TU Delft
Faculty of Electrical Engineering, Mathematics and Computer Science

# Practicum EE2T11 Telecommunication: Signals and Systems

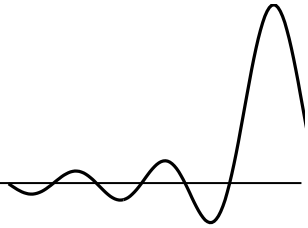
Alle-Jan van der Veen and Jorge Martinez

January 18, 2018

# Contents

*Chapter* **1**

# INTRODUCTION



## Contents



Welcome to the practicum of EE2T11 Telecommunication. In this 2 EC practicum (6 labdays), you will apply some of the theory and tools learned in the EE2S11 "Signals and Systems" course, namely convolution, the Fourier transform and its properties, and filter design. A new tool that will be explored is channel estimation, and we will use the DFT (or FFT) instead of the DTFT. We will also introduce the effects of downsampling and upsampling.

This prepares for the EPO4 (EE2L21) project, and also for the EE2S31 Digital Signal Processing course, where the theory behind the DFT and resampling will be discussed. (There is little connection to EE2T11 Telecommunication except that that course is also based on the same concepts of Signals and Systems.)

The practicum is entirely done with Matlab. We use audio signals (speakers, microphone) as test signals. Your results are documented in a compact report which will be graded Pass/Fail. A Pass is required to enter the EPO4 project and to obtain a valid grade for EE2T11.

The EPO4 project is about driving a toy car from A to B. It is necessary to locate the car. For this, the car has an audio beacon which transmits coded pulse sequences that are received by 4 or 5 microphones at the corners of a test area. By comparing the differences in time-of-arrival of the pulse sequences, the location of the car can be computed.

## 1.1 SCOPE

The practicum is intended to familiarize you with the material learned at the Signals and Systems course. You have seen the definitions of convolution and Fourier transform, but what does it mean when you look at actual signals?

We also apply some topics from linear algebra (matrix inversion, QR) and apply a new tool (SVD) which is related to the eigenvalue decomposition. We will look at

- Convolution, how to interprete a convolution; the related concept of correlation;
- The relation between time domain and frequency domain; basic filtering operations;
- Modeling an audio channel as an FIR filter; convolution of a signal with this filter;
- Channel estimation, i.e., estimation of the audio channel from a measured channel response to a known signal. Design of various known signals to improve the performance of the channel estimate.
- Equalization, i.e., inversion of an estimated channel. This can be used to obtain the original signal from the received signal. This also shows the use of the SVD.
- Filter design: how to use Matlab to design digital filters.
- Sampling: resampling, aliasing and interpolation.

After this practicum, you are able to estimate the channel impulse response from an audio source to a microphone, and estimate the Time-Difference-of-Arrival of a signal arriving at two microphones with some delay. This is used in the EPO-4 project in Q4.

The size of the practicum is planned at 2 EC (56 hours), consisting of 6 lab sessions (24 hours) and 8 self-study sessions (32 hours), used for preparation of the lab sessions and for report writing.

Brightspace is the learning platform for the course. Before the start of the practicum you have to enroll yourself into one "working team" of two students. Each working team falls into one of two possible scheduling categories: "A-teams" and "B-teams".

The lab sessions take place at the Tellegenhall, although if you have a PC with Matlab, loudspeakers and a microphone, you could do most of the exercises at home.

## 1.2 TIME SCHEDULE AND DEADLINES

The practicum has 6 scheduled lab mornings in Q3. Interleaved with these, there are preparation/homework mornings that you have to schedule yourself. Please remember: before the start of the practicum you have to enroll yourself via Brightspace into one "working team" of two students.

- The “A-teams” have scheduled lab mornings on Monday 8:45–12:45.
- The “B-teams” have labdays on Tuesday 8:45–12:45.
- The interim report will be discussed during subsequent lab sessions in week 4 or week 5.
- At the end, an extra session is scheduled for discussion of the final report.

Visit the Brightspace page of the course for the detailed working schedule.

## 1.3 FACILITIES

The labs are carried out at the Tellegenhall facilities. During the assigned weeks, a team will have 1 morning scheduled access to the lab, and on the other morning is expected to work on the project at home.

The following support is available:

- *Student assistants*; they are your primary help.
- *Coordinators*; they have limited availability and occasionally show up in the lab. Coordinators will grade your reports.

Visit the Brightspace page of the course for contact information.

## 1.4 REPORTS

The practicum outcome is documented in a midterm report (week 2) and a final report (week 8).

The reports must be handed in before 1:00 PM on the deadline (refer to Brighspace for the detailed schedule). The reports are to be uploaded into the respective working team submission folder.

Please let the filename of your report start with the group number, include the last names of your group, and send a single pdf (**not** doc), which includes everything.

Reports are *short*, with to-the-point answers to the questions, a graph, a brief explanation of an experiment/simulation. Use a simple cover page, skip introductions, don’t repeat the text of the manual, and simply give answers to the requested tasks as you would do on a written exam. The items to report on are indicated in the text (”report”). Provide an appendix with all Matlab code used to generate the graphs.

Items to report on are indicated in the text by “(report *n*)”. Use the same numbering (as section numbers) in your report.

Although the report is supposed to be compact, this does not mean sloppy.

Answers to most assignments typically consist of a graph and a discussion on the graph, sometimes preceded by a derivation. The discussion consists of two parts:

– Description of the graph: "Figure xx shows $\cdots$". Also mention parameter values and simulation conditions where needed, so someone else is able to redo the graph.

– Discussion on the result: does it make sense? (E.g., if you are asked to design a low pass filter meeting some specs, are the specs met?)

The practicum is done in groups of 2. You are not allowed to copy work by other groups.

## 1.5 GRADING

The grading is based on two aspects:

- *Technical:* Do you show understanding of what you are doing?
  - Is the result correct?
  - If results are sub-optimal, unexpected or clearly wrong, do you remark this? I.e., properly motivate your results.
- *Presentation:* Particular emphasis is placed on the presentation of the graphs:
  - Do all graphs have labels on the axis, a title, and legends (on the used line types) where needed.
  - Are the graphs readable and do they clearly show the "interesting" part (i.e. properly zoom in where needed).

  Also, the Matlab code in the appendix should show some professionality:
  - Suitable comments so someone can quickly understand what is done
  - Description at the beginning to define what the function is supposed to do, what is input and what is output.
  - Include author name and date (and perhaps revision history) at the beginning of each function.

The interim report will be corrected but not graded. This is meant to filter out the most common mistakes.

The final report is graded with Pass/Fail. You need a Pass to enter the EPO-4 project and to obtain a valid grade for EE2T11.

In some cases the final report is graded with a conditional pass. In that case you need to update your report (within a few days) to make it pass, in particular regarding some essential Matlab code which you need for the EPO-4 project.

In case of a fail, you are barred from entering EPO-4 and we will set a new deadline somewhere in Q4 at which you have a chance to hand in an improved report (so that you can still obtain a grade for EE2T11).

## 1.6 FUTHER READING

The book by Chaparro has an introduction to Matlab in relation to Signals and Systems (see Chapter 0).

L. F. Chaparro, *"Signals and Systems using MATLAB"*, Academic Press, 2011.

If you would like a slower-pace Matlab tutorial on Signals and Systems, we recommend the following book:

J.R. Buck, M.M. Daniel, and A.C. Singer, *"Computer explorations in signals and systems using Matlab"*, Prentice Hall, 2002.

### Hints

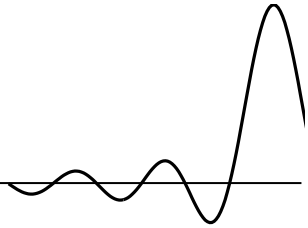
- When things get more difficult, split your Matlab code up into separate functions. Make separate scripts to start a test (`test.m`), which will call a script for data generation (`datagen.m`), run some functions on the data, and calls a script for showing plots (`show.m`).
- At some point you make measurements. Store the data so you can later on run different algorithms on it (instead of repeating the measurements each time the algorithm is changed.)
- Debug every function separately using simple test inputs for which you know the outcome. Don't write code, put it all together and assume that it works right away.
- Make sure your Matlab plots are readable, with a sufficiently large font size. A useful command is

  ```
  set(gcf,'PaperPosition',[0 0 5 4]);
  ```

  which you should give before saving the plot. This sets the printsize of the plot to $5 \times 4$ inch, smaller than the default size of 8 inch, and Matlab makes sure all fonts scale accordingly.

*Chapter* **2**

# LABDAY 1: CONVOLUTION



## Contents



We will study time-domain signals and apply basic filters via convolution. You have seen the definition of a convolution, but what does it really do? And how can you interprete the impulse response of a digital filter?

**Learning objectives** Basic filtering in time-domain, insight in convolution, correlation as a convolution.

**Preparation** Read the chapter so you know what to expect.

**What is needed**

- PC with a built-in loudspeaker and microphone;
- Matlab audio test signals, typically obtained using `load gong`, `load handel`, `load train`. See also the separate directory on Brightspace with some other test signals.

## 2.1 CONVOLUTION

In this exercise, we look at the effect of a convolution on a time-discrete signal $x[n]$. You have learned that an LTI system (filter) is described by an impulse response $h[n]$, and the output of the filter is written as $y = h * x$, or often (mathematically not entirely correct) $y[n] = h[n] * x[n]$, which is defined as

$$y[n] = \sum_{k=-\infty}^{\infty} h[k]x[n-k]\,.$$

If the filter has finite impulse response (FIR) and is causal, then $h[k]$ is nonzero only for $k = 0, \cdots, N_h - 1$, where $N_h$ is the "length" of the filter, and this becomes

$$y[n] = \sum_{k=0}^{N_h-1} h[k]x[n-k] \,.$$

The question is: what does this equation mean? One interpretation is that we can write

$$y[n] = h[0]x[n] + h[1]x[n-1] + \cdots + h[N_h-1]x[n-N_h+1]$$

Thus, the response $y[n]$ consists of scaled and delayed copies of $x[n]$. The delayed copies can be interpreted as echo's of the original signal $x[n]$.

In Matlab, the available samples of $x[n]$ are stored in a vector **x**; obviously we can take only a finite length sequence here. Standard Matlab indexing of a vector starts with $n = 1$, thus, a vector **x** is interpreted as

$$\mathbf{x} = [x[1], x[2], \cdots, x[N_x]]$$

But suppose we need a different time range? The only option is to define, along with **x**, a time-index vector **n** that in the above case is

$$\mathbf{n} = [1, 2, \cdots, N_x]$$

which in Matlab you would define as `n=[1:N_x]`, but in other cases could be $\mathbf{n} = [0, 1, \cdots, N_x - 1]$ or something else. When asked to plot a time-series specified in this way using a "stick" figure, you would use

```
stem(n,x)
```

If you have a large number of samples, it will be more clear to use simply `plot(n,x)`.

In many cases, a discrete time-domain signal is obtained by sampling an analog time-domain signal, at a rate determined by the sampling frequency $F_s$ in hertz (or samples per second). The corresponding sampling period is $T_s = 1/F_s$, in seconds. For example, if $F_s = 40$ kHz, then $T_s = 25\,\mu$s. You can plot the time-series with the proper time-axis using `plot(n*Ts,x)`.

- Consider a simple transformation of $x[n]$ such as $y[n] = x[n-2]$ or $y[n] = x[-n]$ (a time-reversal of the signal). Before you can plot this signal, you have to figure out the correct time index sequence **n**. For these two cases, what is it?

  Define a test signal **x** with time index vector **n** and try it in Matlab! (Plot the results.)

Matlab functions for convolution are

```
y = conv(x,h)
y = filter(b,a,x)
```

The first form corresponds to an FIR filtering where the FIR filter coefficients $h[n]$ are specified (stored in a vector **h**, also finite length). For an FIR filter, these coefficients are equal to its impulse response. The second form corresponds to $Y(z) = X(z)\frac{B(z)}{A(z)}$, and the filter coefficients of $B(z)$ and $A(z)$ are specified:

$$Y(z) = X(z)\frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \cdots + a_N z^{-N}} \quad \Leftrightarrow \quad \sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

If $A(z) = 1$, or $\mathbf{a} = 1$, you obtain an FIR filter $H(z) = B(z)$.

Due to the convolution, signals generally get longer. With `conv`, the standard option is that the tail part is included, whereas with `filter`, the signal **y** has the same number of entries as **x**, hence the tail is truncated.

Note that both functions do not accept a parameter that defines the time index. You will have to keep track of that yourself.

- If **x** has $N_x$ samples and **h** has $N_h$ samples, how many samples will result in the convolution?
- If **x** has time-index vector $\mathbf{n}_x = [a, a+1, \cdots, b]$ and **h** has time-index vector $\mathbf{n}_h = [c, c+1, \cdots, d]$, show that the proper time-index vector for **y** is $\mathbf{n}_y = [a+c, \cdots, b+d]$. Is this consistent with your answer on the first bullet?

Let's continue with the following exercise.

- Generate and plot a simple signal $x[n]$:

```
x = [1 2 1 zeros(1,10)];
subplot(311)
stem(x)
```

- Generate and plot the impulse response of a simple (LTI, discrete time) system:

```
h = [zeros(1,6) -0.5];
subplot(312)
stem(h)
```

- Compute the output signal of the system, $y = x * h$:

```
y = conv(x,h);
subplot(313)
stem(y)
```

How do you interprete the result?

- Repeat with

```
h = [1  zeros(1,5)  -0.5];
```

and with

```
h = [1  0 0 0  0.5  0 0 0  -0.5];
```

Note how the convolution is built step by step. For each nonzero tap of the impulse response $h[n]$, the signal $x[n]$ is delayed and scaled accordingly, and the result is added to the output $y[n]$. The nonzero taps of the impulse response can be interpreted as echo's, or multipath reflections. (Usually, the segments that are being added overlap.)

- The convolution operator is linear, distributive, associative, and the order of the operands can be reversed:

$$\begin{aligned}(h_1+h_2)*x &= h_1*x+h_2*x\\ (h_1*h_2)*x &= h_1*(h_2*x)\\ h*x &= x*h\end{aligned}$$

Verify these properties using `conv`.

- Consider the first-order IIR system defined by

$$H(z)=\frac{1}{1-az^{-1}}\,,\qquad |a|<1\,. \tag{2.1}$$

The filter can be implemented in Matlab using `y = filter(1,[1 -a],x)` where **x** is the input sequence and **y** is the output sequence (remember it is truncated by Matlab to have the same length as **x**). For $0<a<1$, this is a simple lowpass filter, and for $-1<a<0$ it is a highpass filter.

The impulse response truncated to length $N$ is obtained as `h = filter(1,[1 -a],[1 zeros(1,N-1)])`.

  - (report 1) Plot the impulse response of this filter for $a=0.95$ and $a=-0.95$. How does this filter change an impulse? How does it change a more general input signal, such as a step, `x = ones(20,1)`?

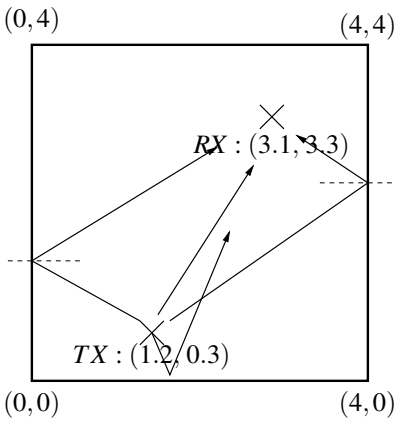
    How can you see from the impulse response or the step response that the filter acts as a lowpass (i.e., averages an input signal) or as a highpass (i.e., magifies differences in the input samples)?

