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

Øyvind Ryan

Jan 13, 2017

Contents

Chapter 1

Sound and Fourier series

Exercise 1.6: 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 1.7: 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 1.8: 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 the right part of Figure 1.1 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 1.9: Playing with different sample rates

If we provide another samples rate `fs` to the `play` functions, the sound card will assume a different time distance between neighboring samples. Play and listen to the audio sample file again, but with three different sample rates: `2*fs`, `fs`, and `fs/2`, where `fs` is the sample rate returned by `audioread`.

Solution. The following code can be used

```
play(x, fs)
```

```
play(x, 2*fs)
```

```
play(x, fs/2)
```

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

Exercise 1.10: Play sound with added noise

To remove noise from recorded sound can be very challenging, but adding noise is simple. There are many kinds of noise, but one kind is easily obtained by adding random numbers to the samples of a sound. For this we can use the function `random.random` as follows.

```
z = x + c*(2*random.random(shape(x))-1)
```

This adds noise to all channels. The function for returning random numbers returns numbers between 0 and 1, and above we have adjusted these so that

they are between -1 and 1 instead, as for other sound which can be played by the computer. c is a constant (usually smaller than 1) that dampens the noise.

Write code which adds noise to the audio sample file, and listen to the result for damping constants `c=0.4` and `c=0.1`. Remember to scale the sound values after you have added noise, since they may be outside $[-1, 1]$ then.

Solution. The following code can be used.

```
z = x + c*(2*random.random(shape(x))-1)
z /= abs(z).max()
play(z, fs)
```

With $c = 0.4$ the result sounds like this, while with $c = 0.1$ it sounds like this.

Exercise 1.11: Playing the triangle wave

Repeat what you did in Example 1.4 in the compendium, but now for the triangle wave of Example 1.5 in the compendium. Start by generating the samples for one period of the triangle wave, then plot five periods, before you generate the sound over a period of three seconds, and play it. Verify that you generate the same sound as in Example 1.5 in the compendium.

Solution. The triangle wave can be plotted as follows:

```
totransl=1-4*abs(t-T/2)/T
plt.figure()
plt.plot(concatenate([t, t+T, t+2*T, t+3*T, t+4*T]),tile(totransl,5), 'k-')
```

The samples for one period are created as follows.

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

Then we repeat one period to obtain a sound with the desired length, and play it as follows.

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

Exercise 1.15: Shifting the Fourier basis vectors

Show that $\sin(2\pi nt/T + a) \in V_{N,T}$ when $|n| \leq N$, regardless of the value of a.

Solution. Write $\sin(2\pi nt/T + a) = \cos a \sin(2\pi nt/T) + \sin a \cos(2\pi nt/T)$.

Exercise 1.16: Playing the Fourier series of the triangle wave

a) Plot the Fourier series of the triangle wave.

Solution. The following code can be used.

```
t = linspace(0, T, 100)
x = zeros(len(t))
for k in range(1, 20, 2):
    x -= (8/(k*pi)**2)*cos(2*pi*k*t/T)
plt.figure()
plt.plot(t, x, 'k-')
```

b) Write code so that you can listen to the Fourier series of the triangle wave. How high must you choose N for the Fourier series to be indistuingishable from the square/triangle waves themselves?

Solution. The following code can be used.

```
x = tile(oneperiod, antsec/T)
play(x, fs)
```

Exercise 1.17: 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

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

Exercise 1.18: 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 1.19: antisymmetric functions are sine-series

Prove the second part of Theorem 1.10 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 1.20: More connections between symmetric-/antisymmetric functions and sine-/cosine series

Show that

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

b) Any sine series $\sum_{n=1}^{N} b_n \sin(2\pi nt/T)$ is an antisymmetric function.

c) Any periodic function can be written as a sum of a symmetric - and an antisymmetric function by writing $f(t) = \frac{f(t)+f(-t)}{2} + \frac{f(t)-f(-t)}{2}$.

d) If $f_N(t) = a_0 + \sum_{n=1}^{N}(a_n \cos(2\pi nt/T) + b_n \sin(2\pi nt/T))$, then

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

Exercise 1.21: 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 1.22: 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.

