFTKernelNonrec)
        fasttime[k - kvals[0]] = time() - start

        start = time()
        FFTImpl(x, FFTKernelSplitradix)
        fastesttime[k - kvals[0]] = time() - start

    plt.plot(kvals, slowtime, 'ro-', \
         kvals, fasttime, 'bo-', \
         kvals, fastesttime, 'go-')
    plt.grid('on')
    plt.title('time usage of the DFT methods')
    plt.legend(['Standard FFT algorithm', \
           'Non-recursive FFT', \
           'Split radix FFT'])
    plt.xlabel('log_2 N')
    plt.ylabel('time used [s]')
    plt.show()
```

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

$$\mathrm{DFT}_N \boldsymbol{x} = \begin{pmatrix} \mathrm{DFT}_{N/2} & D_{N/2}\begin{pmatrix} \mathrm{DFT}_{N/4} & D_{N/4}\mathrm{DFT}_{N/4} \\ \mathrm{DFT}_{N/4} & -D_{N/4}\mathrm{DFT}_{N/4} \end{pmatrix} \\ \mathrm{DFT}_{N/2} & -D_{N/2}\begin{pmatrix} \mathrm{DFT}_{N/4} & D_{N/4}\mathrm{DFT}_{N/4} \\ \mathrm{DFT}_{N/4} & -D_{N/4}\mathrm{DFT}_{N/4} \end{pmatrix} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(e)} \\ \boldsymbol{x}^{(oe)} \\ \boldsymbol{x}^{(oo)} \end{pmatrix}. \tag{2.1}$$

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 (2.1):

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

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 $O(4N\log_2 N)$, 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.

Solution. If you add the split-radix FFT algorithm also to the code from Exercise 2.23, you will see that it performs better than the FFT algorithm,

but worse than the non-recursive algorithm. That it performs better than the FFT algorithm is as expected, since it has a reduced number of arithmetic operations, and also a smaller number of recursive calls. It is not surprising that the non-recursive function performs better, since only that function omits recursive calls, and computes the values in the diagonal matrices once and for all.

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 in the compendium, 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.

Solution. The code for the split-radix algorithm can look as follows

```
def FFTKernelSplitradix(x, forward):
    """
    Compute the FFT of x, using the split-radix FFT algorithm.

    x: a bit-reversed version of the input. Should have only one axis
    forward: Whether the FFT or the IFFT is applied
    """
    N = len(x)
    sign = -1
    if not forward:
        sign = 1
    if N == 2:
        x[:] = [x[0] + x[1], x[0] - x[1]]
    elif N > 2:
        xe, xo1, xo2 = x[0:(N/2)], x[(N/2):(3*N/4)], x[(3*N/4):N]
        FFTKernelSplitradix(xe, forward)
        FFTKernelSplitradix(xo1, forward)
        FFTKernelSplitradix(xo2, forward)
        G = exp(sign*2*pi*1j*arange(float(N/4))/N)
        H = G*exp(sign*2*pi*1j*arange(float(N/4))/(N/2))
        xo1 *= G
        xo2 *= H
        xo = concatenate( [xo1 + xo2, -sign*1j*(xo2 - xo1)] )
        x[:] = concatenate([xe + xo, xe - xo])
```

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.

Solution. Note that, if the bit representation of i ends with $0\underbrace{1\ldots1}_{n}$, then $i+1$ has a bit representation which ends with $1\underbrace{0\ldots0}_{n}$, with the remaining first bits unaltered. Clearly the bit-reverse of i then starts with $\underbrace{1\ldots1}_{n}0$ and the bit-reverse of $i+1$ starts with $1\underbrace{0\ldots0}_{n}$. We see that the bit reverse of $i+1$ can be obtained from the bit-reverse of i by replacing the first consecutive set of ones by zeros, and the following zero by one. This is performed by the line above where j is decreased by m: Decreasing j by $N/2$ when $j \geq N/2$ changes the first bit from 1 to 0, and similarly for the next n bits. The line where j is increased with m changes bit number $n+1$ from 0 to 1.

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.

Solution. Clearly $N - i - 1$ has a bit representation obtained by changing every bit in i. That $N - j - 1$ is the bit-reverse of $N - i - 1$ follows immediately from this. If i is even, the least significant bit is 0. After bit-reversal, this becomes the most significant bit, and the most significant bit of j is 0 which is the case if and only if $j < N/2$.

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.

Solution. If $i < N/2$ is odd, then the least significant bit is 1. This means that the most significant bit of j is 1, so that $j \geq N/2$, so that $j > i$.

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).

Chapter 3

Operations on digital sound: digital filters

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 $N = 8$.

Solution. Here we have that $t_{-1} = 1/4$, $t_0 = 1/4$, $t_1 = 1/4$, and $t_2 = 1/4$. We now get that $s_0 = t_0 = 1/4$, $s_1 = t_1 = 1/4$, and $s_2 = t_2 = 1/4$ (first formula), and $s_{N-1} = s_7 = t_{-1} = 1/4$ (second formula). This means that the matrix of S is

$$S = \frac{1}{4}\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

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.

Solution. The code can look as follows.

```
from sound import *
from time import *
from numpy import *
import matplotlib.pyplot as plt

x, fs = audioread('sounds/castanets.wav')
x = x[:,0]
N= len(x)

kmax=100
vals1 = zeros(kmax/10)
vals2 = zeros(kmax/10)
vals3 = zeros(kmax/10)
ind = 0
for k in range(10, kmax+1,10):
    t = random.random(k)
    start = time()
    convolve(t, x)
    vals1[ind] = time() - start

    z = zeros(N)
    start = time()
    for s in range(k):
        z[(k-1):N] += t[s]*x[(k-1-s):(N-s)]
    vals2[ind] = time() - start

    z = zeros(N)
    start = time()
    for n in range(k-1, N):
        for s in range(k):
            z[n] += t[s]*x[n-s]
    vals3[ind] = time() - start
    ind += 1
plt.plot( range(10, kmax+1,10),log(vals1), 'r-', range(10, kmax+1,10),  log(vals2), 'g-', range(10,
plt.legend(['conv','simple for','double for'])
plt.show()
```

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 in the compendium 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 in the compendium to achieve this.

Exercise 3.6: Implementing filtering with convolution

Implement a function `filterS` which uses Proposition 3.8 in the compendium 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.

Solution. The code can look like this:

```
def filterS(t, x):
    L = (len(t) - 1)/2
    N = len(x)
    y = concatenate([ x[(N - L):], x, x[:L]])
    y = convolve(t, y)
    y = y[(2*L):(len(y)-2*L)]
```

Exercise 3.7: Time reversal is not a filter

In Example 2.6 in the compendium 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.

Solution. The matrix for time reversal is the matrix

$$\begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

This is not a circulant Toeplitz matrix, since all diagonals assume the values 0 and 1, so that they are not constant on each diagonal. Time reversal is thus not a digital filter.

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

Solution. Let S denote time reversal. Clearly $S\boldsymbol{e}_1 = \boldsymbol{e}_{N-2}$. If S was time-invariant we would have that $S\boldsymbol{e}_0 = \boldsymbol{e}_{N-3}$, where we have delayed the input and output. But this clearly is not the case, since by definition $S\boldsymbol{e}_0 = \boldsymbol{e}_{N-1}$.

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.

Solution. The eigenvalues of S are $1, 5, 9$, and are found by computing a DFT of the first column (and multiplying by $\sqrt{N} = 2$). The eigenvectors are the Fourier basis vectors. 1 has multiplicity 2.

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.

Solution. The computer uses some numeric algorithm to find the eigenvectors. However, eigenvectors may not be unique, so you have no control on which eigenvectors it actually selects. In particular, here the eigenspace for $\lambda = 1$ has dimension 2, so that any linear combination of the two eigenvectors from this eigenspace also is an eigenvector. Here it seems that a linear combination is chosen which is different from a Fourier basis vector.

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?

Solution. If we write $S_1 = F_N^H D_1 F_N$ and $S_2 = F_N^H D_2 F_N$ we get

$$S_1 + S_2 = F_N^H (D_1 + D_2) F_N \quad S_1 S_2 = F_N^H D_1 F_N F_N^H D_2 F_N = F_N^H D_1 D_2 F_N$$

This means that the eigenvalues of $S_1 + S_2$ are the sum of the eigenvalues of S_1 and S_2. The actual eigenvalues which are added are dictated by the index of the frequency response, i.e. $\lambda_{S_1+S_2,n} = \lambda_{S_1,n} + \lambda_{S_2,n}$.

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

Solution. As above we have that $S_1 S_2 = F_N^H D_1 F_N F_N^H D_2 F_N = F_N^H D_1 D_2 F_N$, and the same reasoning gives that the eigenvalues of $S_1 S_2$ are the product of the eigenvalues of S_1 and S_2. The actual eigenvalues which are multiplied are dictated by the index of the frequency response, i.e. $\lambda_{S_1 S_2,n} = \lambda_{S_1,n} \lambda_{S_2,n}$.

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)?

Solution. In general there is no reason to believe that there is a formula for the eigenvalues for the sum or product of two matrices, based on eigenvalues of the individual matrices. However, the same type of argument as for filters can be used in all cases where the eigenvectors are equal.

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?

Solution. The matrix for the operation which keeps every second component is

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix},$$

where 1 and 0 are repeated in alternating order along the main diagonal. Since the matrix is not constant on the main diagonal, it is not a circulant Toeplitz matrix, and hence not a filter.

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) $\lambda_S(\omega)$.

Solution. The frequency response is

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

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?

Solution. The filter coefficients are $t_0 = t_3 = 1/3$, $t_1 = t_2 = 1$. We have that

$$\begin{aligned}
\lambda_S(\omega) &= \sum_k t_k e^{-ik\omega} = \frac{1}{3}(1 + 3e^{-i\omega} + 3e^{-2i\omega} + e^{-3i\omega}) \\
&= \frac{2}{3}e^{-3i\omega/2}\frac{1}{2}(e^{3i\omega/2} + 3e^{i\omega/2} + 3e^{-i\omega/2} + e^{-3i\omega/2}) \\
&= \frac{2}{3}e^{-3i\omega/2}(\cos(3\omega/2) + 3\cos(\omega/2)).
\end{aligned}$$

From this expression it is easy to plot the frequency response, but since this is complex, we have to plot the magnitude, i.e. $|\lambda_S(\omega)| = \frac{2}{3}|\cos(3\omega/2)+3\cos(\omega/2)|$. We also see that $\lambda_S(0) = \frac{2}{3}$, and that $\lambda_S(\pi) = 0$, so that the filter is a low-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 $e^{2\pi i 2k/N}$?

Solution. If we use the connection between the vector frequency response and the continuous frequency response we get

$$\lambda_{S,2} = \lambda_S(2\pi 2/N) = \frac{2}{3}e^{-6\pi i/N}(\cos(6\pi/N) + 3\cos(2\pi/N)).$$

Alternatively you can here compute that the first column in the circulant Toeplitz matrix for S is given by $s_0 = t_1$, $s_2 = t_2$, $s_3 = t_3$, and $s_4 = t_4$, and insert this in

the definition of the vector frequency response, $\lambda_{S,2} = \sum_{k=0}^{N-1} s_k e^{-2\pi i 2k/N}$. We know that $e^{2\pi i 2k/N}$ is an eigenvector for S since S is a filter, and that $\lambda_{S,2}$ is the corresponding eigenvalue. We therefore get that

$$S\boldsymbol{x} = \lambda_{S,2}\boldsymbol{x} = \frac{2}{3}e^{-6\pi i/N}(\cos(6\pi/N) + 3\cos(2\pi/N))\boldsymbol{x}.$$

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 $N = 8$.

Solution. Since clearly $t_{-2} = t_2 = 1/16$, $t_{-1} = t_1 = 1/4$, and $t_0 = 6/16$, the first column $\boldsymbol{s}$ in the circulant Toeplitz matrix is given by $s_0 = t_0 = 6/16$, $s_1 = t_1 = 4/16$, $s_2 = t_2 = 1/16$, $s_{N-2} = t_{-2} = 1/16$, $s_{N-1} = t_{-1} = 4/16$. An 8×8 circulant Toeplitz matrix which corresponds to applying S_1 to a periodic signal of length $N = 8$ is therefore

$$\frac{1}{16}\begin{pmatrix} 6 & 4 & 1 & 0 & 0 & 0 & 1 & 4 \\ 4 & 6 & 4 & 1 & 0 & 0 & 0 & 1 \\ 1 & 4 & 6 & 4 & 1 & 0 & 0 & 0 \\ 0 & 1 & 4 & 6 & 4 & 1 & 0 & 0 \\ 0 & 0 & 1 & 4 & 6 & 4 & 1 & 0 \\ 0 & 0 & 0 & 1 & 4 & 6 & 4 & 1 \\ 1 & 0 & 0 & 0 & 1 & 4 & 6 & 4 \\ 4 & 1 & 0 & 0 & 0 & 1 & 4 & 6 \end{pmatrix}.$$

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

Solution. The frequency response is

$$\lambda_{S_1}(\omega) = \frac{1}{16}(e^{2i\omega}+4e^{i\omega}+6+4e^{-i\omega}+e^{-2i\omega}) = \left(\frac{1}{2}(e^{i\omega/2} + e^{-i\omega/2})\right)^4 = \cos^4(\omega/2),$$

where we recognized $(1, 4, 6, 4, 1)$ as a row in Pascal's triangle, so that we could write the expression as a power. From this expression it is easy to plot the frequency response, and it is clear that the filter is a low-pass filter, since $\lambda_{S_1}(0) = 1$, $\lambda_{S_1}(\pi) = 0$.

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 $S_1 S_2$.

Solution. We have that

$$\lambda_{S_2}(\omega) = (e^{i\omega} + 2 + e^{-i\omega})/4 = \left(\frac{1}{2}(e^{i\omega/2} + e^{-i\omega/2})\right)^2 = \cos^2(\omega/2).$$

We then get that

$$\begin{aligned}\lambda_{S_1S_2}(\omega) &= \lambda_{S_1}(\omega)\lambda_{S_2}(\omega) = \cos^4(\omega/2)\cos^2(\omega/2) = \cos^6(\omega/2) \\ &= \left(\frac{1}{2}(e^{i\omega/2} + e^{-i\omega/2})\right)^6 \\ &= \frac{1}{64}(e^{3i\omega} + 6e^{2i\omega} + 15e^{i\omega} + 20 + 15e^{-i\omega} + 6e^{-2i\omega} + e^{-3i\omega}),\end{aligned}$$

where we have used that, since we have a sixth power, the values can be obtained from fra a row in Pascal's triangle also here. It is now clear that

$$S_1S_2 = \frac{1}{64}\{1, 6, 15, \underline{20}, 15, 6, 1\}.$$

You could also argue here by taking the convolution of $\frac{1}{16}(1, 4, 6, 4, 1)$ with $\frac{1}{4}(1, 2, 1)$.

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?

Solution. We have that $\lambda_S(\omega) = \frac{1}{2}(1 + \cos\omega)$. This clearly has the maximum point $(0, 1)$, and the minimum point $(\pi, 0)$.

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?

Solution. We have that $|\lambda_T(\omega)| = \frac{1}{2}(1-\cos\omega)$. This clearly has the maximum point $(\pi, 1)$, and the minimum point $(0, 0)$. The connection between the frequency responses is that $\lambda_T(\omega) = \lambda_S(\omega+\pi)$.

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.

Solution. Here we have that $s_0 = t_0 = 3$, $s_1 = t_1 = 4$, $s_2 = t_2 = 5$, and $s_3 = t_3 = 6$ (first formula), and $s_{N-2} = t_{-2} = 1$, $s_{N-1} = t_{-1} = 2$ (second formula). This means that the matrix of S is

$$S = \begin{pmatrix} 3 & 2 & 1 & 0 & 0 & 6 & 5 & 4 \\ 4 & 3 & 2 & 1 & 0 & 0 & 6 & 5 \\ 5 & 4 & 3 & 2 & 1 & 0 & 0 & 6 \\ 6 & 5 & 4 & 3 & 2 & 1 & 0 & 0 \\ 0 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ 0 & 0 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 0 & 0 & 6 & 5 & 4 & 3 & 2 \\ 2 & 1 & 0 & 0 & 6 & 5 & 4 & 3 \end{pmatrix}$$

The frequency response is

$$\lambda_S(\omega) = e^{2i\omega} + 2e^{i\omega} + 3 + 4e^{-i\omega} + 5e^{-2i\omega} + 6e^{-3i\omega}.$$

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 $\lambda_S(\omega)$.

Solution. The filter coefficients are $t_0 = s_0 = 1/5$, $t_1 = s_1 = 1/5$ (first formula), and $t_{-1} = s_{N-1} = 1/5$, $t_{-2} = s_{N-2} = 1/5$, $t_{-3} = s_{N-3} = 1/5$ (second formula). All other t_k are zero. This means that the filter can be written as $\frac{1}{5}\{1, 1, 1, \underline{1}, 1\}$, using our compact notation.

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?

Solution. The frequency response is

$$\sum_{s=0}^{k} c^s e^{-is\omega} = \frac{1 - c^{k+1}e^{-i(k+1)\omega}}{1 - ce^{-i\omega}}.$$

It is straightforward to compute the limit as $\omega \to 0$ as $c^k(k+1)$. This means that as we increase k or c, this limit also increases. The value of k also dictates oscillations in the frequency response, since the numerator oscillates fastest. When $c = 1$, k dictates how often the frequency response hits 0.

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.

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 in the compendium, 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 in the compendium. Recall that you have to ensure that the sound samples lie in $[-1, 1]$.

Solution. The code can look like this:

```
def play_with_echo(x, fs, c, d):
    """
    Play the sound with an echo added.

    x: the vector of sound samples
    fs: the sample rate
    c:the strength of the echo
    d: the delay of the echo in samples.
    """
    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)]
    z /= abs(z).max()
    play(z, fs)
```

b) Generate the sound from Example 3.32 in the compendium, 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?

Solution. The sum of two digital filters is again a digital filter, and the first column in the sum can be obtained by summing the first columns in the two matrices. This means that the filter coefficients in $\frac{1}{2}(S_1 + S_2)$ can be obtained by summing the filter coefficients of S_1 and S_2, and we obtain

$$\frac{1}{2}\left(\{\underline{1}, 0, \ldots, 0, c\} + \{\underline{1}, 0, \ldots, 0, -c\}\right) = \{\underline{1}\}.$$

This means that $\frac{1}{2}(S_1 + S_2) = I$, since I is the unique filter with $\boldsymbol{e}_0$ as first column. The interpretation in terms of echos is that the echo from S_2 cancels that from S_1.

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 3.32 in the compendium and 3.41 in the compendium.

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.

Solution. The code can look like this. We have used the `convolve` function:

```
def play_with_reduced_bass( x, fs, k):
    t = [1.]
    for kval in range(k):
        t = convolve(t, [1/2., -1/2.])
    N,nchannels=shape(x)
    z = convolve(t, x[:, 0])
    play(z,fs)
```