Make sure you provide labels on the horizontal and vertical axis of the plot (using `xlabel` and `ylabel`), and provide a title (using `title`). If required, zoom in on the interesting part of the plot (you can do it manually on the graph, or set this in a script using `axis`).

## 2.2 ROOM CHANNEL IMPULSE RESPONSE

We will be working with audio signals. Therefore, it is important to understand how to interprete the impulse response of an audio channel.

Consider a room as follows:



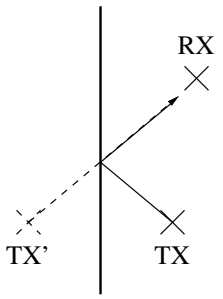

Assume that the walls of the room are perfect reflectors (like mirrors, rather than a diffuse scatterer). During propagation, the signal is damped as function of the propagated distance $r$. The speed of sound varies as function of temperature, humidity, and air pressure, but let us assume it is $c = 340$ m/s. For the attenuation $\alpha(r)$, assume that it is a factor

$$\alpha(r) = \frac{\beta}{r}$$

where $\beta$ is the damping over a reference distance of 1 meter.

- (report 2) Using this model, and with the help of Matlab, can you make a plot of the channel impulse response measured at the receiver? (Limit yourself to the first two reflections; choose some suitable $\beta$; do not spend more than 30 min on this.)

  *Hint:* to compute the effect of a single reflection, you can "mirror" the transmitting source into the wall. This will help you in computing the distances. How many virtual sources do you have after 1 reflection? And after 2? What are their locations? First compute vectors $(\mathbf{x}, \mathbf{y})$ of virtual locations, then compute their distance to the receiver. If you look in detail, you will find that some distances are repeated while some of the corresponding reflections are "not physical". You can skip the investigation of those details.




Note that the above assumptions are very idealistic. In practice, sound waves are partially reflected off many objects, as function of the wavelength and the size of the object. For a signal of 10 kHz, the wavelength is about 3.4 cm, for 100 Hz, it is 3.4 meters. Objects smaller than the wavelength are usually "transparent". Thus, the actual propagation channel is also frequency dependent. Further, objects may partially absorb a sound wave (as function of frequency), and may act as diffuse scatterers.

## 2.3 AUDIO SIGNALS

Now that we understand convolution and audio signal propagation (a bit), we repeat the convolution exercise on an audio signal.

The following Matlab functions could be helpful.

```
[x, Fs, nbits] = wavread('file.wav');
```

Reads a WAV soundfile into a vector **x**; herein is `Fs` the sample rate and `nbits` the number of bits per sample.

```
soundsc(x,Fs);
```

Plays a vector (signal) on the loudspeakers. `Fs` is the sample rate.

To see how these functions work, use the Matlab help function. To see which variables are defined, use `whos`.

Some audio signals are available in the Matlab audio toolbox; try `load gong`, `load handel`, `load train` (Note: these data files define a vector **y**, not **x**. Also the sample rate `Fs` is loaded).

Other test signals are provided on Brightspace (speech signals and audio clips): `T4_ca.wav`, `T5_ha.wav`, `T6_tb.wav`, $\cdots$. These are sampled at 48 kHz, which makes the files quite large and the signals a bit harder to handle.

More recent versions of Matlab are object-oriented. Here, audio is stored in an object `p`, along with meta-data such as the sample rate:

```
p = audioplayer(y,Fs);    % make audioplayer object
play(p)                   % play on loudspeaker
                          % (matlab continues while playing)
get(p)                    % show the metadata
```

**Tasks**

- Read the sound file `train.mat` into a signal **x**. What is the sample rate $F_s$?
- Play the signal on the loudspeaker.
- Generate an impulse response that has a number of reflections:

  ```
  h = [1  zeros(1,Fs/10) 0.9 zeros(1,Fs/30) 0.8];
  ```

- Do a convolution $y = x * h$; listen to the result.
- "Reverberation" is the effect of a group of echos, closely together, and exponentially decaying due to multiple reflections (e.g., multiple reflections in a tunnel).

  Try to create a reverberation effect using a first-order IIR filter of the form (2.1):

  ```
  a = -1/(1+20/Fs);
  hh = filter(1,[1 a],[1 zeros(1,Fs/5)]);
  h2 = [hh  zeros(1,Fs/10) 0.7*hh];
  plot(h2)
  y = conv(x,h2);
  ```

  Listen to the result. Does it sound like you are in a tunnel?

## 2.4 CORRELATION

For a given time-series $x[n]$, consider the convolution of $x[n]$ with its time-reversed series $x[-n]$:

$$r[n] := x[n] * x[-n] \qquad \Leftrightarrow \qquad r[n] = \sum_{k=-\infty}^{\infty} x[k]x[n+k]$$

The second expression shows that $x[k]$ is pointwise multiplied with itself (after a delay of $n$ samples), and then summed. This is interpreted as a *correlation* of $x$ with itself, or an *autocorrelation*, and the delay $n$ is known as the *correlation lag*. (If $x[n]$ is complex, we would correlate with $x^*[-n]$. Also, what is shown

here is a deterministic correlation; for random signals the definition uses an expectation operator, as you will see in EE2S31 *Signal Processing*. Actually the correct definition of the deterministic correlation has a scaling by $1/N$, where $N$ is the number of terms in the summation, so that it converges to the stochastic correlation for large $N$.)

- Create a simple signal $x[n]$:

  ```
  x = [1 2 -1 zeros(1,10)];
  ```

  Compute its autocorrelation:

  ```
  r = conv(x, fliplr(x))
  ```

  Herein, the function `fliplr` reverses the entries of a row vector (similarly, `flipud` does this for a column vector).

  Plot the result. What is the correct time-axis?

Correlation functions such as $r[n]$ have many interesting properties and applications. An important one is that $r[n]$ can be interpreted as showing how well a signal $x$ matches a delayed version of itself. The best match is obtained for a lag $n = 0$, and $r[0]$ is interpreted as the *energy* in the signal. Some properties are:

$$\begin{array}{lrcl} (1) & r[0] & = & \sum_k |x(k)|^2 \\ (2) & r[-n] & = & r[n] \\ (3) & |r[n]| & \leq & r[0] \end{array}$$

The last property is proven from the general property of the inner product between two vectors $\mathbf{x}$ and $\mathbf{y}$, for which $|\mathbf{x}^T\mathbf{y}| \leq \|\mathbf{x}\|\|\mathbf{y}\|$.

- Verify these properties from the plot that you made of $r[n]$. Be sure to use the correct time-axis.

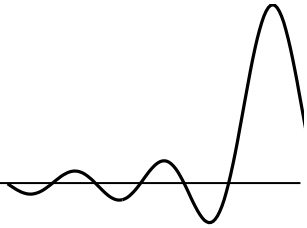
Suppose we transmit the signal $x[n]$ over an unknown FIR channel $h[n]$ and measure the result at the receiver, $y[n] = x[n] * h[n]$. To esimate $h[n]$ we determine $z[n] = y[n] * x[-n]$, this is known as a *matched filter*.

- Give a formula for $z[n]$ in terms of $r[n]$.
- Take $h[n] = \delta[n-2]$, i.e. `h= [0 0 1 zeros(1,10)]`, and compute the received signal $z[n]$. Plot $z[n]$ (make sure you use the right time-axis).
- Next, take $h[n] = \delta[n-2] + 0.5\delta[n-3]$ and plot $z[n]$.

What you should observe is the following. We know $z[n] = h[n] * r[n]$, and if the autocorrelation function $r[n]$ has a sharp peak for $n = 0$, then $z[n] \approx h[n]$. In particular, the delay in the channel is observed from the location of the first peak in $z[n]$. The results are best for signals with "good" autocorrelation properties, i.e., signals with a strong peak at $r[0]$ and low side lobes. We will further investigate the matched filter during Labday 3 and 4.

*Chapter* **3**

# LABDAY 2: FREQUENCY DOMAIN AND THE FOURIER TRANSFORM



## Contents



We look at the relations between time domain and frequency domain on actual signals by applying the DFT (or FFT).

**Learning objectives** Applying the DFT, frequency-domain and time-frequency domain plots, filtering in frequency domain.

**Preparation** Read the chapter so you know what to expect.

**What is needed**

– PC with a built-in loudspeaker and microphone;

– Matlab audio test signals, typically obtained using `load gong`, `load handel`, `load train`.

– Matlab audio test signal from Brightspace: `touch.mat`.

### 3.1 DISCRETE-TIME FOURIER TRANSFORM

You have seen the Discrete-Time Fourier Transform (DTFT) in the *Signals and Systems* course. Recall its definition,[1]

$$X(\omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}. \tag{3.1}$$

Note that $X(\omega)$ is periodic with period $2\pi$. Thus, it is sufficient to consider $0 \leq \omega \leq 2\pi$, or equivalently, $-\pi \leq \omega \leq \pi$. You can interprete this definition as "correlating" an input sequence $x[n]$ with a complex exponential $e^{j\omega n}$, and seeing how well they match for a frequency $\omega$.

Since $X(\omega)$ is complex, if it is asked to plot "the spectrum" you would usually plot the amplitude spectrum, $|X(\omega)|$, and sometimes the phase spectrum, $\angle(X(\omega))$.

The DTFT has two problems that prevents its use in Matlab:

1. We cannot compute an infinite sum, and anyway we usually have only a finite sequence **x**, say $[x[0], \cdots x[N-1]$. In this case, we would need to replace the infinite summation in (3.1) by a finite summation over the provided $N$ samples.

2. We cannot plot a function of a continuous parameter $\omega$. We would have to take a number of frequency-domain samples $\omega_k$. If we take $N$ samples uniformly on the interval $0 \leq \omega \leq 2\pi$, we obtain $\omega_k = (2\pi/N)k$, for $k = 0, \cdots, N-1$.

This leads to the definition of the Digital Fourier Transform (DFT),

$$X[k] := X(\omega_k) = \sum_{0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}, \qquad k = 0, \cdots, N-1\,. \tag{3.2}$$

In this definition, an equal number of frequency-domain samples is taken as we have samples in time-domain. The inverse of this transformation (the IDFT) also exists and it is very similar to the DFT:

$$x[n] = \frac{1}{N}\sum_{0}^{N-1} X[k]e^{j\frac{2\pi}{N}kn}, \qquad n = 0, \cdots, N-1\,. \tag{3.3}$$

Compared to the DTFT (desired but not computable), taking the DFT has two effects:

1. The finite number of samples of $x[n]$ taken into consideration make that the DFT is not the "true" spectrum, it is an approximation.

2. The sampling in frequency domain causes aliasing in time domain. However, this effect is only observable if you start with an infinite length sequence $x[n]$, compute the DTFT at samples $\omega_k$ (i.e., the true spectrum at these frequencies), and then apply the IDFT. The resulting sequence (call it $\tilde{x}[n]$) is periodic as seen from (3.3), and $\tilde{x}[n] = \sum_r x[n - rN]$.

[1] We adopt here the notation where $\omega$ is used for frequencies of discrete-time signals, and $\Omega$ (or $F = \Omega/(2\pi)$ in Hz) is used for frequencies of continuous-time signals.

These effects will be explored in more depth in the EE2S31 *Signal Processing* class. For the moment, suffice it to say that the DFT is exact if $x[n]$ has finite length $N$ or is periodic with period $N$, and otherwise it is an approximation that gets better for larger $N$.

In Matlab, the DFT has an efficient implementation as the Fast Fourier Transform (FFT), and if $\mathbf{y}$ is a vector with $N$ entries, we write `Y = fft(y)`, where $\mathbf{Y}$ also has $N$ entries. Note that the result is complex; to plot it, take the absolute value using `abs(Y)`. The corresponding frequency axis is obtained as `Omega = [0: 2*pi/N : (N-1)*2*pi/N]`. Note that the last entry in this vector is not $2\pi$ but slightly less. Sometimes we use "normalized frequencies" $f = \omega/(2\pi)$, or `f = [0: 1/N : (N-1)/N]`. These run from 0 to slightly less than 1. Frequency $f = 1$ corresponds to $\omega = 2\pi$, which corresponds to $\omega = 0$ due to periodicity. The highest frequency is $f = 1/2$ (if that is a sample point of the DFT).

If a signal was obtained from sampling with sampling frequency $F_s$, you can also plot in terms of the original "analog" or "real" frequencies by scaling $\omega$ to $F = \omega F_s/(2\pi)$ or $F = fF_s$. This maps the angular frequency $\omega = 2\pi$ or normalized frequency $f = 1$ to the real frequency $F = F_s$.

The IDFT in Matlab is called `ifft`. Note that the result is not always real-valued, even if you would expect that, due to round-off error. In that case, round off to real using `real(ifft(Y))`.

- Generate some simple signals $x[n]$ and plot their amplitude response: take

  ```
  x = [ones(1,N) zeros(1,50-N)];
  ```

  for $N = 1, 2, 4, 8$. Make sure you get the frequency axis right.

  Recall from the *Signals and Systems* class that these functions are "Dirichlet functions", similar to sinc functions,

  $$X(\omega) = e^{-j\omega N/2} \frac{\sin(\omega N/2)}{\sin(\omega/2)}.$$

For time-discrete signals, the spectrum is periodic in $2\pi$ (or $F_s$). Often, you would plot the range $\omega \in (-\pi, \pi]$, or $F \in (-\frac{1}{2}F_s, \frac{1}{2}F_s]$. The function `fftshift(X)` can be used to rearrange the samples. Unfortunately, computing the corresponding frequency axis is a bit messy. If $N/2$ is integer then $\omega = -\pi$ is a sample point of the DFT, and we use `Omega = pi*[-1 : 2/N : 1-1/N]`. If $N/2$ is not integer, then we use `Omega = pi*[-1+1/N : 2/N : 1-1/N]`. Note that $\omega = 0$ is always a sample point.

## 3.2 PLOTTING A TRANSFER FUNCTION

To plot the transfer function of a filter $H(z)$, we have two options.

If the filter is FIR, then $H(z) = h_0 + h_1 z^{-1} + \cdots + h_{N-1} z^{-(N-1)}$. Thus, we can define a vector $\mathbf{h}$ that contains the impulse response, $\mathbf{h} = [h_0, h_1, \cdots, h_{N-1}]$. Next, apply the DFT to $\mathbf{h}$: `H = fft(h)`. Define the corresponding frequency axis, `Omega = [0: 2*pi/N : (N-1)*2*pi/N]`. Plot the result using `plot(Omega,abs(H))`.

If the filter is rational, we could compute the impulse response (it is infinitely long), truncate it at some length, and apply the DFT. Alternatively, suppose $H(z) = \frac{B(z)}{A(z)}$ and we have vectors of coefficient **a** and **b**. We can define the frequency axis at $N$ values as `Omega = [0: 2*pi/N : (N-1)*2*pi/N]`, define the corresponding values of $z$ as `Z = exp(sqrt(-1)*Omega)` and evaluate $H(z)$ for these values using `polyval`. But all this is done using the Matlab command

```
[H, Omega] = freqz(b,a,N)
```

If you need just the plot, you can also simply say `freqz(b,a)` and the function will make it for you (although the amplitude spectrum is shown in dB scale).