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 1.23: 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 1.27: 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

$$\begin{aligned}
\langle e^{2\pi in_1 t/T}, e^{2\pi in_2 t/T}\rangle &= \frac{1}{T}\int_0^T e^{2\pi in_1 t/T}e^{-2\pi in_2 t/T}dt = \frac{1}{T}\int_0^T e^{2\pi i(n_1-n_2)t/T}dt \\
&= \left[\frac{T}{2\pi i(n_1-n_2)}e^{2\pi i(n_1-n_2)t/T}\right]_0^T \\
&= \frac{T}{2\pi i(n_1-n_2)} - \frac{T}{2\pi i(n_1-n_2)} = 0.
\end{aligned}$$

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

Exercise 1.28: 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 1.29: Complex Fourier series of polynomials

Repeat Exercise 1, computing the complex Fourier series instead of the real Fourier series.

Exercise 1.30: 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 1.31: 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 1.12 in the compendium for the complex case. Use Theorem **??** 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 **??** in the compendium together with the coefficients $b_n = \frac{2(1-\cos(n\pi)}{n\pi}$ we computed in Example 1.12 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 1.32: Complex Fourier coefficients of the triangle wave

Repeat Exercise 1 for the triangle wave.

Exercise 1.33: 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 **??** in the compendium to verify your calculations from Exercise 1.

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

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

From Exercise 1 we had $b_n = -\frac{T}{\pi n}$, for which Theorem **??** 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

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

From Exercise 1 we had $a_n = \frac{T^2}{\pi^2n^2}$ and $b_n = -\frac{T^2}{\pi n}$, for which Theorem **??** 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 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 **??** 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 1.34: Complex Fourier coefficients for symmetric and antisymmetric functions

In this exercise we will prove a version of Theorem 1.10 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 **??** 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 **??** 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 $\sum_{n=-N}^{N} y_n e^{2\pi i n t/T}$ is antisymmetric when $y_0 = 0$ and $y_{-n} = -y_n$ for all n, and rewrite it as a sine-series.

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

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

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

Exercise 1.36: Fourier series of a delayed square wave

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

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

f is just the square wave, delayed with $d = -T/4$. Compute the Fourier coefficients of f directly, and use Property 4 in Theorem 1.15 in the compendium to verify your result.

Solution. We obtain that

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

The square wave defined in this exercise can be obtained by delaying our original square wave with $-T/4$. Using Property 3 in Theorem 1.15 in the compendium with $d = -T/4$ on the complex Fourier coefficients

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

which we obtained for the square wave in Exercise 1, 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 1.37: 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 1.15 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 **??** 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 1.15 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 1.38: 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 $a_{2n}=y_n-y_{-n}$.

Exercise 1.39: Filters preserve sine- and cosine-series

An analog filter where $\lambda_s(\nu)=\lambda_s(-\nu)$ is also called a *symmetric filter*.

a) Prove that, if the input to a symmetric filter is a Fourier series which is a cosine series/sine-series, then the output also is a cosine/sine series.

Solution. We have that

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

so that s preserves cosine-series. A similar computation holds for sine-series.

b) Show that $s(f)=\int_{-a}^{a}g(s)f(t-s)ds$ is a symmetric filter whenever g is symmetric around 0 and supported on $[-a,a]$.

Solution. We have that

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

so that s is symmetric.

We saw that the symmetric extension of a function took the form of a cosine-series, and that this converged faster to the symmetric extension than the Fourier series did to the function. If a filter preserves cosine-series it will also preserve symmetric extensions, and therefore also map fast-converging Fourier series to fast-converging Fourier series.

Exercise 1.40: Approximation in norm with continuous functions

Show that if f is a function with only a finite number of discontinuities, there exists a continuous function g so that $\|f - g\| < \epsilon$.

Exercise 1.41: The Dirichlet kernel

The Dirichlet kernel is defined as

$$D_N(t) = \sum_{n=-N}^{N} e^{2\pi i n t/T} = 1 + 2\sum_{n=1}^{N} \cos(2\pi n t/T).$$

D_N is clearly trigonometric, and of degree N.

a) Show that $D_N(t) = \frac{\sin(\pi(2N+1)t/T)}{\sin(\pi t/T)}$.

b) Show that $f_N(t) = \frac{1}{T}\int_0^T f(t-u)D_N(u)du$. Proving that $\lim_{N\to\infty} f_N(t) = f(t)$ is thus equivalent to $\lim_{N\to\infty} \frac{1}{T}\int_0^T f(t-u)D_N(u)du = f(t)$.