```
def play_with_reduced_treble( x, fs, k):
    t = [1.]
    for kval in range(k):
        t = convolve(t, [1/2., 1/2.])
    N,nchannels=shape(x)
    z = convolve(t, x[:, 0])
    play(z,fs)
```

b) Generate the sounds you heard in examples 3.32 in the compendium and 3.41 in the compendium, 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 in the compendium. 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 in the compendium 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 in the compendium, 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.

Solution. We have that

$$\begin{aligned}
\lambda_{S_2}(\omega) &= \sum_k \cos(2\pi kn/N)t_k e^{-ik\omega} = \frac{1}{2}\sum_k (e^{2\pi ikn/N} + e^{-2\pi ikn/N})t_k e^{-ik\omega} \\
&= \frac{1}{2}\left(\sum_k t_k e^{-ik(\omega-2\pi n/N)} + \sum_k t_k e^{-ik(\omega+2\pi n/N)}\right) \\
&= \frac{1}{2}(\lambda_{S_1}(\omega - 2\pi n/N) + \lambda_{S_1}(\omega + 2\pi n/N)).
\end{aligned}$$

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?

Solution. In the code a filter is run on the sound samples from the file `castanets.wav`. Finally the new sound is played. In the first two lines after the `for`-loop, the first and the last sound samples in the filtered sound are computed, under the assumption that the sound has been extended to a periodic sound

with period `N`. After this, the sound is normalized so that the sound samples lie in the range between -1 and 1. In this case the filter is a lowpass-filter (as we show in b.), so that we can expect that that the treble in the sound is reduced.

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?

Solution. Compact filter notation for the filter which is run is $\{2, \underline{4}, 2\}$. A 5×5 circulant Toeplitz matrix becomes

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

The frequency response is $\lambda_S(\omega) = 2e^{i\omega} + 4 + 2e^{-i\omega} = 4 + 4\cos\omega$. It is clear that this gives a lowpass 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?

Solution. The frequency response for the new filter is

$$-2e^{i\omega} + 4 - 2e^{-i\omega} = 4 - 4\cos\omega = 4 + 4\cos(\omega + \pi) = \lambda_S(\omega + \pi),$$

where S is the filter from the first part of the exercise. The new filter therefore becomes a highpass filter, since to add π to ω corresponds to swapping the frequencies 0 and π. We could also here refered to Observation 3.40 in the compendium, where we stated that adding an alternating sign in the filter coefficients turns a lowpass filter into a highpass filter and vice versa.

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.

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.

Chapter 4

Symmetric filters and the DCT

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 $N = 4$.

Solution. First we obtain the matrix S as

$$\left(\begin{array}{cccc|cccc} \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} \end{array}\right)$$

where we have drawn the boundaries between the blocks S_1, S_2, S_3, S_4. From this we see that

$$S_1 = \begin{pmatrix} \frac{1}{2} & \frac{1}{4} & 0 & 0 \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ 0 & 0 & \frac{1}{4} & \frac{1}{2} \end{pmatrix} \quad S_2 = \begin{pmatrix} 0 & 0 & 0 & \frac{1}{4} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 & 0 \end{pmatrix} \quad (S_2)^f = \begin{pmatrix} \frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{4} \end{pmatrix}.$$

From this we get

$$S_r = S_1 + (S_2)^f = \begin{pmatrix} \frac{3}{4} & \frac{1}{4} & 0 & 0 \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ 0 & 0 & \frac{1}{4} & \frac{3}{4} \end{pmatrix}.$$

Exercise 4.3: Writing down lower order DCTs

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

Solution. We first see that $d_{0,3} = \sqrt{\frac{1}{3}}$ and $d_{k,3} = \sqrt{\frac{2}{3}}$ for $k = 1, 2$. We also have that

$$\cos\left(2\pi\frac{n}{2N}\left(k+\frac{1}{2}\right)\right) = \cos\left(\pi\frac{n}{3}\left(k+\frac{1}{2}\right)\right),$$

so that the DCT matrix can be written as

$$
\begin{aligned}
\mathrm{DCT}_3 &= \begin{pmatrix} \sqrt{\frac{1}{3}} & \sqrt{\frac{1}{3}} & \sqrt{\frac{1}{3}} \\ \sqrt{\frac{2}{3}}\cos\left(\frac{\pi}{3}\frac{1}{2}\right) & \sqrt{\frac{2}{3}}\cos\left(\frac{\pi}{3}\frac{3}{2}\right) & \sqrt{\frac{2}{3}}\cos\left(\frac{\pi}{3}\frac{5}{2}\right) \\ \sqrt{\frac{2}{3}}\cos\left(\frac{2\pi}{3}\frac{1}{2}\right) & \sqrt{\frac{2}{3}}\cos\left(\frac{2\pi}{3}\frac{3}{2}\right) & \sqrt{\frac{2}{3}}\cos\left(\frac{2\pi}{3}\frac{5}{2}\right) \end{pmatrix} \\
&= \begin{pmatrix} \sqrt{\frac{1}{3}} & \sqrt{\frac{1}{3}} & \sqrt{\frac{1}{3}} \\ \sqrt{\frac{2}{3}}\cos(\pi/6) & \sqrt{\frac{2}{3}}\cos(\pi/2) & \sqrt{\frac{2}{3}}\cos(5\pi/6) \\ \sqrt{\frac{2}{3}}\cos(\pi/3) & \sqrt{\frac{2}{3}}\cos(\pi) & \sqrt{\frac{2}{3}}\cos(5\pi/3) \end{pmatrix} \\
&= \begin{pmatrix} \sqrt{\frac{1}{3}} & \sqrt{\frac{1}{3}} & \sqrt{\frac{1}{3}} \\ \sqrt{\frac{2}{3}}(\sqrt{3}/2 + i/2) & 0 & \sqrt{\frac{2}{3}}(-\sqrt{3}/2 + i/2) \\ \sqrt{\frac{2}{3}}(1/2 + \sqrt{3}i/2) & -\sqrt{\frac{2}{3}} & \sqrt{\frac{2}{3}}(1/2 - \sqrt{3}i/2) \end{pmatrix}
\end{aligned}
$$

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.

Solution. We can write

$$
\cos\left(2\pi\frac{n+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right) = \cos\left(2\pi\frac{2n+1}{4N}\left(k+\frac{1}{2}\right)\right).
$$

If we consider these as vectors of length $2N$, we recognize these as the unit vectors $\boldsymbol{d}_{2n+1}$ in the $2N$-dimensional DCT, divided by the factor $d_n = \sqrt{2/(2N)} = \sqrt{1/N}$, so that these vectors have length $\sqrt{N}$. To see that these vectors are orthogonal when we restrict to the first N elements we use Equation (6.32) in the compendium as follows:

$$
\begin{aligned}
N\delta_{n_1,n_2} =& \sum_{k=0}^{2N-1} \cos\left(2\pi \frac{n_1+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)\cos\left(2\pi \frac{n_2+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right) \\
=& \sum_{k=0}^{N-1} \cos\left(2\pi \frac{n_1+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)\cos\left(2\pi \frac{n_2+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right) \\
&+ \sum_{k=N}^{2N-1} \cos\left(2\pi \frac{n_1+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)\cos\left(2\pi \frac{n_2+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right) \\
=& \sum_{k=0}^{N-1} \cos\left(2\pi \frac{n_1+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)\cos\left(2\pi \frac{n_2+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right) \\
&+ \sum_{k=0}^{N-1} \cos\left(2\pi \frac{n_1+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)\cos\left(2\pi \frac{n_2+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right) \\
=& 2\sum_{k=0}^{N-1} \cos\left(2\pi \frac{n_1+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)\cos\left(2\pi \frac{n_2+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right).
\end{aligned}
$$

This shows that

$$
\sum_{k=0}^{N-1} \cos\left(2\pi \frac{n_1+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right)\cos\left(2\pi \frac{n_2+\frac{1}{2}}{2N}\left(k+\frac{1}{2}\right)\right) = \frac{N}{2}\delta_{n_1,n_2},
$$

so that the vectors are orthogonal with lengths $\sqrt{N/2}$.

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.

Solution. Clearly, columns $0, \ldots, N/2-1$ of the MDCT are columns $N/2, \ldots, N-1$ of the DCT-IV. For the remaining columns, note first that, for $0 \leq k < N$, the properties

$$\begin{aligned} \cos(2\pi(n+1/2)((2N-1-k)+1/2)/(2N)) &= -\cos(2\pi(n+1/2)(k+1/2)/(2N)) \\ \cos(2\pi(n+1/2)((k+2N)+1/2)/(2N)) &= -\cos(2\pi(n+1/2)(k+1/2)/(2N)) \end{aligned}$$

are easy to verify. From the first property it follows that columns $N/2, \ldots, 3N/2-1$ of the MDCT are columns $N-1, N-2, \ldots, 0$ of the DCT-IV, with a sign change (they occur in opposite order). From the second property, it follows that columns $3N/2, \ldots, 2N-1$ of the MDCT are columns $0, \ldots, N/2-1$ of the DCT-IV, with a sign change. This means that, if $\boldsymbol{y}$ is a vector of length $2N$, the MDCT of $\boldsymbol{y}$ can be written as

$$\sqrt{\frac{N}{2}}\mathrm{DCT}_N^{(\mathrm{IV})} \left(\begin{array}{c} -y_{3N/2} - y_{3N/2-1} \\ -y_{3N/2+1} - y_{3N/2-2} \\ \vdots \\ -y_{2N-1} - y_N \\ \hline y_0 - y_{N-1} \\ y_1 - y_{N/2+1} \\ \vdots \\ y_{N/2-1} - y_{N/2} \end{array} \right).$$

The factor $\sqrt{\frac{N}{2}}$ was added since $\sqrt{\frac{2}{N}}$ was added in front of the cosine-matrix in order to make $\mathrm{DCT}_N^{(\mathrm{IV})}$ orthogonal. The result now follows by noting that we can write $\begin{pmatrix} \mathbf{0} & A \\ B & \mathbf{0} \end{pmatrix} \boldsymbol{y}$ for the vector on the right hand side, with A and B as defined in the text of the exercise.

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

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

(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.

Solution. Applying the MDCT first, and then the IMDCT, gives us the matrix

$$\frac{1}{2}\begin{pmatrix} \mathbf{0} & B^T \\ A^T & \mathbf{0} \end{pmatrix}\begin{pmatrix} \mathbf{0} & A \\ B & \mathbf{0} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} B^T B & \mathbf{0} \\ \mathbf{0} & A^T A \end{pmatrix}$$

Note that

$$A^T A = \begin{pmatrix} I_{N/2} & I^f_{N/2} \\ I^f_{N/2} & I_{N/2} \end{pmatrix} \qquad B^T B = \begin{pmatrix} I_{N/2} & -I^f_{N/2} \\ -I^f_{N/2} & I_{N/2} \end{pmatrix}.$$

Inserting this in the above gives

$$\frac{1}{2}\begin{pmatrix} I_{N/2} & -I^f_{N/2} & \mathbf{0} & \mathbf{0} \\ -I^f_{N/2} & I_{N/2} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I_{N/2} & I^f_{N/2} \\ \mathbf{0} & \mathbf{0} & I^f_{N/2} & I_{N/2} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} I_N - I^f_N & \mathbf{0} \\ \mathbf{0} & I_N + I^f_N \end{pmatrix}.$$

Assume now that we have computed the MDCT of $\begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \end{pmatrix} = (x_0, \ldots, x_{2N-1})$, and of $\begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix} = (x_N, \ldots, x_{3N-1})$. Performing the IMDCT on these two thus gives

$$\frac{1}{2}\begin{pmatrix} \boldsymbol{x}_0 - \boldsymbol{x}_0^{\text{rev}} \\ \boldsymbol{x}_1 + \boldsymbol{x}_1^{\text{rev}} \end{pmatrix} \text{ and } \frac{1}{2}\begin{pmatrix} \boldsymbol{x}_1 - \boldsymbol{x}_1^{\text{rev}} \\ \boldsymbol{x}_2 + \boldsymbol{x}_2^{\text{rev}} \end{pmatrix}$$

Adding the second component of the first and the first component of the second gives $\boldsymbol{x}_1$, which proves the result.

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) $0 \leq n < L$:

$$z_n = t_0 x_n + \sum_{k=1}^{n} t_k(x_{n+k} + x_{n-k}) + \sum_{k=n+1}^{L} t_k(x_{n+k} + x_{n-k+N}). \tag{4.1}$$

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

$$z_n = t_0 x_n + \sum_{k=1}^{L} t_k(x_{n+k} + x_{n-k}). \tag{4.2}$$

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

$$z_n = t_0 x_n + \sum_{k=1}^{N-1-n} t_k(x_{n+k} + x_{n-k}) + \sum_{k=N-1-n+1}^{L} t_k(x_{n+k-N} + x_{n-k}). \tag{4.3}$$

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.

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 in the compendium. 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

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

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 $e^{\pi in/(2N)}(y_n - iy_{N-n})$ 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 [**?**].

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

$$\Re(y_n) = \begin{cases} \Re((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) + (C_{N/4}\boldsymbol{z})_n & 0 \le n \le N/4-1 \\ \Re((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) & n = N/4 \\ \Re((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) - (C_{N/4}\boldsymbol{z})_{N/2-n} & N/4+1 \le n \le N/2-1 \end{cases} \tag{4.4}$$

$$\Im(y_n) = \begin{cases} \Im((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) & n = 0 \\ \Im((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) + (C_{N/4}\boldsymbol{w})_{N/4-n} & 1 \le n \le N/4-1 \\ \Im((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n) + (C_{N/4}\boldsymbol{w})_{n-N/4} & N/4 \le n \le N/2-1 \end{cases} \tag{4.5}$$

where $\boldsymbol{x}^{(e)}$ is as defined in Theorem 2.31 in the compendium, 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.

Solution. Taking real and imaginary parts in Equation (2.13) in the compendium for the FFT algorithm we obtain

$$\begin{aligned} \Re(y_n) &= \Re\left((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n + \Re((D_{N/2}\text{DFT}_{N/2}\boldsymbol{x}^{(o)})_n\right) \\ \Im(y_n) &= \Im\left((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n + \Im((D_{N/2}\text{DFT}_{N/2}\boldsymbol{x}^{(o)})_n\right), \end{aligned}$$

These equations explain the first terms $\Re((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n)$ and $\Im((\text{DFT}_{N/2}\boldsymbol{x}^{(e)})_n)$ on the right hand sides in equations (2.13) in the compendium and (4.5) in the compendium. It remains to rewrite $\Re\left((D_{N/2}\text{DFT}_{N/2}\boldsymbol{x}^{(o)})_n\right)$ and $\Im\left((D_{N/2}\text{DFT}_{N/2}\boldsymbol{x}^{(o)})_n\right)$ so that the remaining terms on the right hand sides can be seen. Let us first consider the equation for the real part with $0 \le n \le N/4-1$. In this case we can write

$$
\begin{aligned}
&\Re((D_{N/2}\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)})_n)\\
&= \Re\left(e^{-2\pi in/N}\sum_{k=0}^{N/2-1}(\boldsymbol{x}^{(o)})_k e^{-2\pi ink/(N/2)}\right) = \Re\left(\sum_{k=0}^{N/2-1}(\boldsymbol{x}^{(o)})_k e^{-2\pi in(k+\frac{1}{2})/(N/2)}\right)\\
&= \sum_{k=0}^{N/2-1}(\boldsymbol{x}^{(o)})_k \cos\left(2\pi\frac{n(k+\frac{1}{2})}{N/2}\right)\\
&= \sum_{k=0}^{N/4-1}(\boldsymbol{x}^{(o)})_k \cos\left(2\pi\frac{n(k+\frac{1}{2})}{N/2}\right)\\
&\quad + \sum_{k=0}^{N/4-1}(\boldsymbol{x}^{(o)})_{N/2-1-k} \cos\left(2\pi\frac{n(N/2-1-k+\frac{1}{2})}{N/2}\right)\\
&= \sum_{k=0}^{N/4-1}((\boldsymbol{x}^{(o)})_k + (\boldsymbol{x}^{(o)})_{N/2-1-k}) \cos\left(2\pi\frac{n\left(k+\frac{1}{2}\right)}{N/2}\right)\\
&= \sum_{k=0}^{N/4-1} z_k \cos\left(2\pi\frac{n\left(k+\frac{1}{2}\right)}{N/2}\right),
\end{aligned}
$$

where we have used that cos is periodic with period 2π, that cos is symmetric, and where z is the vector defined in the text of the theorem. When $0 \leq n \leq N/4-1$ this can also be written as

$$
\sum_{k=0}^{N/4-1}(C_{N/4})_{n,k}z_k = (C_{N/4}\boldsymbol{z})_n,
$$

This proves the first formula in Equation (4.4) in the compendium.

For $N/4+1 \leq n \leq N/2-1$, everything above is valid, except for that $\cos(2\pi n(k+1/2)/(N/2))$ are not entries in the matrix $C_{N/4}$, since n is outside the legal range of the indices. However, $N/2-n$ is now a legal index in $C_{N/4}$, and using that

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

we arrive at $-(C_{N/4}\boldsymbol{z})_{N/2-n}$ instead, and this proves the third formula in Equation (4.4) in the compendium. For the case $n = \frac{N}{4}$ all the cosine entries in the sum are zero, and this completes the proof of Equation (4.4) in the compendium.

For the imaginary part, using that sin is periodic with period 2π, and that sin is anti-symmetric, analogous calculations as above give

$$\Im((D_{N/2}\mathrm{DFT}_{N/2}\boldsymbol{x}^{(o)})_n) = \sum_{k=0}^{N/4-1} ((\boldsymbol{x}^{(o)})_{N/2-1-k} - (\boldsymbol{x}^{(o)})_k) \sin\left(2\pi \frac{n\left(k+\frac{1}{2}\right)}{N/2}\right). \tag{4.6}$$

Using that

$$\begin{aligned}\sin\left(2\pi\frac{n(k+\frac{1}{2})}{N/2}\right) &= \cos\left(\frac{\pi}{2} - 2\pi\frac{n(k+\frac{1}{2})}{N/2}\right) = \cos\left(2\pi\frac{(N/4-n)(k+\frac{1}{2})}{N/2} - k\pi\right)\\ &= (-1)^k\cos\left(2\pi\frac{(N/4-n)(k+\frac{1}{2})}{N/2}\right),\end{aligned}$$

Equation (4.6) in the compendium can be rewritten as

$$\begin{aligned}&\sum_{k=0}^{N/4-1} ((\boldsymbol{x}^{(o)})_{N/2-1-k} - (\boldsymbol{x}^{(o)})_k)(-1)^k \cos\left(2\pi\frac{(N/4-n)(k+\frac{1}{2})}{N/2}\right)\\ &= \sum_{k=0}^{N/4-1} w_k \cos\left(2\pi\frac{(N/4-n)(k+\frac{1}{2})}{N/2}\right),\end{aligned}$$

where $\boldsymbol{w}$ is the vector defined as in the text of the theorem. When $n=0$ this is 0 since all the cosines entries are zero. When $1 \leq n \leq N/4$ this is $(C_{N/4}\boldsymbol{w})_{N/4-n}$, since $\cos(2\pi(N/4-n)(k+1/2)/(N/2))$ are entries in the matrix $C_{N/4}$. This proves the second formula in Equation (4.5) in the compendium.

For $N/4 \leq n \leq N/2-1$ we can use that $\cos(2\pi(N/4-n)(k+1/2)/(N/2)) = \cos(2\pi(n-N/4)(k+1/2)/(N/2))$, which is an entry in the matrix $\mathrm{DCT}_{N/4}$ as well, so that we get $(C_{N/4}\boldsymbol{z})_{n-N/4}$. This also proves the third formula in Equation (4.5) in the compendium, and the proof is done.

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 in the compendium.

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

$$M_N = 2(M_{N/4} + 3N/8) + M_{N/2} \quad A_N = 2(A_{N/4} + 3N/8) + A_{N/2} + 3N/2 \tag{4.7}$$

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 in the compendium 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-(4.5) 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-(4.5) 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 $N/2$.

Chapter 5

Motivation for wavelets and some simple examples

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 $(f(0), f(1), \ldots .f(N-1))$.

Solution. We have that $f(t) = \sum_{n=0}^{N-1} c_n \phi_{0,n}$, where c_n are the coordinates of f in the basis $\{\phi_{0,0}, \phi_{0,1}, \ldots, \phi_{0,N-1}\}$. We now get that

$$f(k) = \sum_{n=0}^{N-1} c_n \phi_{0,n}(k) = c_k,$$

since $\phi_{0,n}(k) = 0$ when $n \neq k$. This shows that $(f(0), f(1), \ldots .f(N-1))$ are the coordinates of f.

Exercise 5.2: Proposition 5.12 in the compendium

Prove Proposition 5.12 in the compendium.

Solution. Since f is constant and equal to $f(n)$ on $[n, n+1/2)$, and constant and equal to $f(n+1/2)$ on $[n+1/2, n+1)$, we get that

$$
\begin{aligned}
\langle f, \phi_{0,n} \rangle &= \int_0^N f(t)\phi_{0,n}(t)dt = \int_n^{n+1} f(t)dt \\
&= \int_n^{n+1/2} f(t)dt + \int_{n+1/2}^{n+1} f(t)dt \\
&= \int_n^{n+1/2} f(n)dt + \int_{n+1/2}^{n+1} f(n+1/2)dt \\
&= f(n)/2 + f(n+1/2)/2 = (f(n) + f(n+1/2))/2.
\end{aligned}
$$

The orthogonal decomposition theorem gives that

$$
\text{proj}_{V_0} f = \sum_{n=0}^{N-1} \langle f, \phi_{0,n} \rangle \phi_{0,n} = \sum_{n=0}^{N-1} \frac{f(n) + f(n+1/2)}{2} \phi_{0,n}.
$$

Since $\phi_{0,n}$ is 1 on $[n, n+1)$ and 0 elsewhere, $\text{proj}_{V_0} f$ is the piecewise constant function which is equal to $(f(n) + f(n+1/2))/2$ on $[n, n+1)$.

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 in the compendium to write down the matrix for proj_{V_0} relative to the bases $\boldsymbol{\phi}_1$ and $\boldsymbol{\phi}_0$.

Solution. From Lemma 5.11 in the compendium it follows that

$$
\begin{aligned}
\text{proj}_{V_0}(\phi_{1,2n}) &= \phi_{0,n}/\sqrt{2} \\
\text{proj}_{V_0}(\phi_{1,2n+1}) &= \phi_{0,n}/\sqrt{2}
\end{aligned}
$$

This means that

$$
\begin{aligned}
[\text{proj}_{V_0}(\phi_{1,2n})]_{\boldsymbol{\phi}_0} &= \boldsymbol{e}_n/\sqrt{2} \\
[\text{proj}_{V_0}(\phi_{1,2n+1})]_{\boldsymbol{\phi}_0} &= \boldsymbol{e}_n/\sqrt{2}.
\end{aligned}
$$

These are the columns in the matrix for proj_{V_0} relative to the bases $\boldsymbol{\phi}_1$ and $\boldsymbol{\phi}_0$. This matrix is thus

$$
\frac{1}{\sqrt{2}} \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 1
\end{pmatrix}.
$$

b) Similarly, use Lemma 5.11 in the compendium to write down the matrix for $\text{proj}_{W_0} : V_1 \to W_0$ relative to the bases $\boldsymbol{\phi}_1$ and $\boldsymbol{\psi}_0$.

Solution. From Lemma 5.11 in the compendium it follows that

$$\begin{aligned}\text{proj}_{W_0}(\phi_{1,2n}) &= \psi_{0,n}/\sqrt{2}\\ \text{proj}_{W_0}(\phi_{1,2n+1}) &= -\psi_{0,n}/\sqrt{2}\end{aligned}$$

This means that

$$\begin{aligned}[\text{proj}_{W_0}(\phi_{1,2n})]_{\boldsymbol{\psi}_0} &= \boldsymbol{e}_n/\sqrt{2}\\ [\text{proj}_{W_0}(\phi_{1,2n+1})]_{\boldsymbol{\psi}_0} &= -\boldsymbol{e}_n/\sqrt{2}.\end{aligned}$$

These are the columns in the matrix for proj_{W_0} relative to the bases $\boldsymbol{\phi}_1$ and $\boldsymbol{\psi}_0$. This matrix is thus

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0\\ 0 & 0 & 1 & -1 & 0 & \cdots & 0 & 0 & 0\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -1 \end{pmatrix}.$$

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 $\text{proj}_{V_0}(\psi) = 0$.

Solution. Since $\phi \in V_0$ we must have that $\text{proj}_{V_0}(\phi) = \phi$. Since ψ is in the orthogonal complement of V_0 in V_1 we must have that $\text{proj}_{V_0}(\psi) = 0$.

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.

Solution. The first columns in the matrix of proj_{V_0} relative to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$ are

$$\begin{aligned}[\text{proj}_{V_0}(\phi_{0,0})]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} &= [\phi_{0,0}]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} = \boldsymbol{e}_0\\ [\text{proj}_{V_0}(\phi_{0,1})]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} &= [\phi_{0,1}]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} = \boldsymbol{e}_1\\ &\vdots\vdots\end{aligned}$$

The last columns in the matrix of proj_{V_0} relative to $(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)$ are

$$[\mathrm{proj}_{V_0}(\psi_{0,0})]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} = [\mathbf{0}]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} = \mathbf{0}$$
$$[\mathrm{proj}_{V_0}(\psi_{0,1})]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} = [\mathbf{0}]_{(\boldsymbol{\phi}_0,\boldsymbol{\psi}_0)} = \mathbf{0}$$
$$\vdots$$

It follows 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.

Solution. Follows in the same way as (b).

Exercise 5.5: Computing projections

Show that

$$\mathrm{proj}_{V_0}(f) = \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt \right) \phi_{0,n}(t) \tag{5.1}$$

for any f. Show also that the first part of Proposition 5.12 in the compendium follows from this.

Solution. We have that

$$\mathrm{proj}_{V_0}(f) = \sum_{n=0}^{N-1} \left(\int_0^{N} f(t)\phi_{0,n}(t)dt \right) \phi_{0,n} = \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt \right) \phi_{0,n},$$

where we have used the orthogonal decomposition formula. Note also that, if $f(t) \in V_1$, and $f_{n,1}$ is the value f attains on $[n, n+1/2)$, and $f_{n,2}$ is the value f attains on $[n+1/2, n+1)$, we have that

$$\begin{aligned}
\mathrm{proj}_{V_0}(f) &= \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt \right) \phi_{0,n}(t) \\
&= \sum_{n=0}^{N-1} \left(\frac{1}{2} f_{n,1} + \frac{1}{2} f_{n,2} \right) \phi_{0,n}(t) = \sum_{n=0}^{N-1} \frac{f_{n,1} + f_{n,2}}{2} \phi_{0,n}(t),
\end{aligned}$$

which is the function which is $(f_{n,1} + f_{n,2})/2$ on $[n, n+1)$. This proves the first part of Proposition 5.12 in the compendium.

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

Solution. We have that

$$\begin{aligned}\|f - \text{proj}_{V_0}(f)\|^2 &= \langle f - \text{proj}_{V_0}(f), f - \text{proj}_{V_0}(f)\rangle \\ &= \langle f, f\rangle - 2\langle f, \text{proj}_{V_0}(f)\rangle + \langle \text{proj}_{V_0}(f), \text{proj}_{V_0}(f)\rangle\end{aligned}$$

Now, note that

$$\langle \text{proj}_{V_0}(f), \text{proj}_{V_0}(f)\rangle = \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt\right)^2$$

from what we just showed in Exercise 5.5 (use that the $\phi_{0,n}$ are orthonormal). This means that the above can be written

$$\begin{aligned}&= \langle f, f\rangle - 2\sum_{n=0}^{N-1} \int_0^N \left(\int_n^{n+1} f(s)ds\right) \phi_{0,n}(t) f(t)dt + \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt\right)^2 \\ &= \langle f, f\rangle - 2\sum_{n=0}^{N-1} \int_n^{n+1} \left(\int_n^{n+1} f(s)ds\right) f(t)dt + \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt\right)^2 \\ &= \langle f, f\rangle - 2\sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt\right)^2 + \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt\right)^2 \\ &= \langle f, f\rangle - \sum_{n=0}^{N-1} \left(\int_n^{n+1} f(t)dt\right)^2.\end{aligned}$$

Exercise 5.7: Projecting on W_0

Show that

$$\text{proj}_{W_0}(f) = \sum_{n=0}^{N-1} \left(\int_n^{n+1/2} f(t)dt - \int_{n+1/2}^{n+1} f(t)dt\right) \psi_{0,n}(t) \tag{5.2}$$

for any f. Show also that the second part of Proposition 5.12 in the compendium follows from this.

Solution. The orthogonal decomposition theorem gives that

$$\begin{aligned}\text{proj}_{W_0}(f) &= \sum_{n=0}^{N-1}\langle f,\psi_{0,n}\rangle\psi_{0,n}(t) = \sum_{n=0}^{N-1}\left(\int_0^N f(t)\psi_{0,n}(t)dt\right)\psi_{0,n}(t)\\ &= \sum_{n=0}^{N-1}\left(\int_n^{n+1} f(t)\psi_{0,n}(t)dt\right)\psi_{0,n}(t)\\ &= \sum_{n=0}^{N-1}\left(\int_n^{n+1/2} f(t)dt - \int_{n+1/2}^{n+1} f(t)dt\right)\psi_{0,n}(t),\end{aligned}$$

where we used that $\psi_{0,n}$ is nonzero only on $[n, n+1)$, and is 1 on $[n, n+1/2)$, and -1 on $[n+1/2, n+1)$. Note also that, if $f(t) \in V_1$, and $f_{n,1}$ is the value f attains on $[n, n+1/2)$, and $f_{n,2}$ is the value f attains on $[n+1/2, n+1)$, we have that

$$\begin{aligned}\text{proj}_{W_0}(f) &= \sum_{n=0}^{N-1}\left(\int_n^{n+1/2} f(t)dt - \int_{n+1/2}^{n+1} f(t)dt\right)\psi_{0,n}(t)\\ &= \sum_{n=0}^{N-1}\left(\frac{1}{2}f_{n,1} - \frac{1}{2}f_{n,2}\right)\psi_{0,n}(t) = \sum_{n=0}^{N-1}\frac{f_{n,1}-f_{n,2}}{2}\psi_{0,n}(t),\end{aligned}$$

which is the function which is $(f_{n,1} - f_{n,2})/2$ on $[n, n+1/2)$, and $-(f_{n,1} - f_{n,2})/2$ on $[n+1/2, n+1)$. This proves the second part of Proposition 5.12 in the compendium.

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

$$c_{1,0} = \frac{1}{\sqrt{2}}(c_{0,0} + w_{0,0} + c_{0,(N+1)/2}) \qquad c_{1,N-1} = \frac{1}{\sqrt{2}}c_{0,(N+1)/2}. \tag{5.3}$$

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

$$c_{0,0} = \frac{1}{\sqrt{2}}(c_{1,0+c_{1,1}-c_{1,N-1}}) \quad w_{0,0} = \frac{1}{\sqrt{2}}(c_{1,0-c_{1,1}-c_{1,N-1}}) \quad c_{0,(N+1)/2} = \frac{1}{\sqrt{2}}2c_{1,N-1}. \tag{5.4}$$

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.

Exercise 5.9: Implement IDWT for The Haar wavelet

Write a function `IDWTKernelHaar` which uses the formulas (5.3) in the compendium to implement the IDWT, similarly to how the function `DWTKernelHaar` implemented the DWT using the formulas (5.4) in the compendium.

Solution. The following code can be used:

```
def IDWTKernelHaar(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the Haar wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    x /= sqrt(2)
    if mod(len(x), 2)==1:
        a, b = x[0] + x[1]  + x[-1], x[0] - x[1]
        x[0], x[1] = a, b
        for k in range(2,len(x) - 2, 2):
            a, b = x[k] + x[k+1], x[k] - x[k+1]
            x[k], x[k+1] = a, b
    else:
        for k in range(0,len(x) - 1, 2):
            a, b = x[k] + x[k+1], x[k] - x[k+1]
            x[k], x[k+1] = a, b
```

Exercise 5.10: Computing projections

Generalize Exercise 5.4 to the projections from V_{m+1} onto V_m and W_m.

Solution. Since $\phi_{m,n} \in V_m$ we must have that $T(\phi_{m,n}) = \phi_{m,n}$. Since $\psi_{m,n}$ is in the orthogonal complement of V_m in V_{m+1} we must have that $T(\psi_{m,n}) = 0$. The first half of the columns in the matrix of proj_{V_m} relative to $(\boldsymbol{\phi}_m, \boldsymbol{\psi}_m)$ are

$$
\begin{aligned}
[\text{proj}_{V_m}(\phi_{m,0})]_{(\boldsymbol{\phi}_m,\boldsymbol{\psi}_m)} &= [\phi_{m,0}]_{(\boldsymbol{\phi}_m,\boldsymbol{\psi}_m)} = \boldsymbol{e}_0 \\
[\text{proj}_{V_m}(\phi_{m,1})]_{(\boldsymbol{\phi}_m,\boldsymbol{\psi}_m)} &= [\phi_{m,1}]_{(\boldsymbol{\phi}_m,\boldsymbol{\psi}_m)} = \boldsymbol{e}_1 \\
&\vdots
\end{aligned}
$$

The second half of the columns are

$$
\begin{aligned}
[T(\psi_{m,0})]_{(\boldsymbol{\phi}_m,\boldsymbol{\psi}_m)} &= [\mathbf{0}]_{(\boldsymbol{\phi}_m,\boldsymbol{\psi}_m)} = \mathbf{0} \\
[T(\psi_{m,1})]_{(\boldsymbol{\phi}_m,\boldsymbol{\psi}_m)} &= [\mathbf{0}]_{(\boldsymbol{\phi}_m,\boldsymbol{\psi}_m)} = \mathbf{0} \\
&\vdots
\end{aligned}
$$

Thus, as before, the matrix of proj_{V_m} relative to $(\boldsymbol{\phi}_m, \boldsymbol{\psi}_m)$ is given by the diagonal matrix where the first half of the diagonal consists of 1's, and the second half consists of 0's. (c) follows in the same way.

Exercise 5.11: Scaling a function

Show that $f(t) \in V_m$ if and only if $g(t) = f(2t) \in V_{m+1}$.

Solution. If $f \in V_m$ we can write $f(t) = \sum_{n=0}^{2^m N-1} c_{m,n}\phi_{m,n}(t)$. We now get

$$
\begin{aligned}
g(t) = f(2t) &= \sum_{n=0}^{2^m N-1} c_{m,n}\phi_{m,n}(2t) = \sum_{n=0}^{2^m N-1} c_{m,n} 2^{m/2}\phi(2^m 2t - n) \\
&= \sum_{n=0}^{2^m N-1} c_{m,n} 2^{-1/2} 2^{(m+1)/2}\phi(2^{m+1}t - n) = \sum_{n=0}^{2^m N-1} c_{m,n} 2^{-1/2}\phi_{m+1,n}(t).
\end{aligned}
$$

This shows that $g \in V_{m+1}$. To prove the other way, assume that $g(t) = f(2t) \in V_{m+1}$. This means that we can write $g(t) = \sum_{n=0}^{2^{m+1} N-1} c_{m+1,n}\phi_{m+1,n}(t)$. We now have

$$
\begin{aligned}
f(t) = g(t/2) &= \sum_{n=0}^{2^{m+1}N-1} c_{m+1,n}\phi_{m+1,n}(t/2) = \sum_{n=0}^{2^{m+1}N-1} c_{m+1,n}2^{(m+1)/2}\phi(2^m t - n) \\
&= \sum_{n=0}^{2^m N-1} c_{m+1,n}2^{(m+1)/2}\phi(2^m t - n) + \sum_{n=2^m N}^{2^{m+1}N-1} c_{m+1,n}2^{(m+1)/2}\phi(2^m t - n) \\
&= \sum_{n=0}^{2^m N-1} c_{m+1,n}2^{(m+1)/2}\phi(2^m t - n) + \sum_{n=0}^{2^m N-1} c_{m+1,n+2^m N}2^{(m+1)/2}\phi(2^m t - n - 2^m N) \\
&= \sum_{n=0}^{2^m N-1} c_{m+1,n}2^{(m+1)/2}\phi(2^m t - n) + \sum_{n=0}^{2^m N-1} c_{m+1,n+2^m N}2^{(m+1)/2}\phi(2^m t - n) \\
&= \sum_{n=0}^{2^m N-1} (c_{m+1,n} + c_{m+1,n+2^m N})2^{1/2}2^{m/2}\phi(2^m t - n) \\
&= \sum_{n=0}^{2^m N-1} (c_{m+1,n} + c_{m+1,n+2^m N})2^{1/2}\phi_{m,n}(t) \in V_m
\end{aligned}
$$

The thing which made this a bit difficult was that the range of the n-indices here was outside $[0, 2^m N - 1]$ (which describe the legal indices in the basis V_m), so that we had to use the periodicity of ϕ.

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.

Solution. By definition, $[T_1]_{\mathcal{B}_1} \oplus [T_2]_{\mathcal{B}_2} \oplus \cdots \oplus [T_n]_{\mathcal{B}_n}$ is a block matrix where the blocks on the diagonal are the matrices $[T_1]_{\mathcal{B}_1}$, $[T_2]_{\mathcal{B}_2}$, and so on. If $\boldsymbol{b}_i$ are the basis vectors in $\mathcal{B}_i$, the columns in $[T_i]_{\mathcal{B}_i}$ are $[T(\boldsymbol{b}_j)]_{\mathcal{B}_i}$. This means that

$[T_1]_{\mathcal{B}_1} \oplus [T_2]_{\mathcal{B}_2} \oplus \cdots \oplus [T_n]_{\mathcal{B}_n}$ has $[T(\boldsymbol{b}_j)]_{\mathcal{B}_i}$ in the j'th block, and $\mathbf{0}$ elsewhere. This means that we can write it as

$$\mathbf{0} \oplus \cdots \mathbf{0} \oplus [T(\boldsymbol{b}_j)]_{\mathcal{B}_i} \oplus \mathbf{0} \cdots \mathbf{0}.$$

On the other hand, $[T_1 \oplus T_2 \oplus \ldots \oplus T_n]_{(\mathcal{B}_1,\mathcal{B}_2,\ldots,\mathcal{B}_n)}$ is a matrix of the same size, and the corresponding column to that of the above is

$$\begin{aligned}
&[(T_1 \oplus T_2 \oplus \ldots \oplus T_n)(\mathbf{0} \oplus \cdots \mathbf{0} \oplus \boldsymbol{b}_j \oplus \mathbf{0} \cdots \mathbf{0})]_{(\mathcal{B}_1,\mathcal{B}_2,\ldots,\mathcal{B}_n)} \\
&= [\mathbf{0} \oplus \cdots \mathbf{0} \oplus T(\boldsymbol{b}_j) \oplus \mathbf{0} \cdots \mathbf{0}]_{(\mathcal{B}_1,\mathcal{B}_2,\ldots,\mathcal{B}_n)} \\
&= \mathbf{0} \oplus \cdots \mathbf{0} \oplus [T(\boldsymbol{b}_j)]_{\mathcal{B}_i} \oplus \mathbf{0} \cdots \mathbf{0}.
\end{aligned}$$

Here $\boldsymbol{b}_j$ occurs as the i'th summand. This is clearly the same as what we computed for the right hand side above.

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?

Solution. Assume that λ is an eigenvalue common to both T_1 and T_2. Then there exists a vector $\boldsymbol{v}_1$ so that $T_1\boldsymbol{v}_1 = \lambda\boldsymbol{v}_1$, and a vector $\boldsymbol{v}_2$ so that $T_2\boldsymbol{v}_2 = \lambda\boldsymbol{v}_2$. We now have that

$$\begin{aligned}
(T_1 \oplus T_2)(\boldsymbol{v}_1 \oplus \boldsymbol{v}_2) &= \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \end{pmatrix} \\
&= \begin{pmatrix} T_1\boldsymbol{v}_1 \\ T_2\boldsymbol{v}_2 \end{pmatrix} = \begin{pmatrix} \lambda\boldsymbol{v}_1 \\ \lambda\boldsymbol{v}_2 \end{pmatrix} \\
&= \lambda \begin{pmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \end{pmatrix} = \lambda(\boldsymbol{v}_1 \oplus \boldsymbol{v}_2).
\end{aligned}$$

This shows that λ is an eigenvalue for λ also, and that $\boldsymbol{v}_1 \oplus \boldsymbol{v}_2$ is a corresponding eigenvector.

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 $(A \oplus B)^{-1} = A^{-1} \oplus B^{-1}$.

Solution. We have that

$$
\begin{aligned}
(A \oplus B)(A^{-1} \oplus B^{-1}) &= \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \begin{pmatrix} A^{-1} & 0 \\ 0 & B^{-1} \end{pmatrix} \\
&= \begin{pmatrix} AA^{-1} & 0 \\ 0 & BB^{-1} \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix} = I
\end{aligned}
$$

where we have multiplied as block matrices. This proves that $A \oplus B$ is invertible, and states what the inverse is.

Exercise 5.15: Multiplying direct sums

Let A, B, C, D be square matrices of the same dimensions. Show that $(A \oplus B)(C \oplus D) = (AC) \oplus (BD)$.

Solution. We have that

$$
(A \oplus B)(C \oplus D) = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \begin{pmatrix} C & 0 \\ 0 & D \end{pmatrix} = \begin{pmatrix} AC & 0 \\ 0 & BD \end{pmatrix} = (AC) \oplus (BD)
$$

where we again have multiplied as block matrices.

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 5.1 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.

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) in the compendium 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.

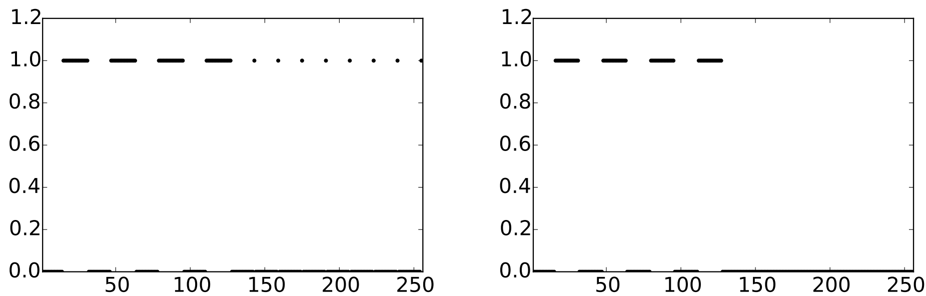

Figure 5.1: 2 vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ which seem equal, but where the DWT's are very different.

Solution. The following code achieves this:

```
[x, fs] = audioread('sounds/castanets.wav')
DWTImpl(x[0:2**17,0], 2, DWTKernelHaar)
plt.plot(x[0:2**17,0])
plt.axis([0,2**17,-1,1])
```

The values from V_0 corresponds to the first 1/4 values in the plot, the values from W_0 corresponds to the next 1/4 values in the plot, while the values from W_1 correspond to the last 1/2 of the values in the plot.

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 $[-1, 1]$?

Solution. There is no reason to believe that sound samples returned by the function lie in $[-1, 1]$. you can check this by printing the maximum value in the returned array on screen inside this method.

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 in the compendium, where you performed a DFT on the sound sample instead, and set some of the DFT coefficients to zero.

Solution. For $m = 2$ we clearly hear a degradation in the sound. For $m = 4$ and above most of the sound is unrecognizable.

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?

Solution. The following code achieves the task

```
def playDWT(m, f, invf, lowres = True):
    """
    Play a sound after removing either the detail or the lowres part.

    m: The number of resolutions
    f: The DWT kernel
    invf: The IDWT kernel
    lowres: If true, set the detail to 0 and play the lowres part.
            If false, set the lowres part to 0 and play the detail.
    """
    x, fs = audioread('sounds/castanets.wav')
    N = 2**17
    x = x[0:N]
    DWTImpl(x, m, f)
    if lowres:
        x[(N/2**m):N] = 0
    else:
        x[0:(N/2**m)] = 0
    IDWTImpl(x, m, invf)
    play(x, fs)
```

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 in the compendium, 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 in the compendium.

Solution. Note first that, similarly to the computation in Exercise 5.7, we have that

$$\int_0^N f(t)\psi_{m,n}(t)dt = 2^{m/2}\left(\int_{n2^{-m}}^{(n+1/2)2^{-m}} f(t)dt - \int_{(n+1/2)2^{-m}}^{(n+1)2^{-m}} f(t)dt\right).$$

With $f(t) = 1 - 2|1/2 - t/N|$ we have two possibilities: when $n < N2^{m-1}$ we have that $[n2^{-m}, (n+1)2^{-m}) \subset [0, N/2]$, so that $f(t) = 2t/N$, and we get

$$\begin{aligned}
w_{m,n} &= 2^{m/2}\left(\int_{n2^{-m}}^{(n+1/2)2^{-m}} 2t/N dt - \int_{(n+1/2)2^{-m}}^{(n+1)2^{-m}} 2t/N dt\right)\\
&= 2^{m/2}[t^2/N]_{n2^{-m}}^{(n+1/2)2^{-m}} - 2^{m/2}[t^2/N]_{(n+1/2)2^{-m}}^{(n+1)2^{-m}}\\
&= \frac{2^{-3m/2}}{N}\left(2(n+1/2)^2 - n^2 - (n+1)^2\right) = -\frac{2^{-3m/2-1}}{N}.
\end{aligned}$$

When $n \geq N2^{m-1}$ we have that $f(t) = 2-2t/N$, and using that $\int_0^N \psi_{m,n}(t)dt = 0$ we must get that $w_{m,n} = \frac{2^{-3m/2-1}}{N}$.

For $f(t) = 1/2 + \cos(2\pi t/N)/2$, note first that this has the same coefficients as $\cos(2\pi t/N)/2$, since $\int_0^N \psi_{m,n}(t)dt = 0$. We now get

$$\begin{aligned}
w_{m,n} &= 2^{m/2}\left(\int_{n2^{-m}}^{(n+1/2)2^{-m}} \cos(2\pi t/N)/2 dt - \int_{(n+1/2)2^{-m}}^{(n+1)2^{-m}} \cos(2\pi t/N)/2 dt\right)\\
&= 2^{m/2}[N\sin(2\pi t/N)/(4\pi)]_{n2^{-m}}^{(n+1/2)2^{-m}} - 2^{m/2}[N\sin(2\pi t/N)/(4\pi)]_{(n+1/2)2^{-m}}^{(n+1)2^{-m}}\\
&= \frac{2^{m/2-2}N}{\pi}\left(2\sin(2\pi(n+1/2)2^{-m}/N) - \sin(2\pi n2^{-m}/N) - \sin(2\pi(n+1)2^{-m}/N)\right).
\end{aligned}$$

There seems to be no more possibilities for simplification here.

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 in the compendium. How do these compare with the coefficients from the Exercise 5.21?

Solution. We get

$$\begin{aligned}
w_{m,n} &= 2^{m/2}\left(\int_{n2^{-m}}^{(n+1/2)2^{-m}} (t/N)^k dt - \int_{(n+1/2)2^{-m}}^{(n+1)2^{-m}} (t/N)^k dt\right) \\
&= 2^{m/2}[t^{k+1}/((k+1)N^k)]_{n2^{-m}}^{(n+1/2)2^{-m}} - 2^{m/2}[t^{k+1}/((k+1)N^k)]_{(n+1/2)2^{-m}}^{(n+1)2^{-m}} \\
&= \frac{2^{-m(k+1/2)}}{(k+1)N^k}\left(2(n+1/2)^{k+1} - n^{k+1} - (n+1)^{k+1}\right).
\end{aligned}$$

The leading term n^{k+1} will here cancel, but the others will not, so there is no room for further simplification here.

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 $\boldsymbol{x}$.

Solution. The vector $\boldsymbol{x}$ is the coordinate vector of the function $f(t) = \sum_{n=0}^{1023}(-1)^n\phi_{10,n}$ in the basis $\boldsymbol{\phi}_{10}$ for V_{10}. Since $\phi_{10,2n} - \phi_{10,2n+1} = \sqrt{2}\psi_{9,n}$, we can write $f(t) = \sum_{n=0}^{1023}\sqrt{2}\psi_{9,n}$. Since a 10-level-DWT gives as a result the coordinate vector of f in

$$(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0, \boldsymbol{\psi}_1, \boldsymbol{\psi}_2, \boldsymbol{\psi}_3, \boldsymbol{\psi}_4, \boldsymbol{\psi}_5, \boldsymbol{\psi}_6, \boldsymbol{\psi}_7, \boldsymbol{\psi}_8, \boldsymbol{\psi}_9),$$

(the DWT is nothing but the change of coordinates from $\boldsymbol{\phi}_{10}$ to this basis), and since $f(t) = \sum_{n=0}^{1023}\sqrt{2}\psi_{9,n}$, it is clear that the coordinate vector of f in this basis has $\sqrt{2}$ in the second part (the $\boldsymbol{\psi}_9$-coordinates), and 0 elsewhere. The 10-level DWT of $\boldsymbol{x}$ therefore gives the vector of length 1024 which is 0 on the first half, and equal to $\sqrt{2}$ on the second half. $m = 10$ is here arbitrarily chosen: The result would have been the same for $m = 1, m = 2$, and so on. The following code verifies the result:

```
x = tile([1.,-1.], 512)
DWTImpl(x, 10, DWTKernelHaar)
print 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.

Solution. The following code can be used.

```
def DWTKernelHaar(x):
    x /= sqrt(2)
    if mod(len(x), 2)==1:
        a, b = x[0] + x[1] - x[-1], x[0] - x[1] - x[-1]
        x[0], x[1] = a, b
        x[-1] *= 2
    else:
        a, b = x[0] + x[1], x[0] - x[1]
        x[0], x[1] = a, b
    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
```

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`.

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.