- Consider the first-order filter (2.1),

$$H(z) = \frac{1}{1-az^{-1}}, \qquad |a| < 1.$$

  Plot the amplitude and phase response of the filter, for $a = 0.95$ and $a = -0.95$. Make sure you get the frequency axis right. For the phase response, use the Matlab command `angle`.

A filter is called linear phase if it is possible to write $H(\omega) = A(\omega)e^{-j(\alpha\omega+\beta)}$, where $A(\omega)$ is real-valued. A simple example is a delay, $H(z) = z^{-1}$, for which $\alpha = 1$ and $\beta = 0$. Linear-phase filters are often used for selecting a certain frequency band, e.g. an ideal lowpass filter, and we want the phase to be linear at least in the passband because the filter will then act as a delay for the passband and will not distort the pulse shape of signals in the passband. The slope of the phase response shows the delay $\alpha$.

Linear-phase filters must be FIR and satisfy a symmetry property:

$$h[n] = \varepsilon h[N-n], \qquad \varepsilon = \pm 1$$

- Consider a linear-phase filter with impulse response

$$\mathbf{h} = [1,2,3,2,1]$$

  Plot the amplitude and phase response of the filter. Is this indeed linear-phase?

  Note that the amplitude response becomes zero at some frequency, indicating that there is a "zero" on the unit circle. The zeros of $H(z)$ are found using `roots(h)`. Plot the zeros using `zplane(roots(h))`.

A filter is called an allpass filter if it satisfies $|H(\omega)| = 1$. Thus, only its phase response is of interest. A simple example is a delay, $H(z) = z^{-1}$. These filters are used to correct the phase response of other filters, and they play an important role in the practical realization of filters, especially highly selective ones. (They play a similar role as unitary matrices do in linear algebra.)

All rational allpass filters are of the form

$$H(z) = \frac{a_N + a_{N-1}z^{-1} + \cdots + z^{-N}}{1 + a_1 z^{-1} + \cdots + a_N z^{-N}}$$

- Consider a first-order allpass

$$H(z) = \frac{a - z^{-1}}{1 - az^{-1}}.$$

Compute and plot the amplitude and phase response (take $a = 0.95$). Check the locations of the poles and zeros using `zplane(roots([a -1]),roots([1 -a]))`.

## 3.3 A SIGNAL IN TIME DOMAIN AND FREQUENCY DOMAIN

In Sec. 2.3, we constructed several time-domain audio signals **x** (e.g., `load train`). Let's see how this signal looks like in the frequency domain.

- (report 3) Plot the time domain signal. Also make sure the labels on the x-axis are correct, i.e., shown in seconds (define a time axis vector **t** of the same length as **x** taking into account the sample rate $F_s$, use `plot(t,x)`, also plot axis labels e.g. using `xlabel('time [s]')`).

  Compute the frequency domain signal using the FFT. Also, plot the correct frequency axis in hertz.

  Can you explain the 'symmetry' in the plot? Does it make sense to plot only the positive frequencies, and how is this done?

Several Matlab commands exist to compute and plot the spectrum of a signal, e.g., `pwelch` and `spectrum.psd`. But it is recommended not to use these until you develop an understanding of how this works. (The MSc course EE4C03 *Statistical Digital Signal Processing* will provide the required theory.)

## 3.4 TIME-FREQUENCY PLOT

The 'train' signal is not stationary: its frequency contents changes over time. This is true for many signals. Thus, it does not make sense to record seconds or hours of an audio signal, and then to take the DFT of the complete sequence. Instead, you would usually split the signal into short segments (e.g., speech signals are usually split into 20 ms segments before audio coding), then do the DFT on each segment, and plot the results into a "time-frequency plot". This is a plot with on the x-axis time, on the y-axis frequency. The amplitude (or intensity) is shown using color.

- Take the 'train' signal, and split it into non-overlapping segments of approximately 20 ms (how many samples are in one segment)? Store the segments into a matrix **X**, one column per segment. You could use the command `X = reshape(x,Fbins,Tbins)` for this.

  Do the DFT on each column separately (in Matlab, you can directly say `Y = fft(X)` as the FFT function is applied column-by-column).

  Define a new time axis vector **t** with steps of 20 ms. Also define a frequency axis vector **f**, taking the sample rate and number of samples per column into account.

- (report 4) Plot the time-frequency plot using `imagesc(t,f,abs(Y))`. Also indicate the correct x-labels and y-labels. Make sure the total signal duration matches with that of previous time-domain plot, and that the frequency axis matches that of the previous frequency-domain plot.

For better time-resolution, people often take overlapping segments. With maximal overlap, the segments are shifted only 1 sample from each other. But statistically, this makes little sense. More often, an overlap of 50% is taken.

It will be useful to put your Matlab code in a function, so you can quickly reuse it for other signals in the future.

### 3.5 ZERO PADDING

Sometimes, the number of samples in a time-domain signal **x** is not very large. Using the DFT, we obtain the same number of samples in the frequency domain as in time domain, and in these cases the resolution is not high. Or, **h** might be the impulse response of a short FIR filter, and our aim is to see the frequency response of the filter in high resolution.

In these cases, you can use the standard trick of "zero padding". Essentially, you extend **x** (or **h**) with a lot of zeros. Then, apply the DFT to this augmented sequence. In Matlab, the same result is obtained by saying `fft(x,N)` where $N$ is larger than the number of samples in **x**.

The theory says that this will nicely interpolate the original samples of the DFT spectrum obtained from the short (un-padded) sequence. This is because the DFT corresponds exactly to a sampled version of the DTFT in case $x[n]$ has a finite length of at most $N$, and zero padding does not change the length of the nonzero part of $x[n]$. You will see the theory in detail later in the course EE2S31 *Signal Processing*, but let's see how this works in practice.

- Let us generate a simple filter that returns a signal and two weaker echos:

```
h = [1 0 1/2 0 1/4]
```

- (report 5) Do the DFT of $h[n]$ and show the amplitude spectrum, but only show the available samples, using `plot(f,abs(H),'o')`. Here, `f` is the frequency axis. Since you don't have a sample rate, you should use 'normalized frequencies', `f = [0: 1/N : (N-1)/N]`, where $N = 5$.
- (report 6) Now extend **h** with $5 \cdot 5 = 25$ zeros to 6 times its original length, recompute the DFT, recompute the frequency axis, and show the amplitude spectrum of the extended sequence in the same plot (first apply `hold on` to keep the previous plot; use a different plot marker, e.g., `'+'` or `'x'`). Do you obtain interpolation? (This means that every 6th sample should coincide with a sample of the previous plot.)

If $N$ is not very large then usually we would zero-pad to $N = 256$ or more, compute the DFT, and plot the spectrum using a continuous line (rather than its individual samples as we did here).

## 3.6 THE CONVOLUTION PROPERTY

We would like to demonstrate the convolution property:

$$y[n] = x[n] * h[n] \quad \Leftrightarrow \quad Y(\omega) = X(\omega)H(\omega)$$

which is true if you talk about the DTFT, but not exactly true if we use the DFT: in that case the property is only valid for a circular convolution (also called a cyclic convolution), as we will study in detail in the course EE2S31 *Signal Processing*. This is because the sequences in the DFT have a finite length. Also, to show this property, the vectors representing $Y(\omega)$, $X(\omega)$ and $H(\omega)$ should be of equal length, i.e., contain an equal number of samples $N$, or else you cannot pointwise multiply these vectors.

- Let **x** be the 'train' signal. Choose the filter impulse response $h[n]$ as before, and do a convolution $y = x * h$.

  For this, read the help of the Matlab `c = conv(a, b, shape)` function, in particular regarding the `shape` options. Apparently there is no Matlab function for a circular convolution.

  To see the convolution property, we should do zero padding on **x** and **h** to make them have the length of the vector **y** that results from the standard `conv` function, i.e., if **x** has length $N_1$ samples and **h** has length $N_2$ samples, we will have **y** having $N_1 + N_2 - 1$ samples.

  We can do the zero padding on **x** and **h** after we computed `y = conv(x,h)`, or we can do the zero padding beforehand and obtain the desired **y** by choosing the right `shape` option of `conv`.

- (report 7) Compute the DFT of the extended **x**, **y** and **h** of equal lengths, and show the resulting amplitude spectra. Does the convolution property hold? (We would expect that $X(\omega)H(\omega)$ is identical to $Y(\omega)$.)

## 3.7 ASSIGNMENT: TELEPHONE TOUCH-TONE DETECTION

If you have an 'old' fixed-line telephone, you may have heard dialing sounds when sending a telephone number: this is the touch-tone signal. Officially these are called dual-tone multi-frequency (DTMF) signals. Each digit is related to a pair of frequencies that are simultaneously being transmitted for a short duration of time. Table 3.1 shows these frequencies in relation to the row and column of a digit in a dialing pad.

- (report 8) From Brightspace, download a data file `touch.mat`.[2] This file contains four dialing signals: `x1, x2, x1hard, x2hard`. The sample rate is $F_s = 8192$ Hz. You can listen to the signals by typing `soundsc(x1,8192)`.

  The assignment is to detect the phone numbers from this file.

[2] File obtained from the Matlab repository for the book "Computer explorations in signals and systems using Matlab", J.R. Buck e.a.

**Table 3.1**. DTMF frequencies

| | | $F_{col}$ [Hz] | | |
|---|---|---|---|---|
| | | 1209 | 1336 | 1477 |
| $f_{row}$ [Hz] | 697 | 1 | 2 | 3 |
| | 770 | 4 | 5 | 6 |
| | 852 | 7 | 8 | 9 |
| | 941 | $*$ | 0 | # |

*Hints:* this is a nonstationary signal. Use a time-frequency plot to plot the transmitted frequencies as function of time. Use the correct frequency axis so that you know which frequencies are transmitted at each time. *Which resolution do you need in frequency domain to distinguish the frequencies? From this, calculate how many samples you need at least in frequency domain. Show this calculation in your report.* What is the resulting resolution in time domain and is that acceptable? Show the time-frequency plots (properly zoom in) and the detected phone numbers in your report. It is OK to do the detection "by hand".

## 3.8 HOMEWORK DAYS — MIDTERM REPORT

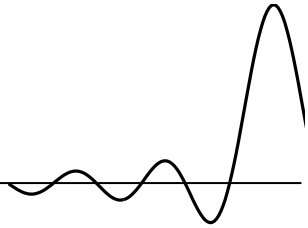
Two homework days are scheduled before the next lab day. Use one homework day to prepare the report for the previous two labdays, and hand it in before the deadline of week 3. Use the second homework day to prepare for the next labday: you have to read quite a lot of text:

- Channel estimation (Chapter 4);
- Deconvolution (Appendix A);
- Matrix inversion, the pseudo-inverse, the SVD, and the condition number (Appendix B).

The two appendices provide background information that is not immediately needed but will improve your understanding of your results. To be able to understand the tutorials, you will have to be up-to-date on linear algebra, in particular matrix inversion and projections.

*Chapter* **4**

# LABDAY 3: CHANNEL ESTIMATION



## Contents



Suppose we transmit a known signal $x[n]$ over a communication channel, and measure the result $y[n]$. The channel acts as a filter, which we will assume to be linear and time-invariant. Therefore, the measured signal is a convolution of the transmitted signal by the channel impulse response $h[n]$, such that $y[n] = h[n] * x[n]$. Knowing the transmitted signal $x[n]$, can we recover the impulse response of the communication channel from $y[n]$? In this form, the channel estimation process is called deconvolution (or equalization), and generally it requires an inversion.

Indeed, if we do this in the frequency domain, we have

$$H(\omega) = \frac{Y(\omega)}{X(\omega)},$$

and we can recover $h[n]$ from an inverse DTFT. However, there are some complications to do this in the frequency domain. Can we use the DFT instead of the DTFT (which leads to sampling in the frequency domain)? Are there numerical problems with the above division (what if $X(\omega) = 0$)? Will $h[n]$ be causal? What about stability?

Alternatively, we can do the deconvolution in the time domain. Also here, there can be numerical problems. We have to choose the known signal $x[n]$ at the transmitter, and (with additive noise) the resulting channel impulse response estimates will be different due to differences in noise enhancement.

We will need channel estimation in the EPO4 project, where each toy car will transmit audio beacon signals, which are recorded by microphones at the corners of the field. From differences in the estimated channels, we will locate the car. We also want to be insensitive to interfering signals: if several beacons are transmitting audio signals at the same time, we only want to detect our own transmitting sequence.

In today's task, we will implement a channel estimation algorithm in Matlab. We will also consider what makes a good transmitting sequence. In Labday 4, we will then use the developed algorithm on signals transmitted by the audio beacon.

**Learning objectives** Basics of channel estimation and deconvolution, application of linear algebra (systems of equations, inversion)

**Preparation** Read the chapter in detail so you know what to expect. We will consider 3 methods for deconvolution. You will have to know the corresponding tutorials before you start the assignments.

Appendix A gives more background on deconvolution in time domain, and is best read before you start on section 4.1. We will use the SVD, an important tool in linear algebra which unfortunately is not covered in EE2M21 *Linear Algebra*; a brief tutorial is in Appendix B.

**What is needed**

– PC with a built-in loudspeaker and microphone;

– Matlab audio test signals

– From Brightspace: Matlab script '`toep.m`'.

## 4.1 CHANNEL ESTIMATION USING MATRIX INVERSION

Estimation of a propagation channel is fundamental in many applications. In radar it allows to estimate the distance of objects, in geophysics it represents the reflections at earth layers, in medical ultrasound it allows to form an image of a patient. In the EPO4 project we will use channel estimates to determine the distance of the toy car to each microphone at the corners of a field, this is further explored in Labday 4.

In subsequent sections, we will consider three methods for channel estimation: (1) via time-domain equalization, which leads to a matrix inversion, (2) the related Matched Filter, which approximates this matrix inversion (needed if the matrix is large), (3) frequency-domain equalization, which is in theory equivalent to time-domain equalization, but may be computationally more efficient.

Suppose we transmit a known sequence of pulses, i.e. a signal $x[n]$, commonly called a training sequence. The training pulses will be deformed by the channel, and we receive $y[n] = h[n] * x[n]$. Since we know $x[n]$, we can "invert" this signal using deconvolution, and recover $h[n]$.

Since we can design the signal, we could use a simple impulse, $x[n] = \delta[n]$, so that $y[n] = h[n]$. However, this method is very sensitive to noise or interference: there is no averaging at all. Also, in practice the amount of energy that can be transmitted in a single pulse is limited, so if we have time, we should send a periodic sequence of pulses (that will also allow for averaging). Therefore, our plan is to use a more interesting $x[n]$, and apply deconvolution to recover $h[n]$.