Solution. We have that

$$\begin{aligned} f_N(t) &= \sum_{n=-N}^{N} \frac{1}{T}\int_0^T f(s)e^{-2\pi i n s/T}ds\, e^{2\pi i n t/T} \\ &= \frac{1}{T}\int_0^T f(s) \sum_{n=-N}^{N} e^{-2\pi i n s/T} e^{2\pi i n t/T} ds \\ &= \frac{1}{T}\int_0^T f(s) \sum_{n=-N}^{N} e^{2\pi i n (t-s)/T} ds = \frac{1}{T}\int_0^T f(s)D_N(t-s)ds. \end{aligned}$$

c) Prove that $D_N(t)$ satisfies only two of the properties of a summability kernel.

d) Write a function which takes N and T as arguments, and plots $D_N(t)$ over $[-T/2, T/2]$.

Exercise 1.42: The Fejer summability kernel

The Fejer kernel is defined as

$$F_N(t) = \sum_{n=-N}^{N} \left(1 - \frac{|n|}{N+1}\right) e^{2\pi i n t/N}.$$

F_N is clearly trigonometric, and of degree N.

a) Show that $F_N(t) = \frac{1}{N+1}\left(\frac{\sin(\pi(N+1)t/T)}{\sin(\pi t/T)}\right)^2$, and conclude from this that $0 \leq F_N(t) \leq \frac{T^2}{4(N+1)t^2}$.

Hint. Use that $\frac{2}{\pi}|u| \leq |\sin u|$ when $u \in [-\pi/2, \pi/2]$.

b) Show that $F_N(t)$ satisfies the three properties of a summability kernel.

c) Show that $\frac{1}{T}\int_0^T f(t-u)F_N(u)du = \frac{1}{N+1}\sum_{n=0}^{N} f_n$.

Hint. Show that $F_N(t) = \frac{1}{N+1}\sum_{n=0}^{N} D_n(t)$, and use Exercis 1 b).

d) Write a function which takes N and T as arguments, and plots $F_N(t)$ over $[-T/2, T/2]$.

Chapter 2

Digital sound and Discrete Fourier analysis

Exercise 2.6: Computing the DFT by hand

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

Solution. As in Example 2.3 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 2.7: Exact form of low-order DFT matrix

As in Example 2.3 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 $\frac{1}{\sqrt{8}} e^{\pi i n k/4}$, 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 2.8: 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 **??** 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 2.9: 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 **??** 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 2.10: 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 2.11: 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 2.12: 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 2.13: DFT implementation

Extend the code for the function `DFTImpl` in Example 2.4 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 2.14: 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

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

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 2.15: 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 **??** 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 **??** 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, axis=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 2.9 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, axis=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'])
```

Exercise 2.22: Extra results for the FFT when $N = N_1N_2$

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

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

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

Exercise 2.23: Extend implementation

Recall that, in Exercise 2, 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 2.18 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 2.24: 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 2.4 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, axis=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'])
```

Exercise 2.25: 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 2.26: 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, 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, 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 $e^{-2\pi i n/N}$ on the diagonal. Show that $D_{N/2} = \begin{pmatrix} G_{N/4} & \mathbf{0} \\ \mathbf{0} & -iG_{N/4} \end{pmatrix}$.

b) Let $H_{N/4}$ be the $(N/4) \times (N/4)$ diagonal matrix $G_{D/4}D_{N/4}$. Verify the following rewriting of Equation (2.1):

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

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 $O\left(4N\log_2 N\right)$, which is a reduction of $N\log_2 N$ from the FFT algorithm.

e) Write an FFT kernel function `FFTKernelSplitradix` for the split-radix algorithm (again this should handle both the forward and reverse transforms). Are there more or less recursive function calls in this function than in the original FFT algorithm? Also compare the execution times with our original FFT algorithm, as we did in Exercise 2. 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, 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 2.15 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).