Solution. Let us write $f(t) = \sum_{n=0}^{N-1} c_n \phi_{0,n}(t)$. If k is an integer we have that

$$f(k) = \sum_{n=0}^{N-1} c_n \phi_{0,n}(k) = \sum_{n=0}^{N-1} c_n \phi(k-n).$$

Clearly the only integer for which $\phi(s) \neq 0$ is $s = 0$ (since $\phi(0) = 1$), so that the only n which contributes in the sum is $n = k$. This means that $f(k) = c_k$, so that $[f]_{\boldsymbol{\phi}_0} = (f(0), f(1), \ldots, f(N-1))$.

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 $\int_n^{n+1} f(t)^2 dt = (x_n^2 + x_n x_{n+1} + x_{n+1}^2)/3$.

Solution. We have that

$$\begin{aligned}
\int_n^{n+1} f(t)^2 dt &= \int_n^{n+1} (x_n + (x_{n+1} - x_n)(t-n))^2 dt = \int_0^1 (x_n + (x_{n+1} - x_n)t)^2 dt \\
&= \int_0^1 (x_n^2 + 2x_n(x_{n+1} - x_n)t + (x_{n+1} - x_n)^2 t^2) dt \\
&= \left[x_n^2 t + x_n(x_{n+1} - x_n)t^2 + (x_{n+1} - x_n)^2 t^3/3\right]_0^1 \\
&= x_n^2 + x_n(x_{n+1} - x_n) + (x_{n+1} - x_n)^2/3 = \frac{1}{3}(x_n^2 + x_n x_{n+1} + x_{n+1}^2).
\end{aligned}$$

b) Show that

$$\begin{aligned}
\int_0^{1/2} (x_0 + (x_1 - x_0)t)\phi_{1,1}(t) dt &= 2\sqrt{2}\left(\frac{1}{12}x_0 + \frac{1}{24}x_1\right) \\
\int_{1/2}^1 (x_0 + (x_1 - x_0)t)\phi_{1,1}(t) dt &= 2\sqrt{2}\left(\frac{1}{24}x_0 + \frac{1}{12}x_1\right).
\end{aligned}$$

Solution. We have that

$$\begin{aligned}
&\int_0^{1/2} (x_0 + (x_1 - x_0)t)\phi_{1,1}(t) dt \\
&= \int_0^{1/2} (x_0 + (x_1 - x_0)t) 2\sqrt{2} t dt = 2\sqrt{2} \int_0^{1/2} (x_0 t + (x_1 - x_0)t^2) dt \\
&= 2\sqrt{2}\left[\frac{1}{2}x_0 t^2 + \frac{1}{3}(x_1 - x_0)t^3\right]_0^{1/2} = 2\sqrt{2}\left(\frac{1}{8}x_0 + \frac{1}{24}(x_1 - x_0)\right) \\
&= 2\sqrt{2}\left(\frac{1}{12}x_0 + \frac{1}{24}x_1\right).
\end{aligned}$$

In the same way

$$\begin{aligned}
&\int_{1/2}^1 (x_0 + (x_1 - x_0)t)\phi_{1,1}(t) dt \\
&= \int_{1/2}^1 (x_0 + (x_1 - x_0)t) 2\sqrt{2}(1-t) t dt = 2\sqrt{2} \int_{1/2}^1 (x_0 + (x_1 - 2x_0)t - (x_1 - x_0)t^2) t dt \\
&= 2\sqrt{2}\left[x_0 t + \frac{1}{2}(x_1 - 2x_0)t^2 - \frac{1}{3}(x_1 - x_0)t^3\right]_{1/2}^1 = 2\sqrt{2}\left(\frac{1}{2}x_0 + \frac{3}{8}(x_1 - 2x_0) - \frac{7}{24}(x_1 - x_0)\right) \\
&= 2\sqrt{2}\left(\frac{1}{24}x_0 + \frac{1}{12}x_1\right).
\end{aligned}$$

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 $\|\phi_{1,1}(t) - \sum_{n=0}^{N-1} x_n\phi_{0,n}(t)\|^2$.

Solution. Using a) and b) we see that the above can be written as

$$\frac{2}{3} + \sum_{n=0}^{N-1} \frac{1}{3}(x_n^2 + x_n x_{n+1} + x_{n+1}^2) - 2\left(2\sqrt{2}\left(\frac{1}{12}x_0 + \frac{1}{24}x_1\right) - 2\sqrt{2}\left(\frac{1}{24}x_0 + \frac{1}{12}x_1\right)\right)$$
$$= \frac{2}{3} + \frac{2}{3}\sum_{n=0}^{N-1} x_n^2 + \frac{1}{3}\sum_{n=0}^{N-1} x_n x_{n+1} - \frac{\sqrt{2}}{2}(x_0 + x_1).$$

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 $k \geq 2$.

Solution. We see that the partial derivatives of the function in c. are

$$\begin{aligned}
\frac{\partial f}{\partial x_0} &= \frac{1}{3}x_{N-1} + \frac{4}{3}x_0 + \frac{1}{3}x_1 - \frac{\sqrt{2}}{2} \\
\frac{\partial f}{\partial x_1} &= \frac{1}{3}x_0 + \frac{4}{3}x_1 + \frac{1}{3}x_1 - \frac{\sqrt{2}}{2} \\
\frac{\partial f}{\partial x_i} &= \frac{1}{3}x_{i-1} + \frac{4}{3}x_i + \frac{1}{3}x_{i+1} \; 2 \leq i < N-1 \\
\frac{\partial f}{\partial x_{N-1}} &= \frac{1}{3}x_{N-2} + \frac{4}{3}x_{N-1} + \frac{1}{3}x_0.
\end{aligned}$$

Moving the two terms $\frac{\sqrt{2}}{2}$ over to the right hand side, setting the gradient equal to zero is the same as solving the system $S\boldsymbol{x} = \boldsymbol{b}$ which we stated.

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 $[0, N]$.

Solution. The following code can be used

```
from numpy import *
import matplotlib.pyplot as plt

N = 16
S = zeros((N, N))
S[0,N-1] = 1/3.; S[0,0] = 4/3.; S[0,1] = 1/3.; # First row
for k in range(1,N-1):
    S[k,(k-1):(k+2)] = [1/3., 4/3., 1/3.]
S[N-1,N-2] = 1/3.; S[N-1,N-1]=4/3.; S[N-1,0]=1/3.; # Last row
b=zeros(N); b[0]=sqrt(2)/2; b[1]=sqrt(2)/2;
plt.plot(range(0,N),linalg.solve(S,b)) # Plots the projection
plt.show()
```

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.

Solution. We have that

$$\begin{aligned}
\langle \phi_{0,n}, \phi_{0,n} \rangle &= \int_{n-1}^{n+1} (1 - |t-n|)^2 dt \\
&= \int_{n-1}^{n+1} \left(1 - 2|t-n| + (t-n)^2\right) dt \\
&= 2 - 2 + \left[\frac{1}{3}(t-n)^3\right]_{n-1}^{n+1} = \frac{2}{3}.
\end{aligned}$$

We also have

$$\begin{aligned}
\langle \phi_{0,n}, \phi_{0,n+1} &= \int_n^{n+1} (1-(t-n))(1+(t-n-1))dt = \int_0^1 (1-u)(1+u-1)du \\
&= \int_0^1 (t-t^2)dt = \frac{1}{2} - \frac{1}{3} = \frac{1}{6}.
\end{aligned}$$

Finally, the supports of $\phi_{0,n}$ and $\phi_{0,n\pm k}$ are disjoint for $k > 1$, so that we must have $\langle \phi_{0,n}, \phi_{0,n\pm k} \rangle = 0$ in that case.

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.

Solution. We have that

$$\chi_{[-1/2,1/2)} * \chi_{[-1/2,1/2)}(x) = \int_{-\infty}^{\infty} \chi_{[-1/2,1/2)}(t)\chi_{[-1/2,1/2)}(x-t)dt.$$

The integrand here is 1 when $-1/2 < t < 1/2$ and $-1/2 < x - t < 1/2$, or in other words when $\max(-1/2, -1/2 + x) < t < \min(1/2, 1/2 + x)$ (else it is 0). When $x > 0$ this happens when $-1/2 + x < t < 1/2$, and when $x < 0$ this happens when $-1/2 < t < 1/2 + x$. This means that

$$\chi_{[-1/2,1/2)} * \chi_{[-1/2,1/2)}(x) = \begin{cases} \int_{-1/2+x}^{1/2} dt = 1 - x & , x > 0 \\ \int_{-1/2}^{1/2+x} dt = 1 + x & , x < 0. \end{cases}$$

But this is by definition ϕ.

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

$$\hat{\psi}(t) = \begin{cases} -\alpha t - \alpha & \text{for } -1 \le t < 0 \\ (2 + \alpha - \beta)t - \alpha & \text{for } 0 \le t < 1/2 \\ (\alpha - \beta - 2)t - \alpha + 2 & \text{for } 1/2 \le t < 1 \\ \beta t - 2\beta & \text{for } 1 \le t < 2 \\ 0 & \text{for all other } t \end{cases}$$

Solution. The function $\hat{\psi}$ is a sum of the functions $\psi = \phi_{1,1}$, ϕ, and $\phi_{0,1}$ (i.e. we have set $n = 0$ in Equation (5.36) in the compendium). All these are continuous and piecewise linear, and we can write

$$\phi_{1,1}(t) = \begin{cases} 2t & 0 \le t < 1/2 \\ 2-2t & 1/2 \le t < 1 \\ 0 & \text{elsewhere} \end{cases}$$

$$\phi(t)(t) = \begin{cases} 1+t & -1 \le t < 0 \\ 1-t & 0 \le t < 1 \\ 0 & \text{elsewhere} \end{cases}$$

$$\phi_{0,1}(t) = \begin{cases} t & 0 \le t < 1 \\ 2-t & 1 \le t < 2 \\ 0 & \text{elsewhere} \end{cases}.$$

It follows that $\hat{\psi}(t) = \phi_{1,1}(t) - \alpha\phi(t) - \beta\phi_{1,1}$ is piecewise linear, and linear on the segments $[-1,0]$, $[0,1/2]$, $[1/2,1]$, $[1,2]$.

On the segment $[-1,0]$ only the function ϕ is seen to be nonzero, and since $\phi(t) = 1+t$ here, we have that $\hat{\psi}(t) = -\alpha(1+t) = -\alpha - \alpha t$ here.

On the segment $[0,1/2]$ all three functions are nonzero, and

$$\begin{aligned} \phi_{1,1}(t) &= 2t \\ \phi(t)(t) &= 1-t \\ \phi_{0,1}(t) &= t \end{aligned}$$

on this interval. This means that $\hat{\psi}(t) = 2t - \alpha(1-t) - \beta t = (2+\alpha-\beta)t - \alpha$ on $[0,1/2]$.

On the segment $[0,1/2]$ all three functions are nonzero, and

$$\begin{aligned} \phi_{1,1}(t) &= 2-2t \\ \phi(t)(t) &= 1-t \\ \phi_{0,1}(t) &= t \end{aligned}$$

on this interval. This means that $\hat{\psi}(t) = 2-2t-\alpha(1-t)-\beta t = (\alpha-\beta-2)t-\alpha+2$ on $[1/2,1]$.

On the segment $[1,2]$ only the function $\phi_{0,1}$ is seen to be nonzero, and since $\phi_{0,1}(t) = 2-t$ here, we have that $\hat{\psi}(t) = -\beta(2-t) = \beta t - 2\beta$ here. For all other values of t, $\hat{\psi}$ is zero. This proves the formulas for $\hat{\psi}$ on the different intervals.

b) Show that

$$\int_0^N \hat{\psi}(t)dt = \frac{1}{2} - \alpha - \beta, \qquad \int_0^N t\hat{\psi}(t)dt = \frac{1}{4} - \beta.$$

Solution. We can write

$$\begin{aligned}
\int_0^N \hat{\psi}(t)dt &= \int_{-1}^2 \hat{\psi}(t)dt = \int_{-1}^0 \hat{\psi}(t)dt + \int_0^{1/2} \hat{\psi}(t)dt + \int_{1/2}^1 \hat{\psi}(t)dt + \int_1^2 \hat{\psi}(t)dt \\
&= \int_{-1}^0 (-\alpha - \alpha t)dt + \int_0^{1/2} (2 + \alpha - \beta)t - \alpha)dt \\
&+ \int_{1/2}^1 ((\alpha - \beta - 2)t - \alpha + 2)dt + \int_1^2 (\beta t - 2\beta)dt \\
&= \left[-\alpha t - \frac{1}{2}\alpha t^2\right]_{-1}^0 + \left[\frac{1}{2}(2 + \alpha - \beta)t^2 - \alpha t\right]_0^{1/2} \\
&+ \left[\frac{1}{2}(\alpha - \beta - 2)t^2 + (2 - \alpha)t\right]_{1/2}^1 + \left[\frac{1}{2}\beta t^2 - 2\beta t\right]_1^2 \\
&= -\alpha + \frac{1}{2}\alpha + \frac{1}{8}(2 + \alpha - \beta) - \frac{1}{2}\alpha + \frac{3}{8}(\alpha - \beta - 2) + \frac{1}{2}(2 - \alpha) + \frac{3}{2}\beta - 2\beta \\
&= \frac{1}{2} - \alpha - \beta,
\end{aligned}$$

$\int_0^N t\hat{\psi}(t)dt$ is computed similarly, so that we in the end arrive at $\frac{1}{4} - \beta$.

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.

Solution. The equation system

$$\begin{aligned}
\frac{1}{2} - \alpha - \beta &= 0 \\
\frac{1}{4} - \beta &= 0
\end{aligned}$$

has the unique solution $\alpha = \beta = \frac{1}{4}$, which we already have found.

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)
```

Solution. In order for ψ to have vanishing moments we must have that $\int \hat{\psi}(t)dt = \int t\hat{\psi}(t)dt = 0$ Substituting $\hat{\psi} = \psi - \alpha\phi_{0,0} - \beta\phi_{0,1}$ we see that, for $k = 0, 1$,

$$\int t^k \left(\alpha\phi_{0,0} + \beta\phi_{0,1}\right) dt = \int t^k \psi(t) dt.$$

The left hand side can here be written

$$\begin{aligned} \int t^k \left(\alpha\phi_{0,0} + \beta\phi_{0,1}\right) dt &= \alpha \int t^k \phi_{0,0} dt + \beta \int t^k \phi_{0,1}(t) dt \\ &= \alpha \int_{-1}^{1} t^k (1 - |t|) dt + \beta \int_0^2 t^k (1 - |t-1|) dt = \alpha a_k + \beta b_k. \end{aligned}$$

The right hand side is

$$\int t^k \psi(t) dt = \int t^k \phi_{1,1}(t) dt = \int_0^1 (1 - 2|t - 1/2|) dt = e_k.$$

The following program sets up the corresponding equation systems, and solves it by finding α, β.

```
A = zeros((2, 2))
b = zeros((2, 1))
for k in range(2):
    res1, err1 = quad(lambda t: t**k*(1-abs(t)), -1, 1)
    res2, err2 = quad(lambda t: t**k*(1-abs(t-1)), 0, 2)
    res3, err3 = quad(lambda t: t**k*(1-2*abs(t-1/2.)), 0, 1)
    A[k,:] = [res1, res2]
    b[k] = res3
linalg.solve(A,b)
```

b) The procedure where we set up a matrix equation in a) allows for generalization to more vanishing moments. Define

$$\hat{\psi} = \psi_{0,0} - \alpha\phi_{0,0} - \beta\phi_{0,1} - \gamma\phi_{0,-1} - \delta\phi_{0,2}. \tag{5.5}$$

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.

Solution. Similarly to a), Equation (5.5) in the compendium gives that

$$\int t^k \left(\alpha\phi_{0,0} + \beta\phi_{0,1} + \gamma\phi_{0,-1} + \delta\phi_{0,2}\right) dt = \int t^k \psi(t) dt.$$

The correspodning equation system is deduced exactly as in a). The following program sets up the corresponding equation systems, and solves it by finding $\alpha, \beta, \gamma, \delta$.

```
A=zeros((4, 4))
b=zeros((4, 1))
for k in range(4):
    res1, err1 = quad(lambda t: t**k*(1-abs(t)), -1, 1)
    res2, err2 = quad(lambda t: t**k*(1-abs(t-1)), 0, 2)
    res3, err3 = quad(lambda t: t**k*(1-abs(t+1)), -2, 0)
    res4, err4 = quad(lambda t: t**k*(1-abs(t-2)), 1, 3)
    res5, err5 = quad(lambda t: t**k*(1-2*abs(t-1/2.)), 0, 1)
    A[k,:] = [res1, res2, res3, res4]
    b[k] = res5
coeffs = linalg.solve(A,b)
```

c) Plot the function defined by (5.5) 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 $\phi_{1,1}(t)$.

Solution. The function $\hat{\psi}$ now is supported on $[-2,3]$, and can be plotted as follows:

```
t=linspace(-2,3,100)
plt.plot(t, (t >= 0)*(t <= 1)*(1-2*abs(t - 0.5)) \
        -coeffs[0]*(t >= -1)*(t <= 1)*(1 - abs(t)) \
        -coeffs[1]*(t >= 0)*(t <= 2)*(1 - abs(t - 1)) \
        -coeffs[2]*(t >= -2)*(t <= 0)*(1 - abs(t + 1)) \
        -coeffs[3]*(t >= 1)*(t <= 3)*(1 - abs(t - 2)))
```

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.

Solution. If we define

$$\hat{\psi} = \psi_{0,0} - \sum_{k=0}^{K} \left(\alpha_k \phi_{0,-k} - \beta_k \phi_{0,k+1}\right),$$

we have $2k$ unknowns. These can be determined if we require $2k$ 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

$$\begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,k-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,k-1} \\ \vdots & \vdots & \vdots & \vdots \\ c_{k-1,0} & c_{k-1,1} & \cdots & c_{k-1,k-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{k-1} \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{k-1} \end{pmatrix} \tag{5.6}$$

b) Write a function `vanishingmomshaar` which takes k as input, solves Equation (5.6) 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.

Solution. The following code can be used:

```
playDWT(m, DWTKernelHaar, IDWTKernelHaar, False)
playDWT(m, DWTKernelpwl0, IDWTKernelpwl0, False)
playDWT(m, DWTKernelpwl2, IDWTKernelpwl2, False)
```

Exercise 5.35: Prove expression for S_r

Prove Theorem 5.43 in the compendium. Use the proof of Theorem 4.9 in the compendium as a guide.

Solution. We compute

$$S_r\boldsymbol{x} = \begin{pmatrix} S_1 & S_2 \end{pmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_{N-2} \\ x_{N-1} \\ x_{N-2} \\ \vdots \\ x_1 \end{pmatrix} = S_1 \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} + S_2 \begin{pmatrix} x_{N-2} \\ \vdots \\ x_1 \end{pmatrix} = S_1 \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} + (S_2)^f \begin{pmatrix} x_1 \\ \vdots \\ x_{N-2} \end{pmatrix}$$

$$= S_1 \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} + \begin{pmatrix} 0 & (S_2)^f & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} = (S_1 + \begin{pmatrix} 0 & (S_2)^f & 0 \end{pmatrix})\boldsymbol{x},$$

so that $S_r = S_1 + \begin{pmatrix} 0 & (S_2)^f \& 0 \end{pmatrix}$.

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 in the compendium. This parallels Theorem 4.6 in the compendium.

a) Explain why, if $\boldsymbol{x} \in \mathbb{R}^{2N-2}$ is a symmetric extension (according to Definition 4.1 in the compendium), then $(\widehat{\boldsymbol{x}})_n = z_n e^{-\pi i n}$, where $\boldsymbol{z}$ is a real vectors which satisfies $z_n = z_{2N-2-n}$

Solution. Using Theorem 4.3 in the compendium with $d = N-1$ and with $2N-2$ for N, we obtain that

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

where $\boldsymbol{z}$ is a real vectors which satisfies $z_n = z_{2N-2-n}$.

b) Show that

$$\left\{ \boldsymbol{e}_0, \left\{ \frac{1}{\sqrt{2}}(\boldsymbol{e}_i + \boldsymbol{e}_{2N-2-i}) \right\}_{n=1}^{N-2}, \boldsymbol{e}_{N-1} \right\} \tag{5.7}$$

is an orthonormal basis for the vectors on the form $\widehat{\boldsymbol{x}}$ with $\boldsymbol{x} \in \mathbb{R}^{2N-2}$ a symmetric extension.

Solution. Clearly these vectors are an orthonormal basis for the set of vectors where $z_n = z_{2N-2-n}$. The vectors from a) are obtained by multiplying these with $e^{-\pi i n}$. But the orthonormality of these vectors are not affected when we multiply with $e^{-\pi i n}$, so we may skip this.

c) Show that

$$
\begin{gathered}
\frac{1}{\sqrt{2N-2}}\cos\left(2\pi\frac{0}{2N-2}k\right)\\
\left\{\frac{1}{\sqrt{N-1}}\cos\left(2\pi\frac{n}{2N-2}k\right)\right\}_{n=1}^{N-2}\\
\frac{1}{\sqrt{2N-2}}\cos\left(2\pi\frac{N-1}{2N-2}k\right)
\end{gathered}
\tag{5.8}
$$

is an orthonormal basis for the symmetric extensions in $\mathbb{R}^{2N-2}$.

Solution. We compute the IDFT for all vectors in (b). Since the IDFT is unitary, this will give us an orthonormal basis for the symmetric vectors in $\mathbb{R}^{2N-2}$. Since $(F_N)^H\boldsymbol{\phi}_n = \boldsymbol{e}_n$ we get that

$$
\begin{aligned}
(F_N)^H\boldsymbol{e}_0 = \boldsymbol{\phi}_0 &= \frac{1}{\sqrt{2N-2}}\cos\left(2\pi\frac{0}{2N-2}k\right)\\
(F_N)^H\left(\frac{1}{\sqrt{2}}(\boldsymbol{e}_n+\boldsymbol{e}_{2N-2-n})\right) &= \frac{1}{\sqrt{2}}\left(\boldsymbol{\phi}_n+\boldsymbol{\phi}_{2N-2-n}\right)\\
&= \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2N-2}}\left(e^{2\pi ikn/(2N-2)}+e^{-2\pi ikn/(2N-2)}\right)\\
&= \frac{1}{\sqrt{N-1}}\cos\left(2\pi\frac{n}{2N-2}k\right)\\
(F_N)^H\boldsymbol{e}_{N-1} = \boldsymbol{\phi}_{N-1} &= \frac{1}{\sqrt{2N-2}}\cos\left(2\pi\frac{N-1}{2N-2}k\right).
\end{aligned}
$$

These coincide with the vectors listed in the exercise.

d) Assume that S is symmetric. Show that the vectors listed in (5.8) 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.

Solution. Since S is symmetric, it preserves vectors which are symmetric around $N-1$. In the frequency domain, applying S to a vector listed in (5.8) in the compendium corresponds to multiplying the vectors listed in Equation (5.7) in the compendium with the frequency response. Since this does not introduce any more components, it is clear that the new vector must be a multiplum of the same vector, so that these vectors indeed are eigenvectors. But then the vectors restricted to $\mathbb{R}^N$ are also eigenvectors for S_r, since this is simply S when viewed on the first N elements. Since the vectors in $\mathbb{R}^{2N-2}$ are linearly independent, it is imemdiate that the corresponding vectors in $\mathbb{R}^N$ also are linearly independent, since the second part of the vectors mirror the first part.

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

$$\left\{\cos\left(2\pi\frac{n}{2N-2}k\right)\right\}_{n=0}^{N-1} \tag{5.9}$$

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

$$\sum_{k=0}^{2N-3}\cos\left(2\pi\frac{n_1}{2N-2}k\right)\cos\left(2\pi\frac{n_2}{2N-2}k\right) = (N-1)\times\begin{cases}2 & \text{if } n_1=n_2\in\{0,N-1\}\\ 1 & \text{if } n_1=n_2\notin\{0,N-1\}\\ 0 & \text{if } n_1\neq n_2\end{cases}. \tag{5.10}$$

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 $2N-2-k$.

Solution. Using that $\cos x = \cos(2\pi - x)$ we can here pair the summands k and $2N-2-k$ to obtain

$$\begin{aligned}
&\sum_{k=0}^{2N-3} \cos\left(2\pi\frac{n_1}{2N-2}k\right)\cos\left(2\pi\frac{n_2}{2N-2}k\right) \\
=&\cos\left(2\pi\frac{n_1}{2N-2}\cdot 0\right)\cos\left(2\pi\frac{n_2}{2N-2}\cdot 0\right) \\
&+2\sum_{k=1}^{N-2}\cos\left(2\pi\frac{n_1}{2N-2}k\right)\cos\left(2\pi\frac{n_2}{2N-2}k\right) \\
&+\cos\left(2\pi\frac{n_1}{2N-2}(N-1)\right)\cos\left(2\pi\frac{n_2}{2N-2}(N-1)\right).
\end{aligned}$$

If we divide by 2 and combine these equations we get the result.

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 in the compendium. 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

$$\sqrt{\frac{2}{N-1}}\begin{pmatrix} 1/\sqrt{2} & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1/\sqrt{2} \end{pmatrix}\left(\cos\left(2\pi\frac{n}{2N-2}k\right)\right)\begin{pmatrix} 1/\sqrt{2} & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1/\sqrt{2} \end{pmatrix}$$

is orthogonal.

c) Explain from b. that $\left(\cos\left(2\pi\frac{n}{2N-2}k\right)\right)^{-1}$ can be written as

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

With the expression we found in c., S_r can now be diagonalized as

$$\left(\cos\left(2\pi\frac{n}{2N-2}k\right)\right)D\left(\cos\left(2\pi\frac{n}{2N-2}k\right)\right)^{-1}.$$

Chapter 6

The filter representation of wavelets

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.

Solution. You can set for instance $H_0 = \{1/4, \underline{1/2}, 1/4\}$, and $H_1 = \{\underline{1}\}$ (when you write down the corresponding matrix you will see that $A_{0,1} = 1/2$, $A_{1,0} = 0$, so that the matrix is not symmetric)

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.

Solution. The Haar wavelet is 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?

Solution. We have that $H_0 = \frac{1}{5}\{1, 1, \underline{1}, 1, 1\}$, and $H_1 = \frac{1}{3}\{-1, \underline{1}, -1\}$. The frequency responses are

$$\begin{aligned} \lambda_{H_0}(\omega) &= \frac{1}{5}e^{2i\omega} + \frac{1}{5}e^{i\omega} + \frac{1}{5} + \frac{1}{5}e^{-i\omega}\frac{1}{5}e^{-2i\omega} \\ &= \frac{2}{5}\cos(2\omega) + \frac{2}{5}\cos\omega + \frac{1}{5} \\ \lambda_{H_1}(\omega) &= -\frac{1}{3}e^{i\omega} + \frac{1}{3} - \frac{1}{3}e^{-i\omega} = -\frac{2}{3}\cos\omega + \frac{1}{3}. \end{aligned}$$

Both filters are 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?

Solution. We have that $G_0 = \{1/4, \underline{1/2}, 1/4\}$, and $G_1 = \{1/16, -1/4, \underline{3/8}, -1/4, 1/16\}$. The frequency responses are

$$\begin{aligned}
\lambda_{G_0}(\omega) &= \frac{1}{4}e^{i\omega} + \frac{1}{2} + \frac{1}{4}e^{-i\omega} \\
&= \frac{1}{2}\cos(\omega) + \frac{1}{2} \\
\lambda_{G_1}(\omega) &= \frac{1}{16}e^{2i\omega} - \frac{1}{4}e^{i\omega} + \frac{3}{8} - \frac{1}{4}e^{-i\omega}\frac{1}{16}e^{-2i\omega} \\
&= \frac{1}{8}\cos(2\omega) - \frac{1}{2}\cos\omega + \frac{3}{8}.
\end{aligned}$$

Both filters are 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 $P_{\mathcal{C}_1 \leftarrow \phi_1}$ for the wavelet corresponding to these filters.

Solution. The frequency responses are

$$\begin{aligned}
\lambda_{H_0}(\omega) &= \frac{1}{16}e^{2i\omega} + \frac{1}{4}e^{i\omega} + \frac{3}{8} + \frac{1}{4}e^{-i\omega}\frac{1}{16}e^{-2i\omega} \\
&= \frac{1}{8}\cos(2\omega) + \frac{1}{2}\cos\omega + \frac{3}{8} \\
\lambda_{H_1}(\omega) &= -\frac{1}{4}e^{i\omega} + \frac{1}{2} - \frac{1}{4}e^{-i\omega} \\
&= -\frac{1}{2}\cos(\omega) + \frac{1}{2}.
\end{aligned}$$

The two first rows in $P_{\mathcal{C}_1 \leftarrow \phi_1}$ are

$$\begin{pmatrix} 3/8 & 1/4 & 1/16 & 0 & \cdots & 1/16 & 1/4 \\ -1/4 & 1/2 & -1/4 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

The remaining rows are obtained by translating these in alternating order.

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 $P_{\phi_1 \leftarrow \mathcal{C}_1}$ for the wavelet corresponding to these filters.

Solution. The frequency responses are

$$\begin{aligned}\lambda_{G_0}(\omega) &= \frac{1}{3}e^{i\omega} + \frac{1}{3} + \frac{1}{3}e^{-i\omega} = \frac{2}{3}\cos\omega + \frac{1}{3} \\ \lambda_{G_1}(\omega) &= \frac{1}{5}e^{2i\omega} - \frac{1}{5}e^{i\omega} + \frac{1}{5} - \frac{1}{5}e^{-i\omega}\frac{1}{5}e^{-2i\omega} \\ &= \frac{2}{5}\cos(2\omega) - \frac{2}{5}\cos\omega + \frac{1}{5}\end{aligned}$$

The two first columns in $P_{\phi_1 \leftarrow \mathcal{C}_1}$ are

$$\begin{pmatrix} 1/3 & -1/5 \\ 1/3 & 1/5 \\ 0 & -1/5 \\ 0 & 1/5 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 1/3 & 1/5 \end{pmatrix}$$

The remaining columns are obtained by translating these in alternating order.

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.

Solution. `c` and `w` represent the coordinates in the wavelet bases $\boldsymbol{\phi}_0$ and $\boldsymbol{\psi}_0$. The code runs a Haar wavelet transform. The sound is normalized so that the sound samples lie in the range between -1 and 1, and the resulting sound is played. The sound is split into two parts, and `c` represents a low-resolution version of the sound (with half the number of samples), so that we first will hear the sound played at double pace. After this we will hear the detail `w` in the sound, also played at double pace. We should also be able to recognize the sound from this detail.

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?

Solution. This corresponds to reconstructing a low-resolution approximation of the sound.

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.

Solution. The code which can be used looks like this:

```
g1 = array([-coeffs[0],-coeffs[1],-coeffs[3],0,0,0,0,-coeffs[2],\
            1, 0, 0, 0, 0, 0, 0, 0])
IDWTImpl(g1, 1, IDWTKernelpwl0, 0)
g1 = hstack([g1[13:16],g1[0:6]]) # Compact filter notation
```

Note that we have used a kernel which does not make symmetric extensions.

b) Plot the frequency response of G_1.

Solution. The code can look as follows:

```
omega = linspace(0,2*pi,100)
plt.plot(omega, g1[4] + g1[5]*2*cos(omega) + g1[6]*2*cos(2*omega) \
           + g1[7]*2*cos(3*omega) + g1[8]*2*cos(4*omega))
```

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 in the compendium).

Solution. The code can look like this:

```
def filterS(t, x, symm):
    tlen = len(t)
    N0 = (tlen - 1)/2
    N = shape(x)[0]

    if symm:
        y = concatenate([ x[N0:0:(-1)], x, x[(N-2):(N - N0 - 2):(-1)] ])
    else:
        y = concatenate([ x[(N - N0):], x, x[:N0]])
    if ndim(x) == 1:
        res = convolve(t, y)
        x[:] = res[(2*N0):(len(res)-2*N0)]
    else:
        n = shape(x)[1]
        for k in range(n):
            res = convolve(t, y[:, k])
            x[:, k] = res[(2*N0):(len(res)-2*N0)]
```

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 5.42 in the compendium and 6.5 in the compendium, 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.

Solution. The code can look like this:

```
def DWTKernelFilters(H0, H1, G0, G1, x, symm, dual):
    f0, f1 = H0, H1
    if dual:
        f0, f1 = G0, G1
    N = len(x)
```

```
    x0 = x.copy()
    x1 = x.copy()
    filterS(f0, x0, symm)
    filterS(f1, x1, symm)
    x[::2] = x0[::2]
    x[1::2] = x1[1::2]
```

```
def IDWTKernelFilters(H0, H1, G0, G1, x, symm, dual):
    f0, f1 = G0, G1
    if dual:
        f0, f1 = H0, H1
    N = len(x)
    x0 = x.copy(); x0[1::2] = 0
    x1 = x.copy(); x1[::2] = 0
    filterS(f0, x0, symm)
    filterS(f1, x1, symm)
    x[:] = x0 + x1
```

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)
```

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.

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) in the compendium 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.

Solution. The code can look as follows:

```
def mp3forwardfbt(x):
    N = len(x)
    z = mat(zeros((N,1)))
    C = mp3ctable() # The analysis window;
    x = x[(N-1)::(-1)]
    x = concatenate([x, zeros(512 - 32)])

    # The 32x64 matrix M
    yvec = arange(0, 32, 1, float)
    yvec = yvec.reshape((32, 1))
    xvec = arange(-16, 48, 1, float)
    xvec = xvec.reshape((1, 64))
    M = cos((2*yvec+1)*xvec*pi/64)

    start = len(x) - 512;
    for n in range(1, N/32 + 1):
        X = x[start:(start + 512)]
        Z = C*X # Pointwise multiplication
        Y = zeros(64)
        for j in range(8):
            Y += Z[(64*j):(64*(j+1))]
        Y = Y.reshape((64, 1))
        z[((n-1)*32):(n*32), 0] = mat(M)*mat(Y)
        start -= 32
    z = array(z).flatten()
    return z
```

```
def mp3reversefbt(z):
    N = len(z)
    z = z. reshape((N,1))
    z = mat(z)
    Ns = N/32
    x = zeros(32*Ns)
    D = mp3dtable()          # The reconstruction window.
    V = mat(zeros((1024,1)))
    # The 64x32 matrix N

    yvec = arange(16, 80, 1, float)
```

```
    yvec = yvec.reshape((64, 1))
    xvec = arange(0, 32, 1, float)
    xvec = xvec.reshape((1, 32))
    Nmatr = mat(cos(yvec*(2*xvec + 1)*pi/64))

    U = zeros(512)
    for n in range(1, Ns+1):
        V[64:1024, 0] = V[0:(1024-64), 0]
        V[0:64, 0] = Nmatr*z[((n-1)*32):(n*32), 0]

        for i in range(8):
            U[(i*64):(i*64 + 32)]       = \
            array(V[(i*128):(i*128 + 32), 0]).flatten()
            U[(i*64 + 32):((i + 1)*64)] = \
            array(V[(i*128 + 96):((i+1)*128), 0]).flatten()

        W = U*D
        for i in range(16):
            x[((n-1)*32):(n*32)] += W[32*i:(32*(i + 1))]
    return x
```

Chapter 7

Constructing interesting wavelets

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.

Solution. The code can look as follows:

```
plotwaveletfunctions(IDWTKernelHaar, -2, 6)
plotwaveletfunctions(IDWTKernelpwl0, -2, 6)
plotwaveletfunctions(IDWTKernelpwl2, -2, 6)
```

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).

Solution. Assuming that the vector `coeffs` has been set as in Exercise 6.10, the code can look as follows

```
m = 10
t = linspace(-2, 6, 8*2**m)
coordsvm = hstack([[-coeffs[0], -coeffs[1], -coeffs[3], 0, 0, 0, 0,\
                     -coeffs[2], 1, 0, 0, 0, 0, 0, 0, 0], \
                    zeros(8*2**m-16)])
IDWTImpl(coordsvm, m, IDWTKernelpwl0, 0)
coordsvm *= 2**(m/2.)
plt.plot(t,hstack([coordsvm[(6*2**m):(8*2**m+1)], coordsvm[0:(6*2**m)]]))
```

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.

Solution. Assume that the kernel transformations of the DWT and the IDWT are H and G, respectively. The formulas for the DWT and the dual DWT are obvious. For the transpose the point is that, while the kernel transformations of the DWT and the dual DWT are H and G^T, we compose the kernel with a permutation matrix when we compute the DWT. When we transpose, the order of the kernel and the permutation changes, so the transpose must use an IDWT implementation instead.
The kernel for the transpose of the DWT is H^T, which is the kernel of the dual IDWT. This explains the third line.
The kernel for the transpose of the dual DWT is $(G^T)^T = G$, which is the kernel of the IDWT. This explains the fourth line.

Exercise 7.4: Compute filters

Compute the filters H_0, G_0 in Theorem 7.12 in the compendium 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).

Solution. We have that

$$\lambda_{H_0}(\omega) = \left(\frac{1}{2}(1+\cos\omega)\right)^{N_1/2} Q_1\left(\frac{1}{2}(1-\cos\omega)\right) = \left(\frac{1}{2}(1+\cos\omega)\right)^{2} Q^{(4)}\left(\frac{1}{2}(1-\cos\omega)\right)$$

$$\begin{aligned}\lambda_{G_0}(\omega) &= \left(\frac{1}{2}(1+\cos\omega)\right)^{N_2/2} Q_2\left(\frac{1}{2}(1-\cos\omega)\right) = \left(\frac{1}{2}(1+\cos\omega)\right)^{2} \\ &= \frac{1}{4}\left(1+\frac{1}{2}e^{i\omega}+\frac{1}{2}e^{-i\omega}\right)^2 = \frac{1}{16}(e^{2i\omega}+4e^{i\omega}+6+4e^{-i\omega}+e^{-2i\omega}).\end{aligned}$$

Therefore $G_0 = \frac{1}{16}\{1,4,\underline{6},4,1\}$. We do not recommend to compute H_0 by hand. With the package `sympy` in Python you can do as follows to compute H_0.

```
x = Symbol('x')
z = expand( ((1+x/2+1/(2*x))/2)**2* \
        (2+8*((1-x/2-1/(2*x))/2)+20*((1-x/2-1/(2*x))/2)**2\
        +40*((1-x/2-1/(2*x))/2)**3) )
```

Here we have substituted `x` for $e^{i\omega}$, `1/x` for $e^{-i\omega}$. The first part represents $\left(\frac{1}{2}(1+\cos\omega)\right)^2$, the second part represents $Q^{(4)}(u) = 2+8u+20u^2+40u^3$ with $u = \frac{1}{2}(1-\cos\omega) = \frac{1}{2}\left(1-\frac{1}{2}e^{i\omega}-\frac{1}{2}e^{-i\omega}\right)$. This gives

$$H_0 = \frac{1}{128}\{-5,20,-1,-96,70,\underline{280},70,-96,-1,20,-5\}.$$

Using Theorem 6.16 in the compendium with $\alpha=1$, $d=0$, we get

$$\begin{aligned}H_1 &= \frac{1}{16}\{1,-4,\underline{6},-4,1\} \\ G_1 &= \frac{1}{128}\{5,20,1,-96,-70,\underline{280},-70,-96,1,20,5\}\end{aligned}$$

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.

Exercise 7.7: Generate plots

Generate the plots from Figure 7.3 in the compendium using the cascade algorithm. Reuse the code from Exercise 7.1 in order to achieve this.

Chapter 8

The polyphase representation and wavelets

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

$$\begin{aligned}
\lambda_{H_0}(\omega) &= \lambda_{H^{(0,0)}}(2\omega) + e^{i\omega}\lambda_{H^{(0,1)}}(2\omega)\\
\lambda_{H_1}(\omega) &= \lambda_{H^{(1,1)}}(2\omega) + e^{-i\omega}\lambda_{H^{(1,0)}}(2\omega)\\
\lambda_{G_0}(\omega) &= \lambda_{G^{(0,0)}}(2\omega) + e^{-i\omega}\lambda_{G^{(1,0)}}(2\omega)\\
\lambda_{G_1}(\omega) &= \lambda_{G^{(1,1)}}(2\omega) + e^{i\omega}\lambda_{G^{(0,1)}}(2\omega).
\end{aligned}$$

Solution. $G^{(0,0)}, G^{(1,1)}$ are the even-indexed filter coefficients of G_0, G_1, respectively, so that $\lambda_{G^{(0,0)}}(2\omega), \lambda_{G^{(1,1)}}(2\omega)$ represents the half of $\lambda_{G_0}(\omega), \lambda_{G_1}(\omega)$, respectively, from the even filter coefficients. $G^{(1,0)}$ are the odd-indexed filter coefficients of G_0. Since coefficient 0 in $G^{(1,0)}$ equals coefficient 1 in G_0, it is clear that $e^{-i\omega}\lambda_{G^{(1,0)}}(2\omega)$ represents the half of $\lambda_{G_0}(\omega)$ from the odd filter coefficients. This proves the first formula. The second formula follows from the same kind of reasoning.

If we transpose H (also in polyphase form), we get an MRA-matrix where the columns are given by the filters $(H_0)^T, (H_1)^T$. Inserting these in the formulas we just proved we get that

$$\begin{aligned}
\lambda_{(H_0)^T}(\omega) &= \lambda_{(H^{(0,0)})^T}(2\omega) + e^{-i\omega}\lambda_{(H^{(0,1)})^T}(2\omega)\\
\lambda_{(H_1)^T}(\omega) &= \lambda_{(H^{(1,1)})^T}(2\omega) + e^{i\omega}\lambda_{(H^{(1,0)})^T}(2\omega).
\end{aligned}$$

If we conjugate these expressions we get

$$\begin{aligned}
\lambda_{H_0}(\omega) &= \lambda_{H^{(0,0)}}(2\omega) + e^{i\omega}\lambda_{H^{(0,1)}}(2\omega)\\
\lambda_{H_1}(\omega) &= \lambda_{H^{(1,1)}}(2\omega) + e^{-i\omega}\lambda_{H^{(1,0)}}(2\omega),
\end{aligned}$$

and the proof is done.

b) In the proof of the last part of Theorem 6.17 in the compendium, we defered the last part, namely that equations 6.17 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).

Solution. The first column in the matrix on the left hand side gives the filter G_0. On the right hand side, a) states that the even-indexed columns are taken from the filter with frequency response

$$\begin{aligned}
&\lambda_{\alpha E_{-d} H^{(1,1)}}(2\omega) + e^{-i\omega}\lambda_{-\alpha E_{-d} H^{(1,0)}}(2\omega)\\
&= \alpha\lambda_{E_{-d}}(2\omega)\left(\lambda_{H^{(1,1)}}(2\omega) - e^{-i\omega}\lambda_{H^{(1,0)}}(2\omega)\right)\\
&= \alpha e^{2id\omega}\left(\lambda_{H^{(1,1)}}(2(\omega+\pi)) + e^{-i(\omega+\pi)}\lambda_{H^{(1,0)}}(2(\omega+\pi))\right) = \alpha e^{2id\omega}\lambda_{H_1}(\omega+\pi).
\end{aligned}$$

This shows that $\lambda_{G_0}(\omega) = \alpha e^{2id\omega}\lambda_{H_1}(\omega+\pi)$. We obtain Equation (8.2) in the compendium easily from this. Now, the second column in the matrix on the left hand side gives the filter coefficients of G_1. On the right hand side, a) states that the odd-indexed columns are taken from the filter with frequency response

$$\begin{aligned}
&\lambda_{\alpha E_{-d} H^{(0,0)}}(2\omega) + e^{i\omega}\lambda_{-\alpha E_{-d} H^{(0,1)}}(2\omega)\\
&= \alpha\lambda_{E_{-d}}(2\omega)\left(\lambda_{H^{(0,0)}}(2\omega) - e^{i\omega}\lambda_{H^{(0,1)}}(2\omega)\right)\\
&= \alpha e^{2id\omega}\left(\lambda_{H^{(0,0)}}(2(\omega+\pi)) + e^{i(\omega+\pi)}\lambda_{H^{(0,1)}}(2(\omega+\pi))\right) = \alpha e^{2id\omega}\lambda_{H_0}(\omega+\pi).
\end{aligned}$$

This shows that $\lambda_{G_1}(\omega) = \alpha e^{2id\omega}\lambda_{H_0}(\omega+\pi)$, which is Equation (8.3) in the compendium.