Appendix A discusses deconvolution in time domain using a matrix inversion technique. We will apply this theory to the problem of channel estimation. The matrix equation corresponding to the convolution

$y[n] = x[n] * h[n] = \sum_{k=0}^{L-1} h[k]x[n-k]$ is

$$\mathbf{y} = \mathbf{X}\mathbf{h} \quad \Leftrightarrow \quad \begin{bmatrix} \boxed{y[0]} \\ y[1] \\ y[2] \\ \vdots \\ \vdots \\ \vdots \\ y[N_y-1] \end{bmatrix} = \begin{bmatrix} \boxed{x[0]} & & \mathbf{0} \\ x[1] & x[0] & \\ x[2] & x[1] & x[0] \\ \vdots & x[2] & x[1] \\ x[N_x-1] & \vdots & x[2] \\ & x[N_x-1] & \vdots \\ \mathbf{0} & & x[N_x-1] \end{bmatrix} \begin{bmatrix} \boxed{h[0]} \\ \vdots \\ h[L-1] \end{bmatrix} \tag{4.1}$$

where the "box" indicates the location of time-index 0, $N_x$ is the length of the input sequence (subsequent samples are supposed to be zero), and $N_y$ is the length of the output sequence. Note that $\mathbf{X}$ has size $N_x + L - 1 \times L$ so that $N_y = N_x + L - 1$. $\mathbf{X}$ is always tall.

The channel estimate is obtained by taking a left inverse $\mathbf{X}^\dagger$ of $\mathbf{X}$, such that $\mathbf{X}^\dagger\mathbf{X} = \mathbf{I}$. We can usually take $\mathbf{X}^\dagger = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ where we assume that $\mathbf{X}^T\mathbf{X}$ is invertible.[1] This results in

$$\hat{\mathbf{h}} = \mathbf{X}^\dagger\mathbf{y} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{4.2}$$

In the tasks we study how this works.

### Tasks

- From Brightspace, download the file `toep.m`. It is a function to create an $N_y \times L$ Toeplitz matrix $\mathbf{X}$ from a sequence $\mathbf{x}$: `X = toep(x,Ny,L)`.
- Make a Matlab script or function `datagen.m` to generate four input test sequences $x_1[n], \cdots, x_4[n]$ as described below, and convolve each of them with a channel impulse response $h[n]$ to obtain measurement data $y_i[n]$.

  As test signals, we will try the following four possibilities:

  1. A "minimum-phase" sequence, represented by

     $$X_1(z) = 1 - \frac{1}{2}z^{-1} \quad \Leftrightarrow \quad \mathbf{x}_1 = [\cdots, 0, \boxed{1}, -\frac{1}{2}, 0, 0, \cdots]^T,$$

     This sequence has a causal stable inverse.
  2. A "maximum-phase" sequence,

     $$X_2(z) = 1 - 2z^{-1} \quad \Leftrightarrow \quad \mathbf{x}_2 = [\cdots, 0, \boxed{1}, -2, 0, 0, \cdots]^T,$$

     This sequence has a noncausal stable inverse.

[1] If this is not the case, then we have to use the SVD to define a more general pseudo-inverse. See Appendix B.

3. A sinusoidal signal ($N$ samples) followed by zeros,

$$x_3[n] = \begin{cases} \cos(\omega n), & n = 0, \cdots, N-1 \\ 0, & \text{elsewhere} \end{cases}$$

   Take e.g. $\omega = 0.2$. This signal cannot be inverted.

4. A random binary sequence with entries randomly selected from $\{-1, 1\}$: in Matlab `x_4 = sign(randn(N,1))`.

For each signal, make a vector $\mathbf{x}_i$ containing the impulse response coefficients starting at $x_i[0]$. Add zeros to the end of each signal to make them all of equal length $N_x = N$. For easier testing, you can initially take a reasonably small value, e.g., $N = 10$.

As channel, we will for the moment take a short "random" sequence, e.g., `h = [1, 2, 3, 2, 1]`, which has length $L = 5$. Taking simple numbers will make it easier to recognize if the Matlab functions give a reasonable result during debugging.

Generate the corresponding output signals (those that will be recorded by the microphone), i.e., $y_i[n] = h[n] * x_i[n], \quad i = 1, \cdots, 4$. In Matlab, we can use the function `conv` for the convolution. The length of each sequence is $N_y$. Verify that $N_y = N_x + L - 1$.

The outputs of the script `datagen.m` are the sequences $y_i[n]$ along with the input sequences $x_i[n]$.

- Make a Matlab function `ch1.m` which implements time-domain channel estimation via inversion using (4.2).

  The input of this function is the measured signal $y[n]$ and transmitted training signal $x[n]$; the output is the estimated channel $\hat{h}[n]$. Use the function `toep(x,Ny,L)` to generate the required Toeplitz matrix $\mathbf{X}$ of size $N_y \times L$. The matrix inversion $\hat{\mathbf{h}} = \mathbf{X}^{\dagger}\mathbf{y}$ is efficiently implemented in Matlab either as `hhat = X\y` or `hhat = pinv(X)*y`.

  As channel length $L$, you could take $L = N_y - N_x + 1$. However, in practice $N_y$ is not well defined (we record a response signal and just truncate it somewhere). Therefore, $L$ is not well defined. Extend your estimation algorithms to use an additional input parameter $\hat{L}$, which should be the length of the estimated channel. E.g., this parameter could be used to zero-pad or truncate $y[n]$ to the right length.

- Make a Matlab script `test1.m` wherein you apply the channel estimation algorithm `ch1.m` to each of the four input signals.

  Debug and test the channel estimation algorithm. How can you be sure you implemented the algorithm correctly? Apply to cases where you know the answer (either very simple input signals or very simple channels). The matrix inversion technique should give exact results in the noise-free case.

- In practice, we don't know $L$. Test for various "estimated" channel lengths. Is this a sensitive parameter?

- Exactly what happens in the case $X(z)$ is non-minimum phase (as it is for signal $x_2[n]$)? Does the inversion pose any problems?

If you compare to the theory of deconvolution in Appendix A, you will notice that there, we inserted an optional reconstruction delay $K$ to improve the estimation in case $\mathbf{x}$ is not minimum-phase (as is likely the case for arbitrary sequences). The problem was that we did not know if the equalizer $G(z)$ is causal. Here, our assumption (on physical grounds) is that the channel $H(z)$ is causal, hence we can model the impulse response using only $h[0],\cdots,h[L-1]$. There is no need to introduce a parameter $K$ for negative time indices.

## 4.2 INVERTIBILITY AND CORRELATIONS

In the matrix inversion step, we compute the pseudo-inverse $\mathbf{X}^\dagger = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$. In fact, `pinv` is implemented in Matlab using the singular value decomposition (SVD). It is relevant to consider the invertibility of $\mathbf{X}^T\mathbf{X}$. Not all training sequences $\mathbf{x}$ lead to an invertible matrix $\mathbf{X}^T\mathbf{X}$. Even if it is numerically invertible, a poorly invertible matrix can lead to large noise enhancements.

In the EE2M21 *Linear Algebra* course, you may have learned that eigenvalues give information on the invertibility of a matrix. The eigenvalues of $\mathbf{X}^T\mathbf{X}$ are the singular values of $\mathbf{X}$, squared. In Matlab, you can compute these using `svd(X)`. Appendix B gives a short tutorial on the SVD in connection to matrix inversion.

Consider $r[n] = x[n] * x[-n]$. This is the autocorrelation sequence of the input signal. It is a symmetric sequence, with a peak at $r[0]$ (see section 2.4). $\mathbf{X}^T\mathbf{X}$ can be interpreted as an auto-correlation matrix,

$$\begin{aligned}
\mathbf{X}^T\mathbf{X} &= \begin{bmatrix} x[0] & x[1] & \cdots & x[N_x-1] & & & \\ & x[0] & x[1] & \cdots & x[N_x-1] & & \\ & & \ddots & & & \ddots & \\ & & & x[0] & x[1] & \cdots & x[N_x-1] \end{bmatrix} \begin{bmatrix} x[0] & & \mathbf{0} \\ x[1] & x[0] & \\ x[2] & x[1] & x[0] \\ \vdots & x[2] & x[1] \\ x[N_x-1] & \vdots & x[2] \\ & x[N_x-1] & \vdots \\ \mathbf{0} & & x[N_x-1] \end{bmatrix} \\
&= \begin{bmatrix} r[0] & r[1] & \cdots & \cdots & r[L-1] \\ r[1] & r[0] & r[1] & & \vdots \\ \vdots & r[1] & r[0] & \ddots & \vdots \\ \vdots & & \ddots & \ddots & r[1] \\ r[L-1] & \cdots & \cdots & r[1] & r[0] \end{bmatrix} =: \mathbf{R}
\end{aligned} \tag{4.3}$$

For invertibility of $\mathbf{X}^T\mathbf{X}$, it is best if this matrix is diagonally dominant, $r[0] \gg |r[i]|$. This implies that the signal has large energy ($r[0]$ large) and low cross-correlations with shifted versions of itself.

A good training signal would be an impulse, $x[n] = \delta[n]$. In this case, $\mathbf{R} = \mathbf{I}$. However, the amount of energy that can be transmitted is limited ($r[0]$ does not grow with growing $N$).

A bad training signal is a sinusoid, $x[n] = \sin(\omega n + \phi)$. For such a signal, it can be shown that (for large $N_x$)

$$r[k] = \frac{1}{2}\cos(\omega k) = \frac{e^{j\omega k} + e^{-j\omega k}}{4}$$

Obviously, this signal correlates very well with shifts of itself. Inserting this in (4.3), it will be seen that $\mathbf{R}$ has rank 2! For $L > 2$, this matrix is never invertible.

**Tasks**

- Compute the SVD of $\mathbf{X}$ for each of the training signals $x_1[n], \cdots, x_4[n]$. Take a large $N$, and $L = 5$.

  Plot the singular values as follows: `plot(svd(X),'+')` (use different plots or different colors/markers for each signal).

- The number of 'large' singular values indicates the rank of $\mathbf{X}$. What can you say about the rank for each of the test signals? Which one is the best?

Small singular values do not carry information but will be inverted in the inversion of $\mathbf{X}$ and lead to noise enhancement. Practically, we use a threshold $\varepsilon$ on the inversion of matrices: singular values smaller than $\varepsilon$ are not inverted but replaced by zero. In Matlab: `pinv(X,eps)`.

## 4.3 CHANNEL ESTIMATION USING A MATCHED FILTER

If $L$ is very large, as we will have with audio channels, then (4.2) requires a large matrix inversion $(\mathbf{X}^T\mathbf{X})^{-1}$. Although it could be computed off-line, it might be prohibitively complex. In these cases, it is often proposed to simply omit this factor (i.e., we approximate $\mathbf{X}^T\mathbf{X} \approx \alpha\mathbf{I}$) and use

$$\hat{\mathbf{h}} = \mathbf{X}^T\mathbf{y} = \begin{bmatrix} x[0] & x[1] & \cdots & x[N_x-1] & & & \\ & x[0] & x[1] & \cdots & x[N_x-1] & & \\ & & \ddots & & & \ddots & \\ & & & x[0] & x[1] & \cdots & x[N_x-1] \end{bmatrix} \begin{bmatrix} y[0] \\ \vdots \\ y[N_y-1] \end{bmatrix} \tag{4.4}$$

This is called the "Matched Filter", and it is used very often in telecommunication applications. It will work if the columns of $\mathbf{X}$ are approximately orthogonal to each other, i.e., $N_x$ is large, and the training sequence $x[n]$ is designed such that this property more or less holds. The scale of $\hat{\mathbf{h}}$ is not correct, we can compute $\alpha$ as $\alpha \approx \|\mathbf{x}\|^2$ and scale by $1/\alpha$.

If we look at $\mathbf{X}^T$ in more detail, we see that it is a Toeplitz matrix, so that multiplication with $\mathbf{X}^T$ can be viewed as a convolution (filter operation). The columns of $\mathbf{X}^T$ are the *reverse* of the sequence $x[n]$, thus

the Matched Filter is a filter operation using the reversed sequence $x[n]$, i.e., $x[-n]$. It can be implemented efficiently in Matlab as[2]

```
Ny = length(y); Nx = length(x); L = Ny - Nx + 1;
x = x(:); y = y(:);    % ensure column vectors
xr = flipud(x);        % reverse the sequence x (assuming a col vector)
h = filter(xr,1,y);    % matched filtering
h = h(Nx+1:end);       % skip the first Nx samples, so length(h) = L
alpha = x'*x;          % estimate scale
h = h/alpha;           % scale down
```

Equation (4.4) further shows that the Matched Filter can be interpreted as *correlating* the received signal $y[n]$ with the transmitted signal $x[n]$, for several lags (represented by the various row shifts in $\mathbf{X}^T$). In terms of convolutions, we have the received signal $y[n] = h[n] * x[n]$ and can write the Matched Filter as

$$\hat{h}[n] := y[n] * x[-n] = h[n] * (x[n] * x[-n]) = h[n] * r[n]$$

Thus, the channel estimate can be written as $\hat{h}[n] = h[n] * r[n]$ with $r[n] := x[n] * x[-n]$. We don't recover the true channel, but the channel convolved with $r[n]$. This convolution will smear $h[n]$ and limit the accuracy of the estimate. The best results are obtained if $r[n]$ approximates a delta spike.

As we have seen in the previous section, the sequence $r[n]$ is the autocorrelation sequence of the input signal. It always has a peak at $r[0]$ (see section 2.4). For long sequences $x[n]$, this peak is usually quite pronounced, with relatively small sidelobes, so that the smearing of $h[n]$ will be relatively minor. Even if $\hat{\mathbf{h}}$ may not be a good approximation of the actual channel, it is expected that at least the peak of $\hat{\mathbf{h}}$ corresponds to the peak of the true channel.

The matrix inversion method for deconvolution corrects for the convolution with $r[n]$, using the inverse of the autocorrelation matrix $\mathbf{R} = \mathbf{X}^T\mathbf{X}$. Apart from the computational advantages of not using the inverse, another advantage of the Matched Filter is that there are no issues with noise enhancement if $\mathbf{X}^T\mathbf{X}$ is poorly conditioned.

The requirement that $\mathbf{X}^T\mathbf{X} \approx \alpha\mathbf{I}$ is satisfied by sequences that have low autocorrelations for all lags except for lag 0. "Gold codes" are an example of this. Also long random sequences are expected to satisfy this. Contrary, a sinusoid is an example of a signal that has strong autocorrelations for all lags.

### Tasks

- Create a Matlab function `ch2.m` which implements channel estimation using the Matched Filter. The inputs and outputs are the same as for `ch1.m`. Also create a similar test function `test2.m`.
- Debug and test the channel estimation algorithm. Use the true channel length $L$. Is the estimated channel close to the true channel? (You probably have to take a large $N$ to get reasonable results.)

[2]Here it is assumed that **x** is a column vector. If it is a row vector, reverse the sequence using fliplr. A simple way to create a column vector out of any vector in Matlab is x = x(:).

The Matched Filter should give the true channel convolved with the autocorrelation sequence of the input signal, $r[n] = x[n] * x[-n]$, in Matlab: `r = conv(x, flipud(x))` if `x` is a column vector (for row vectors, use `fliplr`).

- (report 9) Make plots of the autocorrelation sequence for each of the 4 test signals. How do they differ? Which signal is most suitable? Take a large $N$.

- In practice, we don't know $L$. Test for various "estimated" channel lengths. Is this a sensitive parameter?

- What happens if we have an interfering signal from a neighboring user? If we do a Matched Filter using a training sequence $x_1[n]$ on a recorded sequence $y[n] = h[n] * x_2[n]$, where $x_1[n] \neq x_2[n]$, then we obtain $\hat{h}[n] = y[n] * x_1[-n] = h[n] * (x_2[n] * x_1[-n])$. The cross-correlation $r_{12}[n] = x_2[n] * x_1[-n]$ should ideally be very small. E.g., for random sequences, this cross-correlation will converge to zero as you take long sequences.

  If we do matrix inversion, we can view application of $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ as a Matched Filter, followed by a correction. Thus, also here, we would prefer interfering sequences to have a low crosscorrelation with the desired sequence.

  Consider two random signals of the form $x_4[n]$. Test their cross-correlation properties. Is this family of signals suitable to serve multiple users?

- (report 10) Discuss how the training sequence and its length $N_x$ should be designed, and how $\hat{L}$ should be chosen in the estimation algorithm. You will need this in the EPO4 project.

  *Hint:* Only some general observations are expected. Training sequence design is a large research topic; search e.g., for "Gold codes".

### 4.4 DECONVOLUTION IN FREQUENCY DOMAIN

Deconvolution in time domain may become complicated if the channel length $L$ is large. In that case, $\mathbf{X}$ becomes too large to invert. An alternative to the Matched Filter is to do the deconvolution in frequency domain.

From $y[n] = h[n] * x[n]$ we can derive $Y(\omega) = H(\omega)X(\omega)$ and hence we can estimate $H(\omega) = Y(\omega)/X(\omega)$. However, this property is valid for the DTFT, whereas in practice we have to use the DFT (or in fact the FFT). Let $Y[k]$ be the DFT of $y[n]$, and similarly for $H[k]$ and $X[k]$. We would like to use the property $Y[k] = H[k]X[k]$, for $0 \leq k \leq N-1$. However, this introduces two complications. First of all, we need all sequences to be of equal length $N$. In that case, the DFT sequences will also have $N$ samples, and can be pointwise multiplied. Secondly, the equivalence of convolution to multiplication for the DFT is in fact, as we will see in the EE2S21 *Signal Processing* class,

$$Y[k] = H[k]X[k] \qquad \Leftrightarrow \qquad y[n] = h[n] \circledast x[n]$$

where $\circledast$ is a "circular convolution",

$$h[n] \circledast x[n] = \sum_{m=0}^{N-1} h[m]x[(n-m)_N], \qquad n = 0, \cdots, N-1$$

where the notation $(\cdot)_N$ denotes "modulo $N$". This relation is a bit easier to see in matrix form:

$$\mathbf{y} = \mathbf{X}_c\mathbf{h} \quad \Leftrightarrow \quad \begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ \vdots \\ y[N-1] \end{bmatrix} = \begin{bmatrix} x[0] & x[N-1] & \cdots & x[1] \\ x[1] & x[0] & \cdots & x[2] \\ x[2] & x[1] & \cdots & x[3] \\ \vdots & \vdots & \vdots & \vdots \\ x[N-1] & x[N-2] & \cdots & x[0] \end{bmatrix} \begin{bmatrix} h[0] \\ h[1] \\ h[2] \\ \vdots \\ h[N-1] \end{bmatrix} \tag{4.5}$$

Here, $\mathbf{X}_c$ is called a circulant matrix: each column is a circular shift of the previous column. It is a square $N \times N$ matrix, and replaces the Toeplitz matrix that we had earlier in (4.1). In comparison to before, the upper triangular part is filled in, and the matrix is square rather than tall.

Fortunately, we can obtain this circulant structure from (4.1) if $N_y$ is sufficiently large, particularly it should hold that $N_y \geq N_x + L - 1$. Starting from (4.1) we augment $x[n]$ with zeros for $n = N_x, \cdots, N_y - 1$, and similarly the channel $h[n]$ is padded with zeros such that it has length $N_y$ rather than $L$. Then all sequences have the same length $N_y = N$, and it is not hard to see that (4.5) is equivalent to (4.1). Alternatively, we will see in the Signal Processing course that under these conditions the usual linear convolution is equivalent to circular convolution.

Circulant matrices have an important property: it can be shown that they are diagonalized by the DFT matrix $\mathbf{F}$. The $N \times N$ matrix $\mathbf{F}$ is the operator that maps the sequence $x[n]$ $(0 \leq n \leq N-1)$ into $X[k]$, similarly it maps $y[n]$ into $Y[k]$ and $h[n]$ into $H[k]$. The diagonalizing property is

$$\mathbf{F}\mathbf{X}_c\mathbf{F}^{-1} = \mathbf{\Lambda}_x \qquad \Leftrightarrow \qquad \mathbf{X}_c = \mathbf{F}^{-1}\mathbf{\Lambda}_x\mathbf{F}$$

This is in fact an eigenvalue decomposition of $\mathbf{X}_c$, and the eigenvalues $\mathbf{\Lambda}_x$ correspond to the Fourier coefficients $X[k]$. It follows that

$$\mathbf{F}\mathbf{y} = \mathbf{F}\mathbf{X}_c\mathbf{F}^{-1}\mathbf{F}\mathbf{h} = \mathbf{\Lambda}_x\mathbf{F}\mathbf{h} \qquad \Leftrightarrow \qquad Y[k] = X[k]H[k], \qquad 0 \leq k \leq N-1$$

This shows the equivalence of circular convolution to pointwise multiplication in DFT domain. Hence, we can compute

$$H[k] = \frac{Y[k]}{X[k]}, \qquad 0 \leq k \leq N-1$$

to obtain the estimate of the channel in DFT domain. Next, the inverse Fourier transform is applied to the sequence $H[k]$ to obtain the channel impulse reponse $\mathbf{h}$. Together these steps implement a deconvolution in frequency domain. A major advantage is its computational simplicity: we need 3 FFTs (complexity order $3N\log N$) and a pointwise division (complexity order $N$), rather than a matrix inversion of $\mathbf{X}$ (complexity order $N_y L^2$). We also do not need to store large matrices.

In Matlab, this method is done as follows (assuming column vectors):

```
Ny = length(y); Nx = length(x); L = Ny - Nx + 1;
Y = fft(y);
X = fft([x; zeros(Ny - Nx + 1,1)]);  % zero padding to length Ny
H = Y ./ X;                          % frequency domain deconvolution
h = ifft(H);
-- truncate h to length L
-- make the sequence real if necessary
```

Also the Matched Filter could be computed in frequency domain. In time domain, the Matched Filter computes $\hat{h}[n] = y[n] * x[-n]$. If as before we take care using zero padding that sequences have equal length $N$ and circular convolution is equal to linear convolution, then equivalent of the Matched Filter in frequency domain is

$$\hat{H}[k] = Y[k]X^*[k]\,.$$

### Problems with frequency-domain equalization

Since a pointwise division is done in frequency domain, it is immediately clear that the performance will be low for frequencies where $X[k]$ is small. Thus, this will only work for training sequences that resemble either an impulse or "white noise" random sequences, as these have a wide frequency content, A sinusoid is the worst training sequence, since its frequency content is almost zero everywhere.

If you design a training sequence, it is a good idea to look at its frequency content, even if you will do the equalization in time domain. Invertibility of the $X[k]$ correspond exactly to the invertibility of $\mathbf{X}_c$ (since the $X[k]$ are the eigenvalues of $\mathbf{X}_c$) and this is very similar to the invertibility of $\mathbf{X}$, since (with zero padding) $\mathbf{X}_c$ and $\mathbf{X}$ are nearly the same.

In practice, we should invert $X[k]$ only for frequencies where it is sufficiently strong, and set the estimates of the remaining $H[k]$ equal to zero. (This corresponds to taking a pseudo-inverse of $\mathbf{X}$.) Suppose we use a threshold $\varepsilon$ and set $\hat{H}[k] = 0$ if $|X[k]| < \varepsilon$. Then the channel estimate $\hat{H}[k]$ which we obtain satisfies

$$\hat{H}[k] = H[k]G[k]\,, \qquad \text{where } G[k] = \begin{cases} 1, & |X[k] > \varepsilon \\ 0, & \text{elsewhere} \end{cases} \tag{4.6}$$

In time domain, the estimate $\hat{h}[n] = h[n] * g[n]$ is the convolution of the true channel with this "selector function". This will limit the resolution that can be obtained.

Finally, another problem with frequency-domain equalization is that there is no guarantee that the computed $\mathbf{h}$ corresponds to an FIR channel—in general, this will be a vector that has all $N_y$ entries nonzero. In that case also the correspondence between linear and circular convolution is lost.

### Tasks

- Implement frequency-domain equalization as a Matlab function `ch3.m`. Include a threshold on the inversion of $X[k]$ that is a fraction of the peak amplitude in frequency domain.

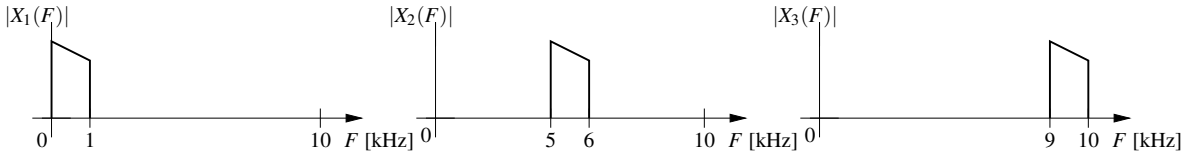
Test the function using the four test signals in `datagen.m`.

*Hint: the function* `find` *allows you to find the indices of the entries of a vector larger or smaller than a threshold. E.g.,* `ii = find(abs(X) > eps)`.

- Suppose we have a training sequence $x[n]$ whose frequency domain $X[k]$ is bandlimited, e.g., only contains frequencies smaller than $F_c = 1kHz$ whereas the sample frequency is $F_s = 20$ kHz.


The effect of the threshold technique on $\hat{h}[n]$ is given by (4.6). What is $G[k]$ in this case? And $g[n]$? Make plots using Matlab.

- (report 11) Suppose that we have a choice between 3 different training sequences, $x_1[n]$, $x_2[n]$ and $x_3[n]$, that only differ because they have a different "carrier frequency".




Compute and plot the corresponding functions $g_1[n]$, $g_2[n]$, $g_3[n]$. What is the effect of the "carrier frequencies" on the resolution of the channel estimate? Which training sequence is preferred?

The above signals had bandwidths of 1 kHz. What changes if these become, e.g., 2 kHz? Can you say something on the desired bandwidth of the sequences?

*Hint:* when defining the spectra, make sure you define "two-sided" spectra (i.e., including the negative frequencies, these end up at the high frequencies close to $F_s$). That way, the IFFT generates real-valued signals.

- (report 12) Let's try to verify the above results. Generate a "baseband" test signal $x_1[n]$:

```
Fs = 20e3;                % sample frequency
N = 50;                   % number of bits
p = ones(10,1);           % pulse shape (square block)
% p = bartlett(21);       % alternative pulse (triangle)
s = sign(randn(N,1));     % random sequence +- 1
ss = kron(s,[1; zeros(9,1)]);   % interpolate s with zeros
x1 = conv(ss,p);          % convolution of p with s
```

Here, $s[n]$ is a random sequence of $N$ bits, $+1$ or $-1$ (a "BPSK signal"), and $p[n]$ is a pulse shape, in this case a square block. The Kronecker product `kron` makes a sequence where each $s[n]$ is replaced by a vector $[s[n]\,0\cdots0]^T$ which creates some spacing between the bits. Convolution with $p[n]$ modulates the BPSK sequence with a square pulse.[3]

Next, define the sequence $x_2[n]$ obtained by modulating $x_1[n]$ with a cosine, $x_2[n] = x_1[n]\cos(\omega_2 n)$. Take $\omega_2 = \pi/2$.

Plot the sequences $x_1[n]$, $x_2[n]$ in time domain and frequency domain. The latter should look like the plots above.

Now take a simple channel $h[n]$, e.g., `h = [1, 2, 3, 2, 1]`, and compute the measured signals $y_1[n], y_2[n]$. Apply your function `ch3.m` to recover $h[n]$. Take a threshold such that small values of $X[k]$ are not inverted. Give plots of the two recovered channels, in time domain and frequency domain. How do these correspond to the original channel?

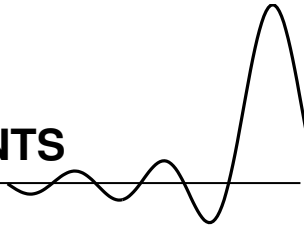
What you will observe is that if the training sequence is bandlimited, the channel is not very accurately recovered. If the selector function $G[k]$ is a bandpass, then $g[n]$ is a modulated sinc pulse and this becomes visible in the channel estimate. There is a loss of resolution.

In the next labday, you will be asked to design an audio beacon signal and a channel estimator, which will be used in the EPO4 project. The beacon signals look like modulated BPSK sequences. Use today's results to guide you.

[3] Square pulses are not often used in telecom because they are not very compact in frequency domain. It is possible to use other pulse shapes, e.g. a triangle pulse is already more concentrated in frequency. In telecom practice, "raised cosine" pulses are often used.

*Chapter* **5**

# LABDAY 4: AUDIO CHANNEL MEASUREMENTS



## Contents



In the EPO4 project, toy cars will transmit beacon signals, which are coded audio pulse trains. The signals are recorded by 4 microphones. Since we know the transmitted signal, we can estimate the 4 channels by deconvolution. By comparing the channel estimates, we can estimate differences in propagation time, which will be used to locate the cars.

For today's lab day, we will look in more detail into audio channel estimation. How does a typical channel look like? Is it estimated well using the deconvolution algorithm? Which transmit sequence should be used? If we compare two channel estimates, can we estimate a difference in propagation time, and how does that translate into physical distance? The results will be needed in the EPO4 project.

**Learning objectives**

- Familiarizing yourself to use Matlab to transmit and record audio data.
- Testing your channel estimation algorithms on actual data. The transmitted signal is a Matlab model of the programmable audio beacon.

**Preparation**

**What is needed**

- PC with 2 microphones and an external loudspeaker, cables and a special plug to insert the microphones into the stereo input of the PC;
- From Brightspace: Matlab scripts '`findInDevID.m`', '`refsignal.m`', and '`send_refsignal.m`'.
- Ruler or measurement tape (1 to 2 meter).

### 5.1 PLAYING WITH THE LOUDSPEAKER AND THE MICROPHONE

Let's see how in Matlab you can record a signal on the microphone. The functions needed for this are

```
Fs = 8000;                          % sample rate of the microphone
recObj = audiorecorder(Fs,16,1);  % create audio object, 16 bits resolution
disp('Start speaking.')
recordblocking(recObj, 2);          % do a 2 second recording (blocking)
disp('End of Recording.');

% Play back the recording.
play(recObj);

% Store data in double-precision vector
y = getaudiodata(recObj);

% Plot the samples.
plot(y);
```

The function `recordblocking` does not return control to Matlab until it finishes its (2-second) recording. The function `getaudiodata(recObj)` converts the audio object `recObj` into the signal $y[n]$ (or vector **y**) that you would use for our processing, filtering, etc.

Alternatively, you can use `record(recObj)` which returns control to Matlab immediately, and keeps recording until you say `stop(recObj)`:

```
recObj = audiorecorder(22050, 16, 1);
disp('Start speaking.')
record(recObj);     % speak into microphone...
pause(2);           % pause for 2 seconds (or do other things)
disp('End of Recording.');
stop(recObj);       % stop recording
play(recObj);       % listen to complete recording
```

During recording, you could do other things (e.g., computations).