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

$$\lambda_{\tilde{G_0}}(\omega) = \lambda_{G_0}(\omega) + \lambda_S(2\omega)e^{-i\omega}\lambda_{G_1}(\omega),$$

Solution. We have that

$$\begin{pmatrix} G^{(0,0)} & G^{(0,1)} \\ G^{(1,0)} & G^{(1,1)} \end{pmatrix} \begin{pmatrix} I & \mathbf{0} \\ S & I \end{pmatrix} = \begin{pmatrix} G^{(0,0)} + SG^{(0,1)} & G^{(0,1)} \\ G^{(1,0)} + SG^{(1,1)} & G^{(1,1)} \end{pmatrix}.$$

Using Exercise 8.1a), the even-indexed columns in this matrix are taken from the filter with frequency response

$$\begin{aligned}
&\lambda_{G^{(0,0)}+SG^{(0,1)}}(2\omega) + e^{-i\omega}\lambda_{G^{(1,0)}+SG^{(1,1)}}(2\omega) \\
&= \lambda_{G^{(0,0)}}(2\omega) + e^{-i\omega}\lambda_{G^{(1,0)}}(2\omega) + \lambda_S(2\omega)\left(\lambda_{G^{(0,1)}}(2\omega) + e^{-i\omega}\lambda_{G^{(1,1)}}(2\omega)\right) \\
&= \lambda_{G_0}(\omega) + \lambda_S(2\omega)e^{-i\omega}\left(\lambda_{G^{(1,1)}}(2\omega) + e^{i\omega}\lambda_{G^{(0,1)}}(2\omega)\right) \\
&= \lambda_{G_0}(\omega) + \lambda_S(2\omega)e^{-i\omega}\lambda_{G_1}(\omega).
\end{aligned}$$

b)

$$G \begin{pmatrix} I & S \\ \mathbf{0} & I \end{pmatrix}$$

is an MRA matrix with filters $G_0, \tilde{G_1}$, where

$$\lambda_{\tilde{G_1}}(\omega) = \lambda_{G_1}(\omega) + \lambda_S(2\omega)e^{i\omega}\lambda_{G_0}(\omega),$$

Solution. We have that

$$\begin{pmatrix} G^{(0,0)} & G^{(0,1)} \\ G^{(1,0)} & G^{(1,1)} \end{pmatrix} \begin{pmatrix} I & S \\ \mathbf{0} & I \end{pmatrix} = \begin{pmatrix} G^{(0,0)} & SG^{(0,0)} + G^{(0,1)} \\ G^{(1,0)} & SG^{(1,0)} + G^{(1,1)} \end{pmatrix},$$

so that the odd-indexed columns in this matrix are taken from the filter with frequency response

$$\begin{aligned}
&\lambda_{SG^{(1,0)}+G^{(1,1)}}(2\omega) + e^{i\omega}\lambda_{SG^{(0,0)}+G^{(0,1)}}(2\omega) \\
&= \lambda_{G^{(1,1)}}(2\omega) + e^{i\omega}\lambda_{G^{(0,1)}}(2\omega) + \lambda_S(2\omega)\left(\lambda_{G^{(1,0)}}(2\omega) + e^{i\omega}\lambda_{G^{(0,0)}}(2\omega)\right) \\
&= \lambda_{G_1}(\omega) + \lambda_S(2\omega)e^{i\omega}\left(\lambda_{G^{(0,0)}}(2\omega) + e^{-i\omega}\lambda_{G^{(1,0)}}(2\omega)\right) \\
&= \lambda_{G_1}(\omega) + \lambda_S(2\omega)e^{i\omega}\lambda_{G_0}(\omega).
\end{aligned}$$

c)

$$\begin{pmatrix} I & \mathbf{0} \\ S & I \end{pmatrix} H$$

is an MRA-matrix with filters $H_0, \tilde{H}_1$, where

$$\lambda_{\tilde{H}_1}(\omega) = \lambda_{H_1}(\omega) + \lambda_S(2\omega)e^{-i\omega}\lambda_{H_0}(\omega).$$

Solution. We transpose the expression to obtain $H^T \begin{pmatrix} I\&S^T & \\ \mathbf{0} & I \end{pmatrix}$ Since H^T has filters $(H_0)^T$ and $(H_1)^T$ in the columns, from b. it follows that $H^T \begin{pmatrix} I\&S^T & \\ \mathbf{0} & I \end{pmatrix}$ has columns given by $(H_0)^T$ and the filter with frequency response

$$\lambda_{(H_1)^T}(\omega) + \lambda_{S^T}(2\omega)e^{i\omega}\lambda_{(H_0)^T}(\omega) = \overline{\lambda_{H_1}(\omega) + \lambda_S(2\omega)e^{-i\omega}\lambda_{H_0}(\omega)},$$

so that $\begin{pmatrix} I & \mathbf{0} \\ S\&I & \end{pmatrix}$ H has row filters H_0 and a filter $\tilde{H}_1$ with frequency response

$$\lambda_{\tilde{H}_1}(\omega) = \lambda_{H_1}(\omega) + \lambda_S(2\omega)e^{-i\omega}\lambda_{H_0}(\omega).$$

d)

$$\begin{pmatrix} I & S \\ \mathbf{0} & I \end{pmatrix} H$$

is an MRA-matrix with filters $\tilde{H}_0, H_1$, where

$$\lambda_{\tilde{H}_0}(\omega) = \lambda_{H_0}(\omega) + \lambda_S(2\omega)e^{i\omega}\lambda_{H_1}(\omega).$$

Solution. We transpose the expression to obtain $H^T \begin{pmatrix} I & \mathbf{0} \\ S^T\&I & \end{pmatrix}$. Since H^T has filters $(H_0)^T$ and $(H_1)^T$ in the columns, using a) we see that $H^T \begin{pmatrix} I & \mathbf{0} \\ S^T\&I & \end{pmatrix}$ has columns given by the filter with frequency response

$$\lambda_{(H_0)^T}(\omega) + \lambda_{S^T}(2\omega)e^{-i\omega}\lambda_{(H_1)^T}(\omega) = \overline{\lambda_{H_0}(\omega) + \lambda_S(2\omega)e^{i\omega}\lambda_{H_1}(\omega)},$$

and $(H_1)^T$, so that $\begin{pmatrix} I\&S & \\ \mathbf{0} & I \end{pmatrix}$ H has a row filter $\tilde{H}_0$ with frequency response

$$\lambda_{\tilde{H}_0}(\omega) = \lambda_{H_0}(\omega) + \lambda_S(2\omega)e^{i\omega}\lambda_{H_1}(\omega),$$

and H_1.

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).

Solution. If S is a filter we have that $S^{(i+r+s_1M) \bmod kM,(j+r+s_2M)kM} = S^{(i+s_1M) \bmod M,(j+s_2M) \bmod M}$, $0 \leq i,j < M$. But since $S^{(i+s_1M) \bmod kM,(j+s_2M)kM} = S^{(i,j)}_{s_1,s_2}$, it follows that $S^{((i+r) \bmod M,(j+r) \bmod M)}_{s_1,s_2} = S^{(i,j)}_{s_1,s_2}$, so that $S^{(i+r) \bmod M,(j+r) \bmod M} = S_{i,j}$.

Exercise 8.4: QMF filter banks

Recall from Definition 6.20 in the compendium 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 in the compendium 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 $|\lambda_{H_0}(\omega)|^2 + |\lambda_{H_0}(\omega+\pi)|^2 = 2$.

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.

Solution. The code can look like this:

```
def liftingstepevensymm(lmbda, x, symm):
    """
    Apply an elementary symmetric lifting step of even type to x.

    lmbda: The common value of the two filter coefficients
    x: The vector which we apply the lifting step to
    symm: Whether to apply symmetric extension to the input
    """
    if (not symm) and mod(len(x), 2)!=0:
        raise AssertionError()
    if symm:
        x[0] += 2*lmbda*x[1] # With symmetric extension
    else:
        x[0] += lmbda*(x[1]+x[-1])
    x[2:-1:2] += lmbda*(x[1:-2:2] + x[3::2])
    if mod(len(x), 2)==1 and symm:
        x[-1] += 2*lmbda*x[-2] # With symmetric extension
```

```
def liftingstepoddsymm(lmbda, x, symm):
    """
    Apply an elementary symmetric lifting step of odd type to x.

    lmbda: The common value of the two filter coefficients
    x: The vector which we apply the lifting step to
    symm: Whether to apply symmetric extension to the input
    """
    if (not symm) and mod(len(x), 2)!=0:
```

```
        raise AssertionError()
    x[1:-1:2] += lmbda*(x[0:-2:2] + x[2::2])
    if mod(len(x), 2)==0:
        if symm:
            x[-1] += 2*lmbda*x[-2] # With symmetric extension
        else:
            x[-1] += lmbda*(x[0]+x[-2])
```

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`.

Solution. The code can look like this:

```
def DWTKernelpwl0(x, symm, dual):
    """
    Apply the DWT kernel transformation for the
    piecewise linear wavelet (i.e. 0 vanishing moments) to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        x /= sqrt(2)
        x = liftingstepevensymm(0.5, x, symm)
    else:
        x *= sqrt(2)
        liftingstepoddsymm(-0.5, x, symm)
```

```
def IDWTKernelpwl0(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the
    pieccewise linear wavelet (i.e. 0 vanishing moments) to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
```

```
    """
    if dual:
        x *= sqrt(2)
        liftingstepevensymm(-0.5, x, symm)
    else:
        x /= sqrt(2)
        liftingstepoddsymm(0.5, x, symm)
```

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`.

Solution. The code can look like this:

```
def DWTKernelpwl2(x, symm, dual):
    """
    Apply the DWT kernel transformation for the
    alternative pieccewise linear wavelet (i.e. 2 van. moms.) to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        liftingstepevensymm(0.5, x, symm)
        liftingstepoddsymm(-0.25, x, symm)
        x /= sqrt(2)
    else:
        liftingstepoddsymm(-0.5, x, symm)
        liftingstepevensymm(0.25, x, symm)
        x *= sqrt(2)
```

```
def IDWTKernelpwl2(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the
    alternative pieccewise linear wavelet (i.e. 2 van. moms.) to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        x *= sqrt(2)
        liftingstepoddsymm(0.25, x, symm)
        liftingstepevensymm(-0.5, x, symm)
    else:
        x /= sqrt(2)
        liftingstepevensymm(-0.25, x, symm)
        liftingstepoddsymm(0.5, x, symm)
```

c) Write the DWT and IDWT kernel transformations for the Spline 5/3 wavelet, using the lifting factorization obtained in Example 8.14 in the compendium. Call your functions `DWTKernel53` and `IDWTKernel53`.

Solution. The code can look like this:

```
def DWTKernel53(x, symm, dual):
    """
    Apply the DWT kernel transformation for the
    Spline 5/3 wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        x[0::2] *= 0.5
        x[1::2] *= 2
        liftingstepevensymm(0.125, x, symm)
        liftingstepoddsymm(-1, x, symm)
    else:
        x[0::2] *= 2
        x[1::2] *= 0.5
        liftingstepoddsymm(-0.125, x, symm)
        liftingstepevensymm(1, x, symm)
```

```
def IDWTKernel53(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the
    Spline 5/3 wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        liftingstepoddsymm(1, x, symm)
        liftingstepevensymm(-0.125, x, symm)
        x[0::2] *= 2
        x[1::2] *= 0.5
    else:
        liftingstepevensymm(-1, x, symm)
        liftingstepoddsymm(0.125, x, symm)
        x[0::2] *= 0.5
        x[1::2] *= 2
```

d) Write the DWT and IDWT kernel transformations for the CDF 9/7 wavelet, using the lifting factorization obtained in Example 8.15 in the compendium. Call your functions `DWTKernel97` and `IDWTKernel97`.

Solution. The code can look like this:

```
def DWTKernel97(x, symm, dual):
    """
    Apply the DWT kernel transformation for the CDF 9/7 wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
```

```
    lambda1=-0.586134342059950
    lambda2=-0.668067171029734
    lambda3=0.070018009414994
    lambda4=1.200171016244178
    alpha=-1.149604398860250
    beta=-0.869864451624777
    if dual:
        x[0::2] /= alpha
        x[1::2] /= beta
        liftingstepevensymm(lambda4, x, symm)
        liftingstepoddsymm(lambda3, x, symm)
        liftingstepevensymm(lambda2, x, symm)
        liftingstepoddsymm(lambda1, x, symm)
    else:
        x[0::2] *= alpha
        x[1::2] *= beta
        liftingstepoddsymm(-lambda4, x, symm)
        liftingstepevensymm(-lambda3, x, symm)
        liftingstepoddsymm(-lambda2, x, symm)
        liftingstepevensymm(-lambda1, x, symm)
```

```
def IDWTKernel97(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the CDF 9/7 wavelet to x

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    lambda1=-0.586134342059950
    lambda2=-0.668067171029734
    lambda3=0.070018009414994
    lambda4=1.200171016244178
    alpha=-1.149604398860250
    beta=-0.869864451624777
    if dual:
        liftingstepoddsymm(-lambda1, x, symm)
        liftingstepevensymm(-lambda2, x, symm)
        liftingstepoddsymm(-lambda3, x, symm)
        liftingstepevensymm(-lambda4, x, symm)
        x[0::2] *= alpha
        x[1::2] *= beta
    else:
        liftingstepevensymm(lambda1, x, symm)
        liftingstepoddsymm(lambda2, x, symm)
        liftingstepevensymm(lambda3, x, symm)
        liftingstepoddsymm(lambda4, x, symm)
        x[0::2] /= alpha
        x[1::2] /= beta
```

e) In Chapter 5 in the compendium, 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.

Solution. The following code can be used for listening to the low-resolution approximations for a given value of m.

```
playDWT(m, DWTKernel53, IDWTKernel53, True)
playDWT(m, DWTKernel97, IDWTKernel97, True)
```

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.

Solution. The code can look as follows.

```
plotwaveletfunctions(IDWTKernel53, -4, 4)
plotwaveletfunctions(IDWTKernel97, -4, 4)
```

In the plot for the CDF 9/7 wavelet, it is seen that the functions and their dual counterparts are close to being equal. This reflects the fact that this wavelet is close to being orthogonal.

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.

Solution. The code can look like this:

```
def liftingstepeven(lmbda1, lmbda2, x):
    """
    Apply an elementary non-symmetric lifting step of even type to x.

    lmbda1: The first filter coefficient
    lmbda2: The second filter coefficient
    x: The vector which we apply the lifting step to
    """
    if mod(len(x), 2)!=0:
        raise AssertionError()
    x[0] += lmbda1*x[1] + lmbda2*x[-1]
    x[2:-1:2] += lmbda1*x[3::2] + lmbda2*x[1:-2:2]
```

```
def liftingstepodd(lmbda1, lmbda2, x):
    """
    Apply an elementary non-symmetric lifting step of odd type to x.

    lmbda1: The first filter coefficient
    lmbda2: The second filter coefficient
    x: The vector which we apply the lifting step to
    """
    if mod(len(x), 2)!=0:
        raise AssertionError()
    x[1:-2:2] += lmbda1*x[2:-1:2] + lmbda2*x[0:-3:2]
    x[-1] += lmbda1*x[0] + lmbda2*x[-2]
```

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.

Solution. The code can look like this:

```
def DWTKernelOrtho( x, symm, dual):
    """
    Apply the DWT kernel transformation for orthonormal wavelets.
    The number of vanishing moments is stored in global variables.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    global lambdas, alpha, beta
    global beta
    if dual:
        x[0::2] /= alpha
        x[1::2] /= beta
        for stepnr in range(lambdas.shape[0] - 1, 0, -2):
            liftingstepodd(lambdas[stepnr, 1], lambdas[stepnr, 0], x)
            liftingstepeven(lambdas[stepnr -1, 1], lambdas[stepnr - 1, 0], x)
        if mod(lambdas.shape[0], 2)==1:
            liftingstepodd(lambdas[0, 1], lambdas[0, 0], x)
    else:
        x[0::2] *= alpha
        x[1::2] *= beta
        for stepnr in range(lambdas.shape[0] - 1, 0, -2):
            liftingstepeven(-lambdas[stepnr, 0], -lambdas[stepnr, 1], x)
            liftingstepodd(-lambdas[stepnr - 1, 0], -lambdas[stepnr - 1, 1], x)
        if mod(lambdas.shape[0], 2)==1:
            liftingstepeven(-lambdas[0, 0], -lambdas[0, 1], x)
```

```
def IDWTKernelOrtho( x, symm, dual):
    """
    Apply the IDWT kernel transformation for orthonormal wavelets.
    The number of vanishing moments is stored in global variables.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    global lambdas
    global alpha
    global beta
    if dual:
        stepnr = 0
        if mod(lambdas.shape[0], 2) == 1: # Start with an odd step
            liftingstepodd(-lambdas[stepnr, 1], -lambdas[stepnr, 0], x)
            stepnr += 1
```

```
        while stepnr < lambdas.shape[0]:
            liftingstepeven(-lambdas[stepnr, 1], -lambdas[stepnr, 0], x)
            liftingstepodd(-lambdas[stepnr + 1, 1], -lambdas[stepnr + 1, 0], x)
            stepnr += 2
        x[0::2] *= alpha
        x[1::2] *= beta
    else:
        stepnr = 0
        if mod(lambdas.shape[0],2) == 1: # Start with an even step
            liftingstepeven(lambdas[stepnr, 0], lambdas[stepnr, 1], x)
            stepnr += 1
        while stepnr < lambdas.shape[0]:
            liftingstepodd(lambdas[stepnr, 0], lambdas[stepnr, 1], x)
            liftingstepeven(lambdas[stepnr + 1, 0], lambdas[stepnr + 1, 1], x)
            stepnr += 2
        x[0::2] /= alpha
        x[1::2] /=beta
```

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.

Solution. The following code can be used for listening to the low-resolution approximations for a given value of m.

```
liftingfactortho(2)
playDWT(m, DWTKernelOrtho, IDWTKernelOrtho, True)

liftingfactortho(3)
playDWT(m, DWTKernelOrtho, IDWTKernelOrtho, True)

liftingfactortho(4)
playDWT(m, DWTKernelOrtho, IDWTKernelOrtho, True)
```

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.

Solution. The code can look as follows.

```
liftingfactortho(2)
plotwaveletfunctions(IDWTKernelOrtho, -4, 4)

liftingfactortho(3)
plotwaveletfunctions(IDWTKernelOrtho, -4, 4)

liftingfactortho(4)
plotwaveletfunctions(IDWTKernelOrtho, -4, 4)
```

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

$$\frac{1}{\sqrt{2}}\begin{pmatrix} I & \mathbf{0} \\ \{1/2, \underline{1/2}\} & I \end{pmatrix}\begin{pmatrix} I & \{-\gamma, \underline{-\alpha}, -\beta, -\delta\} \\ \mathbf{0} & I \end{pmatrix}. \tag{8.1}$$

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.

Solution. We have found constants $\alpha, \beta\gamma\delta$ so that

$$[\hat{\psi}]_{(\boldsymbol{\phi}_0, \boldsymbol{\psi}_0)} = (-\alpha, -\beta, -\delta, 0, 0, 0, 0, -\gamma) \oplus (1, 0, 0, 0, 0, 0, 0, 0),$$

From this it is clear that

$$P_{(\boldsymbol{\phi}_1, \boldsymbol{\psi}_1) \leftarrow (\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)} = \begin{pmatrix} I & S_2 \\ \mathbf{0} & I \end{pmatrix}$$

where $S_2 = \{-\gamma, \underline{-\alpha}, -\beta, -\delta\}$ This gives as before the lifting factorization

$$P_{\mathcal{D}_1 \leftarrow (\boldsymbol{\phi}_1, \hat{\boldsymbol{\psi}}_1)} = \frac{1}{\sqrt{2}}\begin{pmatrix} I & \mathbf{0} \\ S_1 & I \end{pmatrix}\begin{pmatrix} I & \{-\gamma, \underline{-\alpha}, -\beta, -\delta\} \\ \mathbf{0} & I \end{pmatrix}. \tag{8.2}$$

where $S_1 = \{1/2, \underline{1/2}\}$ as before.

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.

Solution. The polyphase factorization of the IDWT is

$$\begin{pmatrix} \frac{1}{16}\{1, \underline{6}, 1\} & \frac{1}{128}\{5, 1, \underline{-70}, -70, 1, 5\} \\ \frac{1}{16}\{4, \underline{4}\} & \frac{1}{128}\{20, -96, \underline{280}, -96, 20\} \end{pmatrix}.$$

We can first apply an even lifting step:

$$\begin{pmatrix} I & -\frac{1}{4}\{\underline{1}, 1\} \\ \mathbf{0} & I \end{pmatrix} \begin{pmatrix} \frac{1}{16}\{1, \underline{6}, 1\} & \frac{1}{128}\{5, 1, \underline{-70}, -70, 1, 5\} \\ \frac{1}{16}\{4, \underline{4}\} & \frac{1}{128}\{20, -96, \underline{280}, -96, 20\} \end{pmatrix} = \begin{pmatrix} \frac{1}{16}\{\underline{4}\} & \frac{1}{128}\{20, \underline{-116}, -116, 20\} \\ \frac{1}{16}\{4, \underline{4}\} & \frac{1}{128}\{20, -96, \underline{280}, -96, 20\} \end{pmatrix}.$$

We can now apply an odd lifting step

$$\begin{pmatrix} I & \mathbf{0} \\ -\{1, \underline{1}\} & I \end{pmatrix} \begin{pmatrix} \frac{1}{16}\{\underline{4}\} & \frac{1}{128}\{20, \underline{-116}, -116, 20\} \\ \frac{1}{16}\{4, \underline{4}\} & \frac{1}{128}\{20, -96, \underline{280}, -96, 20\} \end{pmatrix} = \begin{pmatrix} \frac{1}{4} & \frac{1}{128}\{20, \underline{-116}, -116, 20\} \\ \mathbf{0} & 4 \end{pmatrix}$$

Since

$$\begin{pmatrix} I & -\frac{1}{512}\{20, \underline{-116}, -116, 20\} \\ \mathbf{0} & I \end{pmatrix} \begin{pmatrix} \frac{1}{4} & \frac{1}{128}\{20, \underline{-116}, -116, 20\} \\ \mathbf{0} & 4 \end{pmatrix} = \begin{pmatrix} \frac{1}{4} & \mathbf{0} \\ \mathbf{0} & 4 \end{pmatrix},$$

it follows that

$$\begin{pmatrix} I & -\frac{1}{128}\{5, \underline{-29}, -29, 5\} \\ \mathbf{0} & I \end{pmatrix} \begin{pmatrix} I & \mathbf{0} \\ -\{1, \underline{1}\} & I \end{pmatrix} \begin{pmatrix} I & -\frac{1}{4}\{\underline{1}, 1\} \\ \mathbf{0} & I \end{pmatrix} G = \begin{pmatrix} \frac{1}{4} & \mathbf{0} \\ \mathbf{0} & 4 \end{pmatrix}.$$

b) Implement the kernels of the wavelet of this exercise using what you did in Exercise 6.12.

```
H0 = array([-5, 20, -1, -96, 70, 280, 70, -96, -1, 20, -5])/128.
H1 = array([1, -4, 6, -4, 1])/16.
G0 = array([1, 4, 6, 4, 1])/16.
G1 = array([5, 20, 1, -96, -70, 280, -70, -96, 1, 20, 5])/128.
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)
```

Solution.

c) Listen to the low-resolution approximations and detail components in sound for this wavelet.

Solution. The following code can be used for listening to the low-resolution approximations for a given value of m.

```
playDWT(m, f, invf, True)
```

d) Use the function `plotwaveletfunctions` from Exercise 7.1 to plot all scaling functions and mother wavelets for this wavelet.

Solution. The code can look as follows.

```
plotwaveletfunctions(invf, -4, 4)
```

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 in the compendium. Do you get the exact same result back?

Solution. The following code can be used:

```
x = arange(1,8193)
mp3reversefbt(mp3forwardfbt(x))
plt.plot(x)
```

There are some small errors from the original vector in the resulting vector, when one compensates for the delay of 481 elements.

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 in the compendium that if we define the synthesis prototype filter as therein, and set $c = 1, d = -15$, we have that $GH = 16E_{481-32\cdot 15} = 16E_1$.

Chapter 9

Digital images

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) in the compendium.

Solution. The code can be found in the notebook for generating the figures in this chapter.

Exercise 9.2: Adjust contrast in images 1

Generate the right image in Figure 9.9 in the compendium on your own by writing code which uses the function `contrastadjust`.

Solution. The code can be found in the notebook for generating the figures in this chapter.

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.

Solution. The code could look as follows:

```
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]
```

b) Generate the left image in Figure 9.9 in the compendium on your own by using your code from Exercise 9.2, and instead calling the function `contrastadjust0`.

Solution. The code can be found in the notebook for generating the figures in this chapter.

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 9.1, 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.