What we would like to do now is to play a signal over the loudspeaker, and record it using the microphone. You can choose a sample rate for the transmission, and the same or different sample rate for the recording. That will lead to some interesting effects. The general set-up is as follows:

```
%------------------------ create sound -------------
Fs_TX = 22050;                  % transmitter sample rate
```

```
x = zeros(1,Fs_TX);             % 1 second of silence
for ii = 1:Fs_TX/10:Fs_TX,
   x(ii) = 1;                   % insert some impulses
end
xObj = audioplayer(x,Fs_TX);    % convert one second of sound to audio format

%------------------------- play sound and record the response
Fs_RX = 22050;                      % microphone sample rate
recObj = audiorecorder(Fs_RX,16,1); % 16 bits, 1 channel
play(xObj)                          % play and continue directly
recordblocking(recObj,1);           % start recording 1 second
y = getaudiodata(recObj);           % float representation of recording
```

The PCs in the lab setup have a front audio jack for the microphone; there is also one in the back. Use the Windows control panel (sound panel) to select the right interface: "Soundmax Integrated HD Audio". Then check the level settings of the microphone such that it does not saturate and still measures something. The "listen" option should be switched off to avoid unwanted feedback. Also switch off any additional filtering.

On Brightspace, find the script `findInDevID.m`; with this Matlab function you can search for the ID of the microphone input channel. You need this to make Matlab listen to the right input. Usage:

```
audiodevinfo  % list available audio devices
ID = findInDevID('Ingang achter (SoundMAX Integra'); % find specific ID
r = audiorecorder(Fs,nbits,nchans,ID); % open audio device "ID"
```

### Tasks

- Test to see if you can transmit a signal over the audio channel and record the result. E.g., use impulses and/or the train signal. Check the volume settings of your microphone and loudspeaker until things work well.

  Try various transmit and receive sampling rates (not all recording sampling rates are supported by Matlab or the audio interface).

- Choose `Fs_TX = 22050; Fs_RX = 22050;`

  Transmit a single impulse over the audio channel: this directly gives the impulse response of the audio channel. Plot the time-domain impulse response, and the frequency domain amplitude spectrum. Take care that your x-axis legends are correct; for the frequency plot, use kHz.

  Do the same for `Fs_TX = 22050; Fs_RX = 8000;` Do you expect any aliasing?

  Do the same for `Fs_TX = 4000; Fs_RX = 22050;` Do you expect any aliasing? What is the highest frequency which you see in the spectrum (why?).

With `Fs_RX = 22050;`, what is the highest transmit sample rate that is meaningful?

Note that aliasing is not expected as the sample card uses an anti-aliasing filter before sampling. This is also the reason that not all sampling frequencies are permitted.

- (report 13) At the microphone, what sample rate is at least needed to have a resolution of 1 cm?

  Test various distances between the microphone and the loudspeaker. Do you see (systematic) changes? For a distance of 50 cm and for 1 meter, what propagation delays do you expect? Can you see this back in your time-domain data? (Note that there is a processing delay before the sample card responds and starts to sample, and probably this delay is not entirely constant.)

  For a distance of 1 cm, there is hardly any propagation channel; do you get an 'ideal' impulse or is there some distortion? These would be the loudspeaker and microphone responses. (At this short distance, make sure there is no clipping.)

  Save and store some good examples of the channel impulse responses that you measured (at 1 cm, 50 cm, 1 meter). Give plots of some typical channel impulse responses in your report; also write down the propagation distances. The time-domain axis in these recordings could be very large; make sure you properly zoom in on the "interesting" part of the plot.

- In the previous cases, you had a direct line of sight (LOS). But sometimes, there is no direct line of sight (NLOS), due to some object blocking the direct path. Make a typical recording of a NLOS channel impulse response—how does it compare to the previous LOS cases? Can you still determine the propagation distance?

## 5.2 MANUAL: MODEL FOR THE AUDIO BEACON SIGNAL

In the EPO4 project in Q4, we will use an audio beacon to locate the car. The audio beacon, once switched on, continuously transmits a sequence of pulses. It is controlled by a programmable microcontroller. It is possible to change the number of pulses, duration of each pulse, the sequence itself, and the period after which the sequence is repeated.

The pulse sequence is a modulated binary code sequence with "on-off keying" (OOK). If a bit in the sequence is 0, nothing is transmitted; if the bit is 1, a modulation carrier frequency is transmitted during a certain period.

Besides the actual bit sequence (code word), the parameters that determine the signal are

– Length of the code sequence (number of bits; parameter `Ncodebits`), at most 64 bits;

– Modulation carrier frequency (parameter `Timer0`), at most 30 kHz, although this is probably beyond the specs of the loudspeaker and microphones;

– Duration of a single bit (parameter `Timer1`), this defines the rate at which the modulation carrier signal is switched on or off by the bits in the code word;

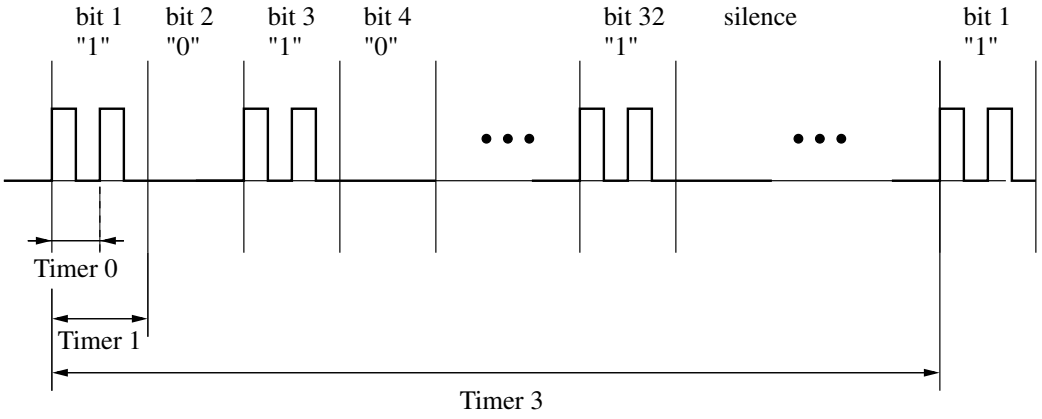



**Figure 5.1**. An example of the pulses generated by the audiobeacon.

**Table 5.1**. Timers frequency configuration table

| Timer index | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Carrier Freq | (Timer0) | [kHz] | 5 | 10 | 15 | 20 | 25 | 30 | | | | |
| Code Freq | (Timer1) | [kHz] | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 | 3.5 | 4.0 | 4.5 | 5.0 | |
| Repeat Freq | (Timer3) | [Hz] | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

– Repetition rate of the bit sequence (parameter `Timer3`). After the code sequence has been played over the loudspeaker, it will be silent for a certain period, and then the sequence is transmitted again, at a rate determined by Timer 3.

In the audio beacon, these parameters can be modified in the program code of the program that runs on a microcontroller. In the Matlab model `refsignal`, the same parameters are used.

The maximal repetition rate is 10 Hz (corresponding to a period of 100 ms). If 64 bits are used at the lowest rate of Timer 1 (1 kHz), then the duration of the sequence will be 64 ms. You will have to choose settings such that the channel impulse response dies out during the remaining period of silence, before the next pulse sequence starts. Possible values for the Timer parameters are listed in Table 5.1; instead of the actual values, the Timer Index values are used.

The default setting of the audio beacon is a code sequence bit-stream of 32 bits with:

| | | | |
|---|---|---|---|
| Code length | = | 32 bits | NCODEBITS = 32 |
| Carrier frequency | = | 20 kHz | TIMER0_INDEX = 3 |
| Code frequency | = | 5 kHz | TIMER1_INDEX = 8 |
| Repeat frequency | = | 2 Hz | TIMER3_INDEX = 2 |
| Code word | = | 92340f0f | (hex) |

## 5.3 TEST USING THE MATLAB AUDIO BEACON

We will now use the audio beacon signal as the transmit signal and make a channel estimation function for it. This function will be needed in the EPO4 project.

For convenience, we will use in this practicum a Matlab function that generates the same sequence as the audio beacon. The advantage of using a Matlab function is that we can quickly change the settings, and the receiver is more or less synchronized with the transmitter. The Matlab function is called `refsignal.m` (provided on Brightspace), and it accepts the same parameters as the actual beacon.

### Design considerations

It is important to choose settings that will will give an optimal channel estimate in the presence of noise or interference. This will generally require long sequences. However, we have to wait until one or two pulse sequences have been received before we can do a channel estimation, and this will form the basis for the location estimate. Faster updates will result in better tracking. For this, it is important to have short sequences!

Furthermore, we need to plan for a "guard interval" of silence between two sequences, long enough for the channel response to return to zero. For a large room and a maximal distance of 5 to 6 meters between the beacon and the microphone, what is that duration (in ms)? This determines the maximal repetition rate that you can hope to achieve.

Another aspect to consider is the dynamic range. The microphone gain will have to be set such that it will not clip even if the transmitter is very close to it, because we want to avoid nonlinear effects. But, in another extreme, over a distance of 5 to 6 meters, the audio signal is already significantly attenuated and may drown in the noise. However, we will have to be able to estimate the channel even over such distances, also in the presence of a nasty interferer close to the microphone. For this, we need long sequences, or to average over several repetitions of the sequence, so that the noise is averaged out and you remain with the channel impulse response.

The highest "data rate" is obtained using the shortest pulses. In practice, this is limited by the capabilities of the transmitter and the microphone. Use the data sheets of the audio beacon (section 5.2 to determine what is the highest frequency that you could use. Consider that you want to sample the microphone at Nyquist rate, i.e., at least twice the highest frequency of the transmitter.

Is there a reason to use very high frequencies? Probably this determines the resolution of your channel estimate. What data rates do you need to achieve a resolution of 1 cm? Can the system achieve this rate? Is there a trade-off here, i.e., a reason not to use higher frequencies than this?

Also consider the computational complexity of your algorithms. For a channel length of $L$ samples and a sequence length of $N_x$ samples, approximately how many operations are needed to estimate the channel? Is that reasonable (or how long will it take Matlab to do the computation)? You can use Matlab `tic, toc` timers to do a benchmark. Choose which of the three channel estimation algorithms you plan to use.

Finally, we have to think of the practical situation where the microphone signal contains beacon signals

of more than one user. Consider what happens if you do the deconvolution using a reference signal that does not match the transmitted signal. E.g., in the Matched Filter, we correlate the received signal with our own code sequence, and hopefully the correlation of someone else's code with our own code is small. Thus, the filter will filter out the other signals.

### Tasks

- Study the signal returned by `refsignal.m` (provided on Brightspace), in time domain and in frequency domain. What is the effect of changing the various parameters? What is the maximal "carrier frequency" (determined by Timer0) that can be used?

  In the time-domain deconvolution algorithms, the Toeplitz matrix $\mathbf{X}$ plays an important role. Study the singular values of this matrix using `plot(svd(X),'+')`. What do you deduce on the invertibility of $\mathbf{X}$?

  Also look at the frequency domain (spectrum) of the signal. We would prefer a wide frequency content, so that we probe the channel at many frequencies.

- Use the function `send_refsignal.m` (provided on Brightspace) to transmit and record a simulated "audio beacon" signal, which is generated using `refsignal.m`.

  Make and store recordings at various distances, e.g., 1 cm, 50 cm, 1 meter. Take care that at 1 cm, the microphone does not clip.

- (report 14) Apply your channel estimation algorithms of Labday 3 to recover the audio channel impulse response.

  Illustrate your report with plots of the transmit sequence and its spectrum, the receive sequence, the recovered impulse response, and compare to the impulse response as obtained by directly transmitting an impulse.

*Hint 1:* Your receive vector is probably very long, and it is not quite possible to construct and invert the corresponding $\mathbf{X}$-matrix. In that case, try some work-arounds: (1) truncate sequences to the interval where they are nonzero, (2) resort to the Matched Filter approximation, implemented as a filter, and/or (3) try frequency-domain channel estimation. You can also try to transmit at a low carrier frequency (Timer0), and use a relatively low recorder sample rate $F_s$.

*Hint 2:* For the deconvolution, you need to know the transmitted signal, and for this you can use two approaches: (1) use the modeled `refsignal.m`, or (2) use the recording at 1 cm as a clean copy of the transmitted signal. Theoretically, the modeled signal should be fine, but in practice, the recorded signal is more reliable: the model may be imprecise, and it does not contain the filter effects of the loudspeaker and microphone. The problem with the modeled signal becomes more important once you use the real audio beacon in the EPO4 project.

- (report 15) Design the 'optimal' parameters of the audio beacon and corresponding microphone settings, for a maximal distance of 5 to 6 meters and a resolution of 1 cm. Document your choices.

As listed before, there are several aspects to the design: sufficient resolution (sample rates), good deconvolution properties, and also good discrimination of your own signal to that of another audio beacon that transmits at the same time.

## 5.4 TDOA ESTIMATION

In the EPO4 project, we will try to locate a car using an audio beacon. We will use time-difference of arrival (TDOA) measurements made at made at microphones positioned at known locations. The audio beacon transmits signals which are received by up to 5 microphones. Depending on the distance to each microphone, the signal arrives a little bit earlier or later, and we can convert that into physical distances. For each pair of microphones, we will compute this TDOA, or the physical difference in propagation distance. If we have a large enough number of microphones (4 should work...), then we can calculate the $(x, y)$ location of the transmitter using a Least Squares algorithm. We will do this in the EPO4 project.

Before we can do localization, we have to work on this question: Given the impulse responses measured by two microphones, how is the Time Difference of Arrival (TDOA) estimated? That is the topic of today's assignment.

The audio beacon transmits a continous stream of pulses, using a certain repetition period $T_R$ (specified via the Timer3 parameter, e.g., 100 ms). If we are not synchronized to the beacon, there is no guarantee that an entire pulse sequence is captured in a period $T_R$: you might have the tail of one sequence, and the head of the next. It is probably easier to capture samples for at least $2T_R$ seconds, i.e., 2 intervals.

The received signal $y_i[n]$ is the convolution of the data sequence, the audio channel for the $i$th microphone, and the filtering effects of the transducers. In the previous section, you have applied knowledge of the data sequence to estimate the audio channel impulse response using deconvolution. Given at least $2T_R$ seconds of data, we first apply the deconvolution filter to the entire available data, for each signal $y_i[n]$ in the same way. That will give channel estimates $h_i[n]$.

The next step is to synchronize to the start of the first impulse in the first channel. The assumption is that there is line of sight, so that the first strong pulse is the line-of-sight pulse. Before that pulse, the response is mostly zero over a long interval. We can make a guess on how long that interval will usually be (it also depends on the repetition period and the typical duration of the impulse response); let's say this interval is known to be larger than $T_I$. We can search for an interval of duration $T_I$ with energy below a certain threshold, and then continue to search for the highest peak that comes immediately after that. Let's say this happens at time index $n_{max}$.

Next, we go to the second channel and search for the highest peak in a window before/after $n_{max}$. We can determine a suitable length $T_S$ of this window by computing the maximal delay that you expect; e.g., for a maximal distance of 5 m, that delay is about 15 ms.

After we have found the matching peak in the second trace, we can compute the difference between both peaks (in samples), and convert this into a time difference and then a physical distance (knowing the speed of sound).