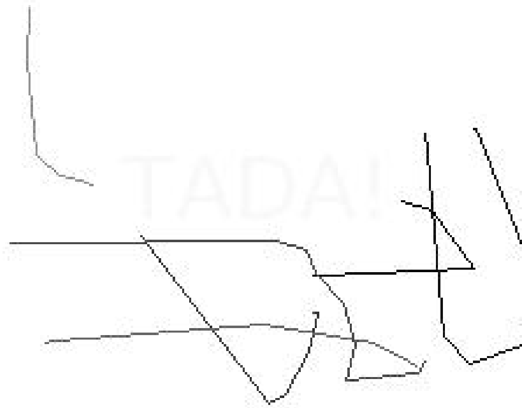

Figure 9.1: Secret message.

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.

Solution. The secret message is revealed in Figure 9.2.

Figure 9.2: Secret message revealed!

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`.

Solution. The following code can be used:

```
def tensor_impl(X, S1, S2):
    M, N = shape(X)[0:2]
    for n in range(N):
        S1(X[:,n])
    for m in range(M):
        S2(X[m, :])
```

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 in the compendium.

b) The right image in Figure 9.13 in the compendium.

c) The images in figures 9.14 in the compendium.

d) The images in Figure 9.15 in the compendium.

e) The images in Figure 9.16 in the compendium.

Solution. The code can be found in the notebook for generating the figures in this chapter.

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 $F(\boldsymbol{x}, \alpha \boldsymbol{y}_1 + \beta \boldsymbol{y}_2) = \alpha F(\boldsymbol{x}, \boldsymbol{y}_1) + \beta F(\boldsymbol{x}, \boldsymbol{y}_2)$.

Solution. We have that

$$\begin{aligned} F(\alpha \boldsymbol{x}_1 + \beta \boldsymbol{x}_2, \boldsymbol{y}) &= (\alpha \boldsymbol{x}_1 + \beta \boldsymbol{x}_2) \otimes \boldsymbol{y} = (\alpha \boldsymbol{x}_1 + \beta \boldsymbol{x}_2)\boldsymbol{y}^T \\ &= \alpha \boldsymbol{x}_1 \boldsymbol{y}^T + \beta \boldsymbol{x}_2 \boldsymbol{y}^T = \alpha(\boldsymbol{x}_1 \otimes \boldsymbol{y}) + \beta(\boldsymbol{x} \otimes \boldsymbol{y}) \\ &= \alpha F(\boldsymbol{x}_1, \boldsymbol{y}) + \beta F(\boldsymbol{x}_1, \boldsymbol{y}). \end{aligned}$$

The second statement follows similarly.

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.

Solution. Multiplicaton with the matrix

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

reverses the elements in a vector. This means that

$$((S \otimes I)(\boldsymbol{x} \otimes \boldsymbol{y}))_{i,j} = ((S\boldsymbol{x}) \otimes \boldsymbol{y})_{i,j} = (S\boldsymbol{x})_i y_j = x_{M-1-i} y_j = (\boldsymbol{x} \otimes \boldsymbol{y})_{M-1-i,j}.$$

This means that also $((S \otimes I)X)_{i,j} = X_{M-1-i,j}$ for all X, so that $S \otimes I$ reverses rows, and thus is a solution to a).

b) The mapping which reverses the order of the columns in a matrix.

Solution. Similarly one shows that $I \otimes S$ reverses columns, and is thus a solution to b..

c) The mapping which transposes a matrix.

Solution. It turns out that it is impossible to find S_1 and S_2 so that transposing a matrix X corresponds to computing $(S_1 \otimes S_2)X$. To see why, S_1 and S_2 would need to fulfill

$$(S_1 \otimes S_2)(\boldsymbol{e}_i \otimes \boldsymbol{e}_j) = (S_1 \boldsymbol{e}_i) \otimes (S_2 \boldsymbol{e}_j) = \boldsymbol{e}_j \otimes \boldsymbol{e}_i,$$

since $\boldsymbol{e}_j \otimes \boldsymbol{e}_i$ is the transpose of $\boldsymbol{e}_i \otimes \boldsymbol{e}_j$. This would require that $S_1 \boldsymbol{e}_i = \boldsymbol{e}_j$ for all i, j, which is impossible.

Exercise 9.11: Computational molecules

Let the filter S be defined by $S = \{1, \underline{2}, 1\}$.

a) Write down the computational molecule of $S \otimes S$.

Solution. The computational molecule of $S \otimes S$ is

$$(1, \underline{2}, 1) \otimes (1, \underline{2}, 1) = (1, \underline{2}, 1) \otimes (1, \underline{2}, 1) = \begin{pmatrix} 1 \\ \underline{2} \\ 1 \end{pmatrix} \begin{pmatrix} 1 & \underline{2} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}.$$

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 $A = \boldsymbol{x} \otimes \boldsymbol{y} + \boldsymbol{z} \otimes \boldsymbol{w}$.

Solution. We get that

$$
\begin{aligned}
A &= \begin{pmatrix}1\\2\\3\end{pmatrix}\begin{pmatrix}3 & 2 & 1\end{pmatrix} + \begin{pmatrix}2\\2\\2\end{pmatrix}\begin{pmatrix}1 & 4 & 2\end{pmatrix} \\
&= \begin{pmatrix}3 & 2 & 1\\6 & 4 & 2\\9 & 6 & 3\end{pmatrix} + \begin{pmatrix}2 & 8 & 4\\2 & 8 & 4\\2 & 8 & 4\end{pmatrix} = \begin{pmatrix}5 & 10 & 5\\8 & 12 & 6\\11 & 14 & 7\end{pmatrix}.
\end{aligned}
$$

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?

Solution. We need to compute $(S \otimes S)A = SAS^T$, which corresponds to first applying S to every column in the image, and then applying S to every row in the resulting image. If we apply S to every column in the image we first get the matrix $SA = \begin{pmatrix}29\&46\&23\\32\&48\&24\\35\&50\&25\end{pmatrix}$. If we apply the filter to the rows here we get $SAS^T = \begin{pmatrix}127\&144\&121\\136\&152\&128\\145\&160\&135\end{pmatrix}$. Since the filter which is applied is a lowpass filter, the new image should look a bit more smooth than the original image.

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?

Solution. Note forst that the filter is a smoothing filter (a lowpass filter). We know that $S \otimes I$ corresponds to applying S to the columns of the matrix, so that we get the result by applying the smoothing filter to the columns of the matrix. The result of this is that horizontal edges are smoothed. Similarly, the tensor product $I \otimes S$ corresponds to applying S to the rows of the matrix, so that vertical edges are smoothed. Finally, $S \otimes S$ corresponds to applying S first to the columns of the matrix, then to the rows. The result is that both horizontal and vertical edges are smoothed. You could also have computed the computational molecules for $S \otimes I$, $I \otimes S$, and $S \otimes S$, by taking the tensor product of the filter coefficients $\frac{1}{4}\{1, \underline{2}, 1\}$ with itself. From these molecules it is also clear that they either work on the columns, the rows, or on both rows and columns.

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).

Solution. A 4×4 circulant Toeplitz matrix for S is

$$\frac{1}{4}\begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \end{pmatrix}.$$

From this we can quickly compute that

$$S\boldsymbol{x} = \frac{1}{4}\begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \end{pmatrix}\begin{pmatrix} 4 \\ 8 \\ 8 \\ 4 \end{pmatrix} = \begin{pmatrix} 2+2+1 \\ 4+2+1 \\ 4+2+1 \\ 2+2+1 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 7 \\ 5 \end{pmatrix}$$

$$S\boldsymbol{y} = \frac{1}{4}\begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \end{pmatrix}\begin{pmatrix} 8 \\ 4 \\ 8 \\ 4 \end{pmatrix} = \begin{pmatrix} 4+1+1 \\ 2+2+2 \\ 4+1+1 \\ 2+2+2 \end{pmatrix} = \begin{pmatrix} 6 \\ 6 \\ 6 \\ 6 \end{pmatrix}.$$

From this it is clear that

$$(S \otimes S)(\boldsymbol{x} \otimes \boldsymbol{y}) = (S\boldsymbol{x})(S\boldsymbol{y})^T = \begin{pmatrix} 5 \\ 7 \\ 7 \\ 5 \end{pmatrix}\begin{pmatrix} 6 & 6 & 6 & 6 \end{pmatrix} = \begin{pmatrix} 30 & 30 & 30 & 30 \\ 42 & 42 & 42 & 42 \\ 42 & 42 & 42 & 42 \\ 30 & 30 & 30 & 30 \end{pmatrix}.$$

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?

Solution. In the code the filter $S = \{1/4, \underline{1/2}, 1/4\}$ is applied to the clumns and the rows in the image. We have learnt that this corresponds to applying the tensor product $S \otimes S$ to the image. `k=1` in the outer `for`-loop corresponds to

applying S on the columns, `k=2` corresponds to applying S on the rows. The first and last lines in the inner `for`-loop are necessary since we apply S to the periodic extension of the image. Since S is a smoothing filter, the effect will be that the image is smoothed vertically and horizontally.

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 9.3, 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 9.3: Tensor products

Solution. We have that

$$\begin{aligned}
&(A \otimes^k B)(\boldsymbol{x} \otimes^k \boldsymbol{y}) \\
&= \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1M}B \\ a_{21}B & a_{22}B & \cdots & a_{2M}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{p1}B & a_{p2}B & \cdots & a_{pM}B \end{pmatrix} \begin{pmatrix} x_1\boldsymbol{y} \\ x_2\boldsymbol{y} \\ \vdots \\ x_M\boldsymbol{y} \end{pmatrix} = \begin{pmatrix} (a_{11}x_1 + \ldots + a_{1M}x_m)B\boldsymbol{y} \\ (a_{21}x_1 + \ldots + a_{2M}x_m)B\boldsymbol{y} \\ \vdots \\ (a_{p1}x_1 + \ldots + a_{pM}x_m)B\boldsymbol{y} \end{pmatrix} \\
&= \begin{pmatrix} (A\boldsymbol{x})_1 B\boldsymbol{y} \\ (A\boldsymbol{x})_2 B\boldsymbol{y} \\ \vdots \\ (A\boldsymbol{x})_p B\boldsymbol{y} \end{pmatrix} = (A\boldsymbol{x}) \otimes_k (B\boldsymbol{y}).
\end{aligned}$$

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 $L_{M,N}$.

Solution. We have that

$$\begin{aligned}
\langle \boldsymbol{x}_1 \otimes \boldsymbol{y}_1, \boldsymbol{x}_2 \otimes \boldsymbol{y}_2 \rangle &= \left\langle \begin{pmatrix} (\boldsymbol{x}_1)_0 \boldsymbol{y}_1 \\ \vdots \\ (\boldsymbol{x}_1)_{M-1} \boldsymbol{y}_1 \end{pmatrix}, \begin{pmatrix} (\boldsymbol{x}_2)_0 \boldsymbol{y}_2 \\ \vdots \\ (\boldsymbol{x}_2)_{M-1} \boldsymbol{y}_2 \end{pmatrix} \right\rangle = \sum_{i=0}^{M-1} (\boldsymbol{x}_1)_i (\boldsymbol{x}_2)_i \langle \boldsymbol{y}_1, \boldsymbol{y}_2 \rangle \\
&= \langle \boldsymbol{y}_1, \boldsymbol{y}_2 \rangle \sum_{i=0}^{M-1} (\boldsymbol{x}_1)_i (\boldsymbol{x}_2)_i = \langle \boldsymbol{x}_1, \boldsymbol{x}_2 \rangle \langle \boldsymbol{y}_1, \boldsymbol{y}_2 \rangle.
\end{aligned}$$

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.

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.

```
def DFTImpl8(x):
    N = shape(x)[0]
    for n in range(0,N,8):
        x[n:(n+8), :] = fft.fft(x[n:(n+8), :], None, 0)

def IDFTImpl8(x):
    N = shape(x)[0]
    for n in range(0, N, 8):
        x[n:(n+8), :] = fft.ifft(x[n:(n+8), :], None, 0)

def DCTImpl8(x):
    N = shape(x)[0]
    for n in range(0,N,8):
        x[n:(n+8), :] = dct(x[n:(n+8), :], norm='ortho', axis=0)

def IDCTImpl8(x):
    N = shape(x)[0]
    for n in range(0, N, 8):
        x[n:(n+8), :] = idct(x[n:(n+8), :], norm='ortho', axis=0)
```

Solution.

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`.

Solution. The following code can be used.

```
def DFTImplFull(x):
    fft.fft(x, None, 0)

def IDFTImplFull(x):
    fft.ifft(x, None, 0)

def DCTImplFull(x):
    x[:] = dct(x, norm='ortho', axis=0)

def IDCTImplFull(x):
    x[:] = idct(x, norm='ortho', axis=0)
```

```
tensor_impl(X, DFTImplFull, DFTImplFull)
tensor_impl(X, IDFTImplFull, IDFTImplFull)
tensor_impl(X, DCTImplFull, DCTImplFull)
tensor_impl(X, IDCTImplFull, IDCTImplFull)
```

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.

Solution. In the first part of the code one makes a change of coordinates with the DFT. More precisely, this is a change of coordinates on a tensor product, as we have defined it. In the last part the change of coordinates is performed the opposite way. Both these change of coordinates is performed is performed the way we have described them, first on the rows in the matrix, then on the columns. The parameter `threshold` is used to neglect DFT-coefficients which are below a certain value. We have seen that this can give various visual artefacts in the image, even though the main contents of the image still may be visible. If we increase `threshold`, these artefacts will be more dominating since we then neglect many DFT-coefficients.

Chapter 10

Using tensor products to apply wavelets to images

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'.

Solution. The following code can be used:

```
def DWT2Impl(X, nres, f, symm=True, dual=False):
    """
    Compute a 2-dimensional DWT. The one-dimensional DWT is applied
    to each row and column in X at each stage. X may have a third
    axis, as is the case for images with more than one color
    component. The DWT2 is then applied to each color component.

    X: A 2-dimensional object for which we apply the 2-dim DWT
    nres: The number of stages
    f: Wavelet kernel to apply. See DWTImpl for documentation
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    M, N = shape(X)[0:2]
    for res in range(nres):
        for n in range(0,N,2**res):
            f(X[0::2**res, n], symm, dual)
        for m in range(0,M,2**res):
            f(X[m, 0::2**res], symm, dual)
    reorganize_coefficients2(X, nres, True)
```

```
def IDWT2Impl(X, nres, f, symm=True, dual=False):
    """
    Compute a 2-dimensional IDWT. The one-dimensional IDWT is applied
    to each row and column in X at each stage. X may have a third
    axis, as is the case for images with more than one color
    component. The IDWT2 is then applied to each color component.

    X: A 2-dimensional object for which we apply the 2-dim IDWT
    nres: The number of stages
    f: Wavelet kernel to apply. See IDWTImpl for documentation
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    reorganize_coefficients2(X, nres, False)
    M, N = shape(X)[0:2]
    for res in range(nres - 1, -1, -1):
        for n in range(0, N, 2**res):
            f(X[0::2**res, n], symm, dual)
        for m in range(0, M, 2**res):
            f(X[m, 0::2**res], symm, dual)

def DWTImpl(x, nres, f, symm=True, dual=False):
    """
    Compute the DWT of x for a given number of resolutions, using
    wavelet kernel f. The kernel is assumed to compute one level
    of the DWT in-place. DWTImpl is responsible for reorganizing the
    output so that the low resolution coefficients comes first, as
    required by the DWT. The DWT is computed along the first axis.
    x may have a second axis, as is the case for sound with more
    than one channel. The DWT is then applied to each channel.

    x: The vector which we apply the DWT to.
    nres: The number of stages
    f: The wavelet kernel to apply. Supported kernels are
        DWTHaarKernel (Haar wavelet),
        DWTKernelpwl0, DWTKernelpwl2 (piecewise
            linear wavelets with different number of van. moms.),
        DWTKernel53 (Spline 5/3 wavelet, used for lossless
            compression in JPEG2000),
        DWTKernel97 (CDF 9/7 wavelet, used for lossy compression in
            JPEG2000),
        DWTKernelOrtho (Daubechies orthonormal wavelets with number
            of vanishing moments dictated by a global variable)
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    for res in range(nres):
        f(x[0::2**res], symm, dual)
    reorganize_coefficients(x, nres, True)

def IDWTImpl(x, nres, f, symm=True, dual=False):
    """
    Compute the IDWT of x for a given number of resolutions, using
    wavelet kernel f. The kernel is assumed to compute one level
    of the IDWT in-place. IDWTImpl is responsible for reorganizing
    the input so that the kernel can perform in-place calculation.
    The IDWT is computed along the first axis. x may have a second
    axis, as is the case for sound with more than one channel.
    The IDWT is then applied to each channel.

    x: The vector which we apply the IDWT to.
    nres: The number of stages
    f: The wavelet kernel to apply. Supported kernels are
```

```
        IDWTHaarKernel (Haar wavelet),
        IDWTKernelpwl0, DWTKernelpwl2 (piecewise
            linear wavelets with different number of van. moms.),
        IDWTKernel53 (Spline 5/3 wavelet, used for lossless
            compression in JPEG2000),
        IDWTKernel97 (CDF 9/7 wavelet, used for lossy compression in
            JPEG2000),
        IDWTKernelOrtho (Daubechies orthonormal wavelets with number
            of vanishing moments dictated by a global variable)
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    reorganize_coefficients(x, nres, False)
    for res in range(nres - 1, -1, -1):
        f(x[0::2**res], symm, dual)

# Lifting steps

def liftingstepevensymm(lmbda, x, symm):
    """
    Apply an elementary symmetric lifting step of even type to x.

    lmbda: The common value of the two filter coefficients
    x: The vector which we apply the lifting step to
    symm: Whether to apply symmetric extension to the input
    """
    if (not symm) and mod(len(x), 2)!=0:
        raise AssertionError()
    if symm:
        x[0] += 2*lmbda*x[1] # With symmetric extension
    else:
        x[0] += lmbda*(x[1]+x[-1])
    x[2:-1:2] += lmbda*(x[1:-2:2] + x[3::2])
    if mod(len(x), 2)==1 and symm:
        x[-1] += 2*lmbda*x[-2] # With symmetric extension

def liftingstepoddsymm(lmbda, x, symm):
    """
    Apply an elementary symmetric lifting step of odd type to x.

    lmbda: The common value of the two filter coefficients
    x: The vector which we apply the lifting step to
    symm: Whether to apply symmetric extension to the input
    """
    if (not symm) and mod(len(x), 2)!=0:
        raise AssertionError()
    x[1:-1:2] += lmbda*(x[0:-2:2] + x[2::2])
    if mod(len(x), 2)==0:
        if symm:
            x[-1] += 2*lmbda*x[-2] # With symmetric extension
        else:
            x[-1] += lmbda*(x[0]+x[-2])

def liftingstepeven(lmbda1, lmbda2, x):
    """
    Apply an elementary non-symmetric lifting step of even type to x.

    lmbda1: The first filter coefficient
    lmbda2: The second filter coefficient
    x: The vector which we apply the lifting step to
    """
    if mod(len(x), 2)!=0:
        raise AssertionError()
    x[0] += lmbda1*x[1] + lmbda2*x[-1]
```

```
    x[2:-1:2] += lmbda1*x[3::2] + lmbda2*x[1:-2:2]

def liftingstepodd(lmbda1, lmbda2, x):
    """
    Apply an elementary non-symmetric lifting step of odd type to x.

    lmbda1: The first filter coefficient
    lmbda2: The second filter coefficient
    x: The vector which we apply the lifting step to
    """
    if mod(len(x), 2)!=0:
        raise AssertionError()
    x[1:-2:2] += lmbda1*x[2:-1:2] + lmbda2*x[0:-3:2]
    x[-1] += lmbda1*x[0] + lmbda2*x[-2]

# The Haar wavelet

def DWTKernelHaar(x, symm, dual):
    """
    Apply the DWT kernel transformation for the Haar wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    x /= sqrt(2)
    if mod(len(x), 2)==1:
        a, b = x[0] + x[1] - x[-1], x[0] - x[1] - x[-1]
        x[0], x[1] = a, b
        x[-1] *= 2
    else:
        a, b = x[0] + x[1], x[0] - x[1]
        x[0], x[1] = a, b
    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

def IDWTKernelHaar(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the Haar wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    x /= sqrt(2)
    if mod(len(x), 2)==1:
        a, b = x[0] + x[1]  + x[-1], x[0] - x[1]
        x[0], x[1] = a, b
        for k in range(2,len(x) - 2, 2):
            a, b = x[k] + x[k+1], x[k] - x[k+1]
            x[k], x[k+1] = a, b
    else:
        for k in range(0,len(x) - 1, 2):
            a, b = x[k] + x[k+1], x[k] - x[k+1]
            x[k], x[k+1] = a, b

# Piecewise linear wavelets

def DWTKernelpwl0(x, symm, dual):
    """
    Apply the DWT kernel transformation for the
    piecewise linear wavelet (i.e. 0 vanishing moments) to x.
```

```
    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        x /= sqrt(2)
        x = liftingstepevensymm(0.5, x, symm)
    else:
        x *= sqrt(2)
        liftingstepoddsymm(-0.5, x, symm)

def IDWTKernelpwl0(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the
    pieccewise linear wavelet (i.e. 0 vanishing moments) to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        x *= sqrt(2)
        liftingstepevensymm(-0.5, x, symm)
    else:
        x /= sqrt(2)
        liftingstepoddsymm(0.5, x, symm)


def DWTKernelpwl2(x, symm, dual):
    """
    Apply the DWT kernel transformation for the
    alternative pieccewise linear wavelet (i.e. 2 van. moms.) to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        liftingstepevensymm(0.5, x, symm)
        liftingstepoddsymm(-0.25, x, symm)
        x /= sqrt(2)
    else:
        liftingstepoddsymm(-0.5, x, symm)
        liftingstepevensymm(0.25, x, symm)
        x *= sqrt(2)

def IDWTKernelpwl2(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the
    alternative pieccewise linear wavelet (i.e. 2 van. moms.) to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        x *= sqrt(2)
        liftingstepoddsymm(0.25, x, symm)
        liftingstepevensymm(-0.5, x, symm)
    else:
        x /= sqrt(2)
        liftingstepevensymm(-0.25, x, symm)
        liftingstepoddsymm(0.5, x, symm)
```

```
# JPEG2000-related wavelet kernels

def DWTKernel53(x, symm, dual):
    """
    Apply the DWT kernel transformation for the
    Spline 5/3 wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        x[0::2] *= 0.5
        x[1::2] *= 2
        liftingstepevensymm(0.125, x, symm)
        liftingstepoddsymm(-1, x, symm)
    else:
        x[0::2] *= 2
        x[1::2] *= 0.5
        liftingstepoddsymm(-0.125, x, symm)
        liftingstepevensymm(1, x, symm)

def IDWTKernel53(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the
    Spline 5/3 wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    if dual:
        liftingstepoddsymm(1, x, symm)
        liftingstepevensymm(-0.125, x, symm)
        x[0::2] *= 2
        x[1::2] *= 0.5
    else:
        liftingstepevensymm(-1, x, symm)
        liftingstepoddsymm(0.125, x, symm)
        x[0::2] *= 0.5
        x[1::2] *= 2


def DWTKernel97(x, symm, dual):
    """
    Apply the DWT kernel transformation for the CDF 9/7 wavelet to x.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    lambda1=-0.586134342059950
    lambda2=-0.668067171029734
    lambda3=0.070018009414994
    lambda4=1.200171016244178
    alpha=-1.149604398860250
    beta=-0.869864451624777
    if dual:
        x[0::2] /= alpha
        x[1::2] /= beta
        liftingstepevensymm(lambda4, x, symm)
        liftingstepoddsymm(lambda3, x, symm)
        liftingstepevensymm(lambda2, x, symm)
```

```
        liftingstepoddsymm(lambda1, x, symm)
    else:
        x[0::2] *= alpha
        x[1::2] *= beta
        liftingstepoddsymm(-lambda4, x, symm)
        liftingstepevensymm(-lambda3, x, symm)
        liftingstepoddsymm(-lambda2, x, symm)
        liftingstepevensymm(-lambda1, x, symm)

def IDWTKernel97(x, symm, dual):
    """
    Apply the IDWT kernel transformation for the CDF 9/7 wavelet to x

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    lambda1=-0.586134342059950
    lambda2=-0.668067171029734
    lambda3=0.070018009414994
    lambda4=1.200171016244178
    alpha=-1.149604398860250
    beta=-0.869864451624777
    if dual:
        liftingstepoddsymm(-lambda1, x, symm)
        liftingstepevensymm(-lambda2, x, symm)
        liftingstepoddsymm(-lambda3, x, symm)
        liftingstepevensymm(-lambda4, x, symm)
        x[0::2] *= alpha
        x[1::2] *= beta
    else:
        liftingstepevensymm(lambda1, x, symm)
        liftingstepoddsymm(lambda2, x, symm)
        liftingstepevensymm(lambda3, x, symm)
        liftingstepoddsymm(lambda4, x, symm)
        x[0::2] /= alpha
        x[1::2] /= beta

# Orthonormal wavelets

def DWTKernelOrtho( x, symm, dual):
    """
    Apply the DWT kernel transformation for orthonormal wavelets.
    The number of vanishing moments is stored in global variables.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    global lambdas, alpha, beta
    global beta
    if dual:
        x[0::2] /= alpha
        x[1::2] /= beta
        for stepnr in range(lambdas.shape[0] - 1, 0, -2):
            liftingstepodd(lambdas[stepnr, 1], lambdas[stepnr, 0], x)
            liftingstepeven(lambdas[stepnr -1, 1], lambdas[stepnr - 1, 0], x)
        if mod(lambdas.shape[0], 2)==1:
            liftingstepodd(lambdas[0, 1], lambdas[0, 0], x)
    else:
        x[0::2] *= alpha
        x[1::2] *= beta
        for stepnr in range(lambdas.shape[0] - 1, 0, -2):
            liftingstepeven(-lambdas[stepnr, 0], -lambdas[stepnr, 1], x)
```

```
            liftingstepodd(-lambdas[stepnr - 1, 0], -lambdas[stepnr - 1, 1], x)
        if mod(lambdas.shape[0], 2)==1:
            liftingstepeven(-lambdas[0, 0], -lambdas[0, 1], x)

def IDWTKernelOrtho( x, symm, dual):
    """
    Apply the IDWT kernel transformation for orthonormal wavelets.
    The number of vanishing moments is stored in global variables.

    x: The vector which we apply this kernel transformation to
    symm: Whether to apply symmetric extension to the input
    dual: Whether to apply the wavelet kernel or dual wavelet kernel.
    """
    global lambdas
    global alpha
    global beta
    if dual:
        stepnr = 0
        if mod(lambdas.shape[0], 2) == 1: # Start with an odd step
            liftingstepodd(-lambdas[stepnr, 1], -lambdas[stepnr, 0], x)
            stepnr += 1
        while stepnr < lambdas.shape[0]:
            liftingstepeven(-lambdas[stepnr, 1], -lambdas[stepnr, 0], x)
            liftingstepodd(-lambdas[stepnr + 1, 1], -lambdas[stepnr + 1, 0], x)
            stepnr += 2
        x[0::2] *= alpha
        x[1::2] *= beta
    else:
        stepnr = 0
        if mod(lambdas.shape[0],2) == 1: # Start with an even step
            liftingstepeven(lambdas[stepnr, 0], lambdas[stepnr, 1], x)
            stepnr += 1
        while stepnr < lambdas.shape[0]:
            liftingstepodd(lambdas[stepnr, 0], lambdas[stepnr, 1], x)
            liftingstepeven(lambdas[stepnr + 1, 0], lambdas[stepnr + 1, 1], x)
            stepnr += 2
        x[0::2] /= alpha
        x[1::2] /=beta


# Code to test wavelets on sound

def playDWT(m, f, invf, lowres = True):
    """
    Play a sound after removing either the detail or the lowres part.

    m: The number of resolutions
    f: The DWT kernel
    invf: The IDWT kernel
    lowres: If true, set the detail to 0 and play the lowres part.
            If false, set the lowres part to 0 and play the detail.
    """
    x, fs = audioread('sounds/castanets.wav')
    N = 2**17
    x = x[0:N]
    DWTImpl(x, m, f)
    if lowres:
        x[(N/2**m):N] = 0
    else:
        x[0:(N/2**m)] = 0
    IDWTImpl(x, m, invf)
    play(x, fs)

def showDWT(m, f, invf, lowres = True):
```

```
    """
    Show an image after removing either the detail or the lowres part

    m: The number of resolutions
    f: The DWT kernel
    invf: The IDWT kernel
    lowres: If true, set the detail to 0 and show the lowres part.
            If false, set the lowres part to 0 and show the detail.
    """
    img = double(imread('images/lena.png'))
    M, N = shape(img)[0:2]
    DWT2Impl(img, m, f)
    if lowres:
        tokeep = img[0:(M/(2**m)), 0:(N/(2**m))]
        img=zeros_like(img)
        img[0:(M/(2**m)),0:(N/(2**m))] = tokeep
    else:
        img[0:(M/2**m), 0:(N/2**m)] = 0
    IDWT2Impl(img, m, invf)
    mapto01(img)
    img *= 255
    imshow(img.astype(uint8))

# testcode




def h0h1computeortho(N):
    vals = computeQN(N)
    rts=roots(vals)
    rts1=rts[nonzero(abs(rts)>1)]
    g0=array([1])
    for rt in rts1:
        g0=convolve(g0,[-rt,1])

    K=sqrt(vals[0]*(-1)**(len(rts1))/prod(rts1))
    g0=K*g0
    for k in range(N):
        g0=convolve(g0,[1/2.,1/2.])

    g0=real(g0)
    h0=g0[::-1]
    g1=g0[::-1]*(-1)**(array(range(len(g0))))
    h1=g1[::-1]
    return h0, h1

def liftingfactortho(N):
    """
    Assume that len(h1)==len(h0), and that h0 and h1 are even length and as symmetric as possible, w
    This function computes lifting steps l1, l2,...,ln, and constants alpha, beta so that ln ... l2
    This gives the following recipes for
        Computing H: first multiply with diag(alpha,beta), then the inverses of the lifting steps in
        Computing G: apply the lifting steps in the order given, finally multiply with diag(1/alpha,
    ln is always odd, so that l1 is odd if and only if n is odd.
    All even lifting steps have only filter coefficients 0,1. All odd lifting steps have only filter
    """
    global lambdas, alpha, beta
    h0, h1 = h0h1computeortho(N)
    stepnr=0
    start1, end1, len1, start2, end2, len2 = 0, len(h0)/2-1, len(h0)/2,  0, len(h1)/2-1, len(h1)/2
    lambdas=zeros((len1+1,2))
    if mod(len1,2)==0: # Start with an even step
        h00, h01 = h0[0:len(h0):2], h0[1:len(h0):2]
```

```
        h10, h11 = h1[0:len(h1):2], h1[1:len(h1):2]

        lambda1=-h00[0]/h10[0]
        h00=h00+lambda1*h10
        h01=h01+lambda1*h11
        start1, end1, len1 = 1, len1-1, len1-1
        lambdas[stepnr,:] = [lambda1,0]
    else: # Start with an odd step
        h00, h01 = h0[1:len(h0):2], h0[0:len(h0):2]
        h10, h11 = h1[1:len(h1):2], h1[0:len(h1):2]

        lambda1=-h10[end1]/h00[end1]
        h10=h10+lambda1*h00
        h11=h11+lambda1*h01
        start2, end2, len2 = 0, len2 - 2, len2-1
        lambdas[stepnr,:] = [0,lambda1]

    #[h00 h01; h10 h11]
    #convolve(h00,h11)-convolve(h10,h01)
    stepnr=stepnr+1

    # print [h00 h01; h10 h11], convolve(h00,h11)-convolve(h10,h01)
    while len2>0: # Stop when the second element in the first column is zero
        if len1>len2: # Reduce the degree in the first row.
            lambda1=-h00[start1]/h10[start2]
            lambda2=-h00[end1]/h10[end2]
            h00[start1:(end1+1)] = h00[start1:(end1+1)]+convolve(h10[start2:(end2+1)],[lambda1,lambd
            h01[start1:(end1+1)] = h01[start1:(end1+1)]+convolve(h11[start2:(end2+1)],[lambda1,lambd
            start1, end1, len1 = start1+1, end1-1, len1-2
        else: # reduce the degree in the second row.
            lambda1=-h10[start2]/h00[start1]
            lambda2=-h10[end2]/h00[end1]
            h10[start2:(end2+1)] = h10[start2:(end2+1)]+convolve(h00[start1:(end1+1)],[lambda1,lambd
            h11[start2:(end2+1)] = h11[start2:(end2+1)]+convolve(h01[start1:(end1+1)],[lambda1,lambd
            start2, end2, len2 = start2+1, end2-1, len2-2
        lambdas[stepnr,:]=[lambda1,lambda2]
        stepnr=stepnr+1

    # print [h00 h01; h10 h11], convolve(h00,h11)-convolve(h10,h01)

    # Add the final lifting, and compute alpha,beta
    alpha=sum(h00)
    beta=sum(h11)
    lastlift=-sum(h01)/beta
    if mod(len(h0)/2,2)==0:
        lambdas[stepnr,:] = [0,lastlift]
    else:
        lambdas[stepnr,:] = [lastlift,0]
    # [h00 h01; h10 h11]

def computeQN(N):
    """
    Compute the coefficients of the polynomial Q^(N)((1-cos(w))/2).
    """
    QN=zeros(N)
    for k in range(N):
        QN[k] = 2*math.factorial(N+k-1)/(math.factorial(k)*math.factorial(N-1))
    vals = array([QN[0]])
    start = array([1.0])
    for k in range(1,N):
        start = convolve(start,[-1/4.0,1/2.0,-1/4.0])
        vals = hstack([0,vals])
        vals = hstack([vals,0])
        vals = vals + QN[k]*start
```

```
    return vals



def h0h1compute97():
    QN = computeQN(4)

    rts = roots(QN)
    rts1 = rts[nonzero(abs(imag(rts))>0.001)] # imaginary roots
    rts2 = rts[nonzero(abs(imag(rts))<0.001)] # real roots

    h0=array([1])
    for rt in rts1:
        h0 = convolve(h0, [-rt,1])
    for k in range(2):
        h0 = convolve(h0,[1/4.,1/2.,1/4.])
    h0=h0*QN[0]

    g0=array([1])
    for rt in rts2:
        g0 = convolve(g0, [-rt,1])
    for k in range(2):
        g0 = convolve(g0,[1/4.,1/2.,1/4.])

    g0, h0 = real(g0), real(h0)
    x = sqrt(2)/abs(sum(h0))
    g0, h0 = g0/x, h0*x
    N= g0.shape[0]
    h1=g0*(-1)**(array(range(-(N-1)/2,(N+1)/2)))
    N= h0.shape[0]
    g1=h0*(-1)**(array(range(-(N-1)/2,(N+1)/2)))
    #print h0, h1, g0, g1
    return h0, h1

def liftingfact97():
    h0, h1 = h0h1compute97() # Should have 9 and 7 filter coefficients.
    h00, h01 = h0[0:9:2], h0[1:9:2]
    h10, h11 = h1[0:7:2], h1[1:7:2]

    lambdas=zeros(4)

    lambdas[0] = -h00[0]/h10[0]
    h00[0:5] = h00[0:5]+convolve(h10[0:4],[lambdas[0],lambdas[0]])
    h01[0:4] = h01[0:4]+convolve(h11[0:3],[lambdas[0],lambdas[0]])

    lambdas[1] = -h10[0]/h00[1]
    h10[0:4] = h10[0:4]+convolve(h00[1:4],[lambdas[1],lambdas[1]])
    h11[0:3] = h11[0:3]+convolve(h01[1:3],[lambdas[1],lambdas[1]])

    lambdas[2] = -h00[1]/h10[1]
    h00[1:4] = h00[1:4]+convolve(h10[1:3],[lambdas[2],lambdas[2]])
    h01[1:3] = h01[1:3]+convolve(h11[1:2],[lambdas[2],lambdas[2]])

    lambdas[3] = -h10[1]/h00[2]
    h10[0:4] = h10[0:4]+convolve(h00[1:4],[lambdas[3],lambdas[3]])
    h11[0:3] = h11[0:3]+convolve(h01[1:3],[lambdas[3],lambdas[3]])

    alpha, beta = h00[2], h11[1]
    return lambdas, alpha, beta

def _test_kernel(f,invf,text):
    print text
    res = random.random(16)
    x = zeros(16)
```

```
    x[:] = res[:]
    DWTImpl(x,2,f)
    IDWTImpl(x,2,invf)
    diff = abs(x-res).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

    res = random.random((16,2))
    x = zeros((16,2))
    x[:] = res[:]
    DWTImpl(x,2,f)
    IDWTImpl(x,2,invf)
    diff = abs(x-res).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

def _test_kernel_ortho():
    print 'Testing orthonormal wavelets'
    liftingfactortho(4)
    res = random.random(16) # only this assumes that N is even
    x = zeros(16)

    print 'Testing that the reverse inverts the forward transform'
    x[0:16] = res[0:16]
    DWTImpl(x, 2, DWTKernelOrtho)
    IDWTImpl(x, 2, IDWTKernelOrtho)
    diff = max(abs(x-res))
    assert diff < 1E-13, 'bug, diff=%s' % diff

    print 'Testing that the transform is orthogonal, i.e. that the transform and its dual are equal'
    x[0:16] = res[0:16]
    DWTImpl(x, 2, DWTKernelOrtho)
    DWTImpl(res, 2, DWTKernelOrtho, False, True)
    diff = max(abs(x-res))
    assert diff < 1E-13, 'bug, diff=%s' % diff

def _test_dwt_different_sizes():
    print 'Testing the DWT on different input sizes'
    m = 4
    f = DWTKernel97
    invf = IDWTKernel97

    print 'Testing the DWT for greyscale image'
    img = random.random((32,32))
    img2 = zeros_like(img)
    img2[:] = img[:]
    DWT2Impl(img2, m, f)
    IDWT2Impl(img2, m, invf)
    diff = abs(img2-img).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

    print 'Testing the DWT for RGB image'
    img = random.random((32, 32, 3))
    img2 = zeros_like(img)
    img2[:] = img[:]
    DWT2Impl(img2, m, f)
    IDWT2Impl(img2, m, invf)
    diff = abs(img2-img).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

    print 'Testing the DWT for sound with one channel'
    sd = random.random(32)
    sd2 = zeros_like(sd)
    sd2[:] = sd[:]
    DWTImpl(sd2, m, f)
    IDWTImpl(sd2, m, invf)
```

```
    diff = abs(sd2-sd).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

    print 'Testing the DWT for sound with two channels'
    sd = random.random((32,2))
    sd2 = zeros_like(sd)
    sd2[:] = sd[:]
    DWTImpl(sd2, m, f)
    IDWTImpl(sd2, m, invf)
    diff = abs(sd2-sd).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

def _test_orthogonality():
    print 'Testing that the wavelet and the dual wavelet are equal for orthonormal wavelets'
    liftingfactortho(4)
    x0 = random.random(32)

    print 'Testing that the IDWT inverts the DWT'
    x = x0.copy()
    DWTImpl(x, 2, DWTKernelOrtho, 0, 0)
    IDWTImpl(x, 2, IDWTKernelOrtho, 0, 0);
    diff = abs(x-x0).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

    print 'Apply the transpose, to see that the transpose equals the inverse'
    x = x0.copy()
    DWTImpl(x, 2, DWTKernelOrtho, 0, 0)
    IDWTImpl(x, 2, IDWTKernelOrtho, 0, 1)
    diff = abs(x-x0).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

    print 'To see this at the level of kernel transformations'
    x = x0.copy()
    DWTKernelOrtho(x, 0, 0)
    IDWTKernelOrtho(x, 0, 1)
    diff = abs(x-x0).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff

    print 'See that the wavelet transform equals the dual wavelet transform'
    x = x0.copy()
    DWTImpl(x, 2, DWTKernelOrtho, 0, 1)
    DWTImpl(x0, 2, DWTKernelOrtho, 0, 0)
    diff = abs(x-x0).max()
    assert diff < 1E-13, 'bug, diff=%s' % diff



if __name__=='__main__':
    _test_dwt_different_sizes()
    _test_kernel_ortho()
    _test_orthogonality()
    _test_kernel( DWTKernel97, IDWTKernel97, 'Testing CDF 9/7 wavelet')
    _test_kernel( DWTKernel53, IDWTKernel53, 'Testing Spline 5/3 wavelet')
    _test_kernel( DWTKernelpwl0, IDWTKernelpwl0, 'Testing piecewise linear wavelet')
    _test_kernel( DWTKernelpwl2, IDWTKernelpwl2, 'Testing alternative piecewise linear wavelet')
    _test_kernel( DWTKernelHaar, IDWTKernelHaar, 'Testing Haar wavelet')
```

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.

Solution. The code runs a DWT over one level, and the Haar wavelet is used. Inside the `for`-loops the DWT is applied to every row and column in the image. `k=1` i the `for`-loop corresponds to applying the DWT to the columns, `k=2` corresponds to applying the DWT to the rows. In the upper left corner we will see a low-resolution version of the image. In the other three corners you will see different tyoes of detail: In the upper right corner you will see detail which corresponds to quick vertical changes, in the lower left corner you will see detail which corresponds to quick horizontal changes, and in the lower right corner you will see points where quick changes both vertically and horizontally occur simultaneously.

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?

Solution. By zeroing out the two corners you remove detail which correpond to quick horizontal and vertical changes. But since we keep the lower right corner, we keep detail which corresponds to simultaneous changes vertically and horizontally. The result after the inverse transformation is that most edges have been smoothed, but we see no smoothing effect in points where quick changes occur both horizontally and vertically. In Example 10.14 in the compendium, this corresponds to that we emphasize the gridpoints in the chess pattern, mut that we smooth out the horizontal and vertical edges in the chess pattern.

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 $\phi_0 \otimes \phi_0$),
- 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.

Solution. The following code achieves the task:

```
def showDWT(m, f, invf, lowres = True):
    """
    Show an image after removing either the detail or the lowres part

    m: The number of resolutions
    f: The DWT kernel
    invf: The IDWT kernel
```

```
    lowres: If true, set the detail to 0 and show the lowres part.
            If false, set the lowres part to 0 and show the detail.
    """
    img = double(imread('images/lena.png'))
    M, N = shape(img)[0:2]
    DWT2Impl(img, m, f)
    if lowres:
        tokeep = img[0:(M/(2**m)), 0:(N/(2**m))]
        img=zeros_like(img)
        img[0:(M/(2**m)),0:(N/(2**m))] = tokeep
    else:
        img[0:(M/2**m), 0:(N/2**m)] = 0
    IDWT2Impl(img, m, invf)
    mapto01(img)
    img *= 255
    imshow(img.astype(uint8))
```

b) Do the image samples returned by `showDWT` lie in $[0, 255]$?

Solution. There is no reason to believe that image samples returned by the function lie in $[0, 255]$. You can check this by printing the maximum value in the returned array on screen inside this method.

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 in the compendium, set `lowres` to false in the call to `showDWT` also for the other wavelets. and repeat what you did in d..

Solution. After the replacements we get the following code.

```
showDWT(m, DWTKernelHaar, IDWTKernelHaar, False)
showDWT(m, DWTKernel53, IDWTKernel53, False)
showDWT(m, DWTKernel97, IDWTKernel97, False)
```

Exercise 10.5: Experiments on a test image

In Figure 10.1 we have applied the DWT2 with the Haar wavelet to an image very similar to the one you see in Figure 10.6 in the compendium. You see here, however, that there seems to be no detail components, which is very different from Figure 10.6 in the compendium, even though the images are very similar. Attempt to explain what causes this to happen.

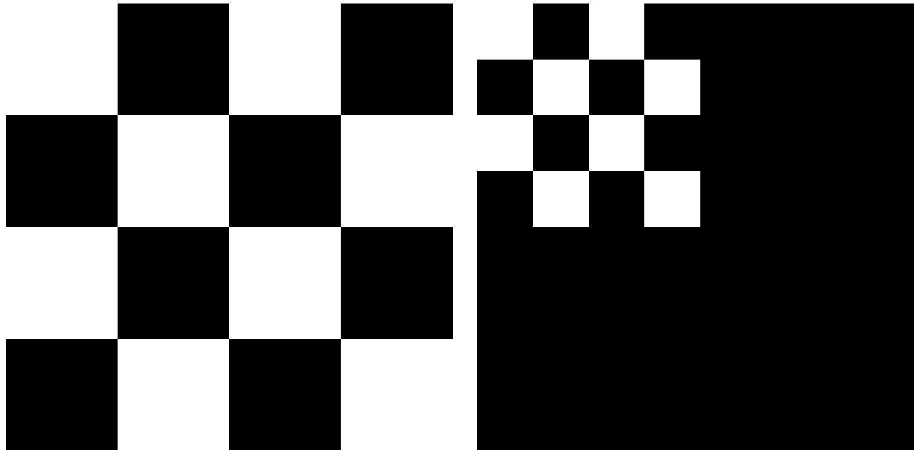

Figure 10.1: A simple image before and after one level of the DWT2. The Haar wavelet was used.

Hint. Compare with Exercise 5.17.

Solution. In Figure 10.6 in the compendium, the borders in the chess pattern was chosen so that they occur at odd numbers. This means that the image can not be represented exactly in $V_{m-1} \otimes V_{m-1}$, so that there is detail present in the image at all the borders in the chess pattern. In Figure 10.1, the borders in the chess pattern was chosen so that they occur at even numbers. This means that the image can be represented exactly in $V_{m-1} \otimes V_{m-1}$, so that there is no detail components present in the image.

Exercise 10.6: Implement the fingerprint compression scheme

Write code which generates the images shown in figures 10.6 in the compendium, 10.6 in the compendium, and 10.23 in the compendium. Use the functions `DW2TImpl` and `IDW2TImpl` with the CDF 9/7 wavelet kernel functions as input.

Appendix A

Basic Linear Algebra

Appendix B

Signal processing and linear algebra: a translation guide