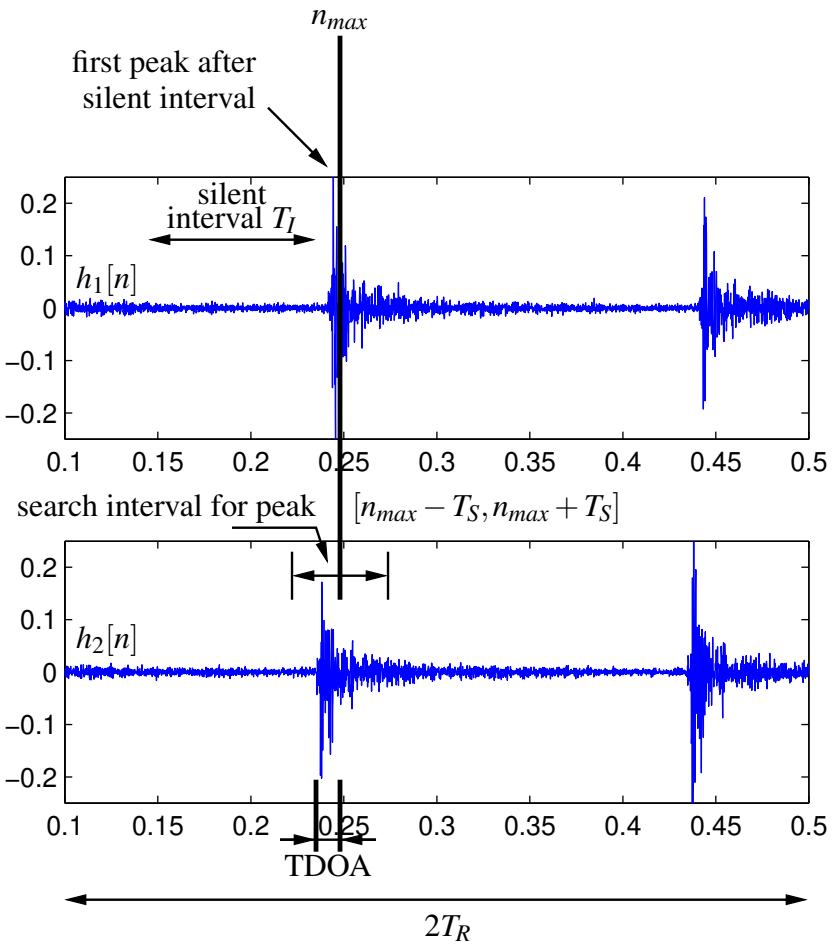



**Figure 5.2**. TDOA estimation: for the impulse response $h_1[n]$ of the first microphone, find the first peak after the silent interval, then go to the second microphone $h_2[n]$ and look for a matching peak in the search window.

Robust TDOA estimation is an active research topic, so there are many possible alternatives for the above algorithm.

### Tasks

- (report 16) At the end of section 5.3, you determined suitable values for the beacon parameters and the microphone settings (such as sample rate).

  Based on this, determine to how many samples a maximal propagation delay corresponding to 5 m will correspond. That will set your search interval.

  What is a typical channel impulse response length (in milliseconds, and samples)? The duration of a code sequence is determined by the number of symbols and Timer1. What is the total duration of the received signal? Choose the sequence repetition period (Timer3) to be larger than that.

- Implement a TDOA estimation algorithm along the lines discussed above, or invent your own algorithm. Describe the algorithm in the report.

  Test the algorithm using synthetic measurement data where you know the true time offsets, e.g., a measured impulse response and a delayed copy of the same response. Try various offsets to see that your peak detection is robust for all possible offsets (positive and negative), for up to 5 m.

  *Hint:* Matlab commands that may be helpful: `find, max`. A more advanced command is `findpeaks`.

- Make an experimental set-up with two microphones, placed in a line with the loudspeaker of the PC so that you can easily figure out the true propagation delay. For this, you will need a specially prepared cable that allows to plug in two microphones into the stereo input of the PC.

  Make 2-channel recordings of various seconds using the Matlab audio beacon reference model as a transmitter. Annotate the data (sampling rate, distance between microphones, setting of the audio beacon, etc.). Store the data for later use.

  *Hint:* Stereo recordings can be made in Matlab using `audiorecorder(Fs,Nbits,Nchan,ID)`, where you use `Nchan=2`.

  (report 17) Apply your channel estimation and peak detection algorithms and see if you can estimate the distance between the microphones correctly.

  Make graphs to show a typical data segment, and the locations of the estimated peaks.

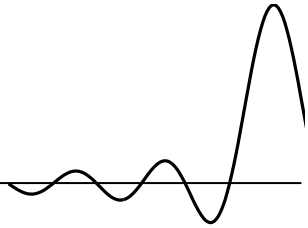
  How accurate can the TDOA be estimated? Is the estimate robust or are there outliers?

  What is the impact of "non-line-of-sight" (NLOS), where the direct path of the loudspeaker to one of the microphones is blocked?

So far, you probably segmented the data "by hand". For actual use in a tracking loop (as in EPO4), this needs to be automated; also note that the recording will not be synchronized to the transmitter. There is no best way to do this. You'll have to invent some technique for the EPO4 project.

*Chapter* **6**

# LABDAY 5: FILTER DESIGN



## Contents



We will design basic lowpass filters using (1) the inverse DFT technique which leads to FIR filters, and (2) the analog design technique followed by a transformation to the digital domain. We will use that to make a detector for the telephone touchtone signal.

**Learning objectives** Basic filter design using Matlab functions.

**Preparation** Read the chapter so you know what to expect. Refresh your knowledge on filter design from the *Signals and Systems* course.

**What is needed**

– PC with a built-in loudspeaker;

– Matlab audio test signals, typically obtained using `load gong`, `load handel`, `load train`.

– Matlab audio test signal from Brightspace (Labday 2): `touch.mat`.

Our objective in this chapter is to design a digital lowpass filter specified as follows:

| | |
|---|---|
| Sample frequency | $F_s = 8192$ Hz |
| Cut-off frequency (-3dB) | $F_c = 1$ kHz |
| Stopband frequency | $F_1 = 1.5$ kHz |
| Stopband damping | 40 dB |

**Table 6.1**. Characteristics of windows

| Type | Width of main lobe | Peak sidelobe (dB) |
|---|---|---|
| Rectangular | $4\pi/M$ | -13 |
| Bartlett | $8\pi/M$ | -25 |
| Hamming | $8\pi/M$ | -41 |
| Blackman | $12\pi/M$ | -57 |

## 6.1 DIGITAL FIR FILTER DESIGN USING THE WINDOW METHOD

In this section we design a digital linear-phase FIR filter. A linear phase filter has the form

$$H(\omega) = A(\omega)e^{-j(\omega\alpha-\beta)}, \qquad -\pi \leq \omega \leq \pi$$

where $A(\omega)$ is real. This design is typically used for ideal lowpass or bandpass filters, where the filter appears like a delay of $\alpha$ samples in the passband. Hence, signals in the passband signals are not deformed, and this is important e.g., in the case of pulse shapes of communication signals. The linear-phase condition in frequency domain translates to a symmetry condition ($h[N-n] = h[n]$ or $h[N-n] = -h[n]$) on the impulse response, and if we also require causality it follows that only FIR filters can have linear phase.

Recall from the *Signals and Systems* class the design procedure: (1) specify the design in the frequency domain, in the form of a desired transfer function $H_d(\omega)$, (2) do an inverse DFT to obtain the impulse response, (3) truncate the impulse response to a finite length $N$. The last step results in an approximation and usually the design process is repeated a few times before a filter is obtained that meets the specifications.

### Window design

The truncation modifies the ideal design and typically leads to "ringing", i.e., the Gibb's phenomenon, which gives a ripple in the passband and limits the damping in the stopband. A time-domain window $w[n]$ is applied to reduce this effect; this converts into a convolution $W(\omega)$ in the frequency domain.

$W(\omega)$ should be like an impulse: a narrow peak (called the main lobe) and low sidelobes. The rectangular window is a sinc-function in frequency domain, it has a relatively narrow peak but its side lobes are high and go only very slowly to zero (as $1/\omega$). Better windows are e.g. Bartlett and Hamming. They have a main lobe that is twice as wide, but sidelobes that are much smaller. A Kaiser window has even better performance as it is designed using optimization techniques. (It also has an additional parameter $\beta$.) Windows are characterized by the width of their main lobe, and the maximal sidelobe level, see table 6.1, where $M = N+1$ is the number of coefficients. While this gives a very good indication, unfortunately, it does not exactly translate to what you obtain if you convolve the window with an ideal freqency response.

In Matlab, windows are obtained using

```
w = bartlett(N);
w = hamming(N);
w = kaiser(N,beta);
```

The function `window` gives a more complete collection. Matlab also has a "window design & analysis tool", `wintool`, that allows you to see the time domain and frequency domain response of a window, and which also reports the width of the main lobe and the sidelobe attenutation.

The first step in filter design is the selection of a suitable window, as it determines the filter order.

- Which window do you apply to reach 40 dB damping on the sidelobes? What is the width of this window (in terms of the filter order $N$)?
- Determine the required filter order $N$ to reach the specifications.
- Determine and plot the time domain and frequency domain response of this window. Does it satisfy the requirements?

  *Hint:* If $N$ is not very large, applying the DFT directly will result is a rough-looking spectrum. Use zero padding on $w[n]$ to obtain a smooth spectrum (e.g. extend to 500 samples).

### Ideal transfer function design

The next step is to define the desired (ideal) filter in frequency domain and to translate that into the time domain. A complication is that the design in frequency domain should use a continuous $\omega$, and thus we need to use the inverse Fourier transform. However, a continuous function cannot be implemented in Matlab, and the IFT does not exist in Matlab. We could either do this part of the design "by hand", or define a sufficiently fine sampled version in frequency domain and use the IFFT.

For the requested lowpass filter, the desired transfer function is

$$H_d(\omega) = \begin{cases} e^{-j\omega N/2}, & |\omega| < \omega_c \\ 0, & \text{elsewhere} \end{cases} \tag{6.1}$$

where the cut-off frequency $\omega_c$ follows from the design specifications. The filter amplitude is nonzero only for $|\omega| < \omega_c$. For the filter phase we took a delay of half the filter length. That will ensure later on that we can truncate the time-domain filter in the interval $[0, N]$. Without the phase, we would need to truncate between $[-N/2, N/2]$.

- Translate the design specifications from frequencies in hertz into angular frequencies $\omega$.

Doing the inverse Fourier transformation by hand, the impulse response of the rectangular function $H_d(\omega)$ is obtained as

$$h_d[n] = \frac{1}{2\pi}\int_{-\omega_c}^{\omega_c} e^{-j\omega N/2} d\omega = \frac{\sin(\omega_c(n-N/2))}{\pi(n-N/2)} \tag{6.2}$$

which is recognized as a sinc-function. Note how the phase shift by $e^{-j\omega N/2}$ in the frequency domain translated into a delay of $N/2$ samples.

- Make plots of $h_d[n]$ and $H_d(\omega)$. Then truncate $h_d[n]$ to the interval $n \in [0,N]$ and make plots of the resulting $h[n]$ and the corresponding $H(\omega)$.
- Generate a suitable window, e.g.

  ```
  w = hamming(N);
  ```

  and apply apply the window to compute the filter coefficients

  ```
  h = h_d .* w;
  ```

  Plot $h[n]$ and $H(\omega)$. Does the filter meet the specifications?
- Load a test signal, e.g. `load gong`, `load handel`, `load train` (recall that these functions define a vector $y$, not $x$, so you will need to do a reassignment `x=y`).
- Apply the filter and listen to the result:

  ```
  y = filter(h,1,x);
  soundsc(x);
  soundsc(y);
  ```

  Also plot $Y(\omega)$ to see what happened in the frequency domain.

### Automated functions

The preceding filter design steps are rather tedious, and the calculation "by hand" is often not feasible. To start with, we should replace this by the IFFT of a sufficiently fine sampled frequency response. This is a critical step, because there are regions where we don't need a fine resolution, whereas around the transition band we want to be more accurate. Since filter design is such a common event, several Matlab functions are available to make the process easier and more suitable for non-specialists.

The function `fir1(N,F)` designs an $N$th order FIR filter and returns the impulse response ($N+1$ filter coefficients). The frequency specifications are in `F`. If `F` is a scalar, it represents the cut-off frequency $f_c$ for a lowpass filter (where the damping at $f_c$ is designed at -6 dB). A high-pass filter is obtained using `fir1(N,F,'high')`. For a bandpass filter, use `F=[F1,F2]` and `fir1(N,F,'bandpass')`, where the passband will be $f_1 < f < f_2$.

Take note that in this function, frequencies specified in `F` are normalized frequencies such that $f = 1$ corresponds to $\omega = \pi$ or half the sample frequency $F_s/2$; unfortunately this is not the convention we

usually use (the difference is a factor 2), but appears everywhere like this in the Matlab filter design toolbox.

Without further specification, `fir1` uses Hamming windows. Other windows can be used with `fir1(N,Wn,win)`. E.g., a Kaiser window can be specified using `fir1(N,Wn,kaiser(N+1,beta))`. The function `kaiserord` allows you to find the parameters $N, \beta$ of the Kaiser window, see the Matlab help for this function.

The entire filter design can now be summarized into these commands:

```
Fs = 8192; % sample frequency
F0 = 1000; % passband frequency
F1 = 1500; % stopband frequency
alpha0 = 10^(-3/20); % max passband ripple (linear units)
alpha1 = 10^(-40/20); % max stopband ripple (linear units)

[N,Wn,beta,filtype] = kaiserord( [F0 F1], [1 0], [alpha0 alpha1], Fs );
h = fir1(N, Wn, filtype, kaiser(N+1,beta), 'noscale');
```

In the `kaiserord` function, the vector $[1,0]$ specifies that at frequency $F_0$, the filter should be 'high' and at $F_1$, the filter should be 'low', i.e., a lowpass response.

- (report 18) Carry out the above steps, plot the filter response (amplitude in dB), and verify whether the resulting design indeed meets the specifications. What is the filter order?

  *Hints:* Also plot horizontal/vertical reference lines at the specified dampings/frequencies).

  In Matlab, use `log10` for 10-base logarithm, not `log` which gives ln.

  Make sure the vertical axis is limited between (say) -60 dB and +5 dB; if there is a zero on the unit circle, Matlab by itself plots very negative numbers which are meaningless.

  If the impulse response is short, use zero padding to increase the resolution in frequency domain.

The above design process leads to a filter that approximately meets the specifications. If necessary, the filter order $N$ needs to be incremented. A problem is that in this design process, the passband ripple and stopband ripple cannot be designed independently (the strongest requirement is satisfied). This may lead to filters that are longer than necessary. In addition, there is no exact control on the passband and stopband frequencies that result in the end.

## 6.2 OPTIMAL EQUIRIPPLE FIR FILTER DESIGN USING PARKS-MCCLELLAN

A more general approach to FIR filter design is known as Parks-McClellan, which uses optimization techniques to find the best approximation. The desired frequency response is specified by a vector of frequencies (band edges) $\mathbf{f} = [f_0, f_1, \cdots]$ and their corresponding amplitudes in a vector $\mathbf{a} = [a_0, a_1, \cdots]$.

The ideal response is obtained by the line connecting the points $(f_k, a_k)$ to $(f_{k+1}, a_{k+1})$ for all $k$; this is a piecewise linear function.

The corresponding Matlab function is `firpm(N,F,A)`, where $N$ is the filter order, `F` is the vector **f**, and `A` is the vector **a**. Also in this function, frequencies specified in `F` are normalized frequencies such that $f = 1$ corresponds to $\omega = \pi$ or half the sample frequency $F_s/2$. For example:

```
h = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
```

is a design for a lowpass filter of order $N = 30$, where the passband runs from $f = 0$ to $f = 0.1$ and the stopband runs from $f = 0.2$ to $f = 0.5$.

Note that in this design, the filter order $N$ is specified, which limits the control we have over the ripples. The function does allow to specify the relative errors for each interval in a vector **w**, which helps a bit, but this is not the same as directly specifying the ripples. To this end, function `firpmord` allows to design $N$ and **w**, which is then input for `firpm`.

Our example lowpass filter design can now be summarized into these commands:

```
Fs = 8192; % sample frequency
F0 = 1000; % passband frequency
F1 = 1500; % stopband frequency
alpha0 = 10^(-3/20); % max passband ripple (linear units)
alpha1 = 10^(-40/20); % max stopband ripple (linear units)

[N,F,A,W] = firpmord( [F0 F1], [1 0], [alpha0 alpha1], Fs );
h = firpm(N,F,A,W);
```

- (report 19) Carry out the above steps, plot the filter response (in dB), and verify whether the resulting design indeed meets the specifications. What is the filter order?

  Compare to the previous design (using `fir1`). Which one is better?

Also here, the value of $N$ is often underestimated, and you will need to increment $N$ until the specifications are met.

### 6.3 DIGITAL IIR FILTER DESIGN VIA ANALOG FILTER DESIGN

The FIR design process often leads to filters that have a high order. Lower orders may be obtained if we also allow for poles. Thus, we need to design IIR filters.

IIR filter design is often done in the 'analog time domain', because that topic existed long before digital filter design. Simple filters are Butterworth filters, these are maximally flat around $\omega = 0$ and $\omega =$

$\pi$. This is generalized by Chebyshev filters, which allow for ripples either in the passband or (after transformation) in the stop band. More general filters are elliptic (or Cauer) filters, which have ripples in both the passband and the stopband.

To transform filters designs from the analog domain to the digital domain, we nearly always use the bilinear transform.

### Butterworth filter

We will first design an IIR Butterworth filter in the analog time domain, and then use the bilinear transformation to transform it into a digital filter.

The prototype Butterworth filter has the form

$$|H_a(\Omega)|^2 = \frac{1}{1+(\Omega/\Omega_c)^{2N}}$$

where $\Omega_c$ is the 3dB cut-off frequency in the analog domain. The only other design parameter is the filter order $N$. At the stopband frequency $\Omega_1$ we have

$$|H_a(\Omega_1)|^2 = \frac{1}{1+(\Omega_1/\Omega_c)^{2N}}$$

and $N$ is determined by setting this equal to the desired damping in the stopband.

To start the design, we need to know the "analog" frequency $\Omega_c$. Since we will apply a bilinear transformation in the end that will transform $\Omega_c$ into $\omega_c = 2\tan^{-1}(\Omega_c)$, we will first have to "prewarp" the "digital" frequency $\omega_c$ into $\Omega_c$, using

$$\Omega_c = \tan(\omega_c/2)$$

Be aware of not confusing this transformation with the transformation of the design requirements given in the beginning of the chapter, which are also specified as "analog" frequencies $F_c$ and $F_s$, and are transformed into "digital" frequencies as $\omega_c = 2\pi \cdot F_c/F_s$. **You cannot directly set** $\Omega_c = 2\pi F_c$ because the bilinear transform will not map that to the right $\omega_c$ in the end.

- (report 20) Our design requirements given at the beginning of the chapter specify $F_s = 8192$ Hz and $F_c = 1$ kHz. First compute $\omega_c = 2\pi \cdot F_c/F_s$ and then compute $\Omega_c = \tan(\omega_c/2)$.

  Determine $N$ for the design requirements given at the beginning of the chapter.

The computation of $N$ can also be done in Matlab using a function `buttord`, which we call as follows:

```
Fs = 8192;          % sample frequency
F0 = 1000;          % passband frequency
F1 = 1500;          % stopband frequency
R0 = 3;             % max passband damping (dB)
```

```
R1 = 40;            % min stopband damping (dB)

[N, omega_c] = buttord( F0/Fs*2, F1/Fs*2, R0, R1);
```

The input to `buttord` is a passband frequency $f_0 = 2F_0/F_s$ (in normalized units, where $f = 1$ corresponds to $\omega = \pi$, hence not our usual normalization), $f_1 = 2F_1/F_s$ is the stopband frequency, and $R_0, R_1$ are the corresponding dampings at these frequencies, this time specified in dB. The prewarping is done within this function.

- Also compute $N$ using `buttord`. Is it the same as you obtained by hand?

Knowing $N$, the the filter is completely specified. Next, we have to factorize $|H_a(\Omega)|^2$ to obtain the poles of the analog filter $H_a(s)$ (there are no zeros). In theory,

$$|H_a(\Omega)|^2 = H_a(s)H_a(-s)\Big|_{s=j\Omega} \quad \Rightarrow \quad H_a(s)H_a(-s) = \frac{1}{1+(-s/\Omega_c)^{2N}}$$

and we see that to obtain the poles we need to determine the roots of

$$1+(-s/\Omega_c)^{2N} = 0$$

The poles of $H_a(s)$ are the roots that lie in the right-hand plane, and for Butterworth they are seen to lie on a circle with radius $\Omega_c$.

The bilinear transformation to obtain a digital filter $H(z)$ is a substitution in $H_a(s)$ of $s$ to

$$s = \frac{1-z^{-1}}{1+z^{-1}}$$

so that $H(z) = H_a(s)$ with the above substitution. This transforms all the poles (following the same substitution) and also zeros are introduced. The "prewarping" function $\Omega = \tan(\omega/2)$ also follows from this relation, upon setting $s = j\Omega$ and $z = e^{j\omega}$.

In Matlab, the computation of the poles and the bilinear transformation are combined in a function `butter`:

```
[b,a] = butter(N, F_c/F_s*2);
```

This provides the coefficients of the numerator and denominator polynomial, $H(z) = B(z)/A(z)$.

The roots of a polynomial with coefficients specified in a vector `b` (or `a`) are obtained using `roots(b)`, but plotting these roots along with the unit circle is more easily done using

```
zplane(b,a);
```

- (report 21) Compute the filter coefficients `b,a` and plot the poles and zeros using `zplane`.

- Determine the first 100 coefficients of the filter impulse response (apply the filter to an impulse):

```
h = filter(b,a, [1, zeros(1,99)]);
plot(h);
```

- (report 22) Plot the frequency response using `H = fft(h)`. Also plot the phase response using `unwrap(angle(H))`. Does the filter meet the requirements? Did the $-3$ dB frequency end up in the right place?

  Remark: the frequency response can also be determined by evaluating $b(z)/a(z)$ for $z = e^{j\omega}$, and several values for $\omega$. This is implemented by the function `freqz`:

```
[H,omega] = freqz(b,a); % evaluate the transfer function (omega is vector)
F = omega/(2*pi) * Fs;
plot(F, abs(H));
```

  To directly obtain a plot using a frequency axis in Hz, you can also specify:

```
[H,F] = freqz(b,a,[],Fs);
```

- Apply the filter and listen to the result:

```
y = filter(b,a,x);
soundsc(y);
```

- What are the differences between this filter and the various FIR filters that we obtained? E.g., filter order, linear phase yes/no.

  Can you hear any differences?

## 6.4 ELLIPTIC FILTER DESIGN

More accurate than a Butterworth filter is an elliptic filter, which has both poles and zeros. The design cannot be done by hand, but in Matlab the process is quite similar to that of a Butterworth filter:

```
Fs = 8192; % sample frequency
F0 = 1000; % passband frequency
F1 = 1500; % stopband frequency
R0 = 3; % max passband damping (dB)
R1 = 40; % min stopband damping (dB)

[N, Wp] = ellipord(F0/Fs*2, F1/Fs*2, R0, R1)
[b,a] = ellip(N,R0,R1,Wp)
```

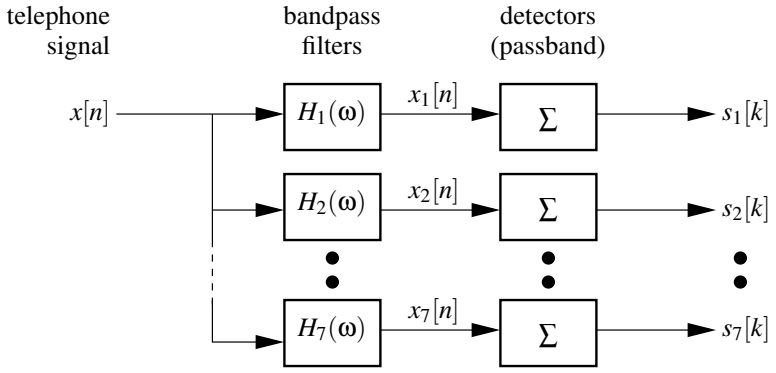



**Figure 6.1**. Filter banks for the detection of a touchtone signal.

- (report 23) For our design specifications, compute the filter coefficients `b` and `a`, plot the location of the poles and zeros using `zplane`, and plot the impulse response and frequency response (amplitude and phase).

  What are differences compared to the Butterworth design?

The elliptic filter design leads to filters that have a much lower order than the Butterworth filters, and is often used in practice. You should always verify whether the resulting filter is stable! For difficult design requirements, it sometimes happens that `ellipord` returns a filter order $N$ that is too small, after which the subsequent `ellip` has been seen to return an unstable filter. If this happens, you should increment $N$.

## 6.5 ASSIGNMENT: TOUCHTONE DETECTOR

In Section 3.7, we have looked at the touch-tone telephone dialing signal. Suppose we want to make a Matlab decoder for that signal. The system uses 7 distinct frequencies, 697, 770, 852, 941, 1209, 1336, 1477 Hz. Recall that our sampling rate was $F_s = 8192$ Hz.

For a decoder, we could design bandpass filters for every frequency band, 7 in total. We then apply each of these filters, and for each or the outputs detect if a frequency signal was present or not, as a function of time. See Fig. 6.1.

- What are the design parameters of each bandpass filter? Compute the center frequencies $\omega_i$ (or normalized frequencies $f_i$), and the required relative bandwidth $B_i = (f_{i,\max} - f_{i,\min})/f_i$, where $f_{i,\max}$ is the maximal frequency that the filter should pass, and $f_{i,\min}$ the minimal frequency.

Instead of designing 7 filters, it may be easier to design a single lowpass filter, and transform that into the required passband filters. Anyway, they all look very similar, only the center frequencies are different. However, we will look at an alternative that is even simpler.

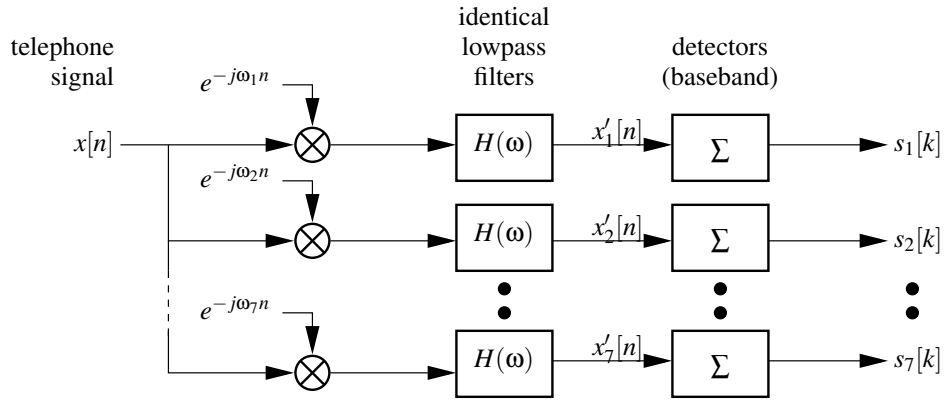



**Figure 6.2**. Demodulation solution for the detection of a touchtone signal.

The output of the $i$th filter is a sinusoid $\cos(\omega_i n)$ at the corresponding frequency $\omega_i$ (ignoring a possible phase offset), switched on and off by a switching pattern $s_i[n] \in \{0,1\}$. In the end we are only interested in the switching pattern, which can be considered a "baseband signal" (that concept will appear in the Telecom B course). The data model of the $i$th tone is thus

$$x_i[n] = s_i[n]\cos(\omega_i n)$$

The detection of $s_i[n]$ from $x_i[n]$ may be done in several ways, but the best is to remove the modulation by the frequency; this is called demodulation. Inverting a "cos" function directly is not recommended as it frequently becomes zero, but if we write $\cos(\omega_i n) = (e^{j\omega_i n} + e^{-j\omega_i n})/2$ we may demodulate $x_i[n]$ simply by multiplying it with $e^{-j\omega_i n}$:

$$x_i'[n] := x_i[n]e^{-j\omega_i n} = s_i[n]\frac{1+e^{-2j\omega_i n}}{2}.$$

This signal consists of the baseband signal $s_i[n]$ at DC, and a similar term at twice the original frequency (but negative). The latter term is easily removed using a lowpass filter $H(\omega)$. If we apply the same demodulation to $x[n] = \sum_i x_i[n]$ which is the original touch-tone signal containing all tones, then we see that after lowpass filtering, still only $s_i[n]$ results.

(Note that $x_i'[n]$ is complex. In our case this does not really matter, but in general it can be avoided by also modulating $x_i[n]$ with $e^{-j\omega_i n}$ and adding the results.)

The complete design is shown in Fig. 6.2.

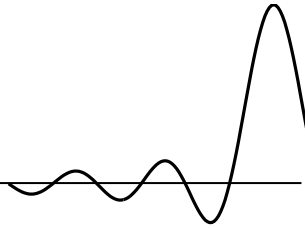
- (report 24) The assignment is to implement and test this design.

  What is the specification for the lowpass filter $H(\omega)$? Should you use a linear-phase filter or phase doesn't matter? What is the resulting filter design? Filter-order? Time-domain impulse response? Verify that the time-domain response is sufficiently compact (why is that necessary?).

Test the demodulation and filtering to one of the "hard" signals in the provided `.mat` files. Show your results (time-domain signals in 7 separate plots). A simple energy detection should then lead to the desired $s_i[n]$ switching patterns. (We will not discuss the detectors here and you do not have to implement them.)

*Chapter* **7**

# LABDAY 6: SAMPLING



## Contents



Digital signal processing starts with sampling. An analog signal is discretized in time and as a consequence the spectrum becomes periodic, which leads to aliasing if we are not careful. Today we look at some of the details.

Because analog signals are poorly reproduced and require additional hardware, we will instead work with signals that are already sampled, but at a very high rate (48 kHz). The same issues with aliasing occur if we want to convert such a signal to a lower rate, e.g., 8 kHz. We will look at such resampling operations.

Image processing is a special case of signal processing, and we can also apply the resampling theory here, to scale an image to a different size.

**Learning objectives** Understanding how sampling modifies signal spectra; the effects of downsampling (aliasing) and upsampling (interpolation)

**Preparation** Read the chapter so you know what to expect.

**What is needed**

– PC with a built-in loudspeaker;

– Matlab audio test signals and test images

## 7.1 SAMPLING, ALIASING AND DOWNSAMPLING

In the theory of sampling, we start with a continuous-time signal (also called an analog signal), $x_a(t)$, and sample it uniformly at a rate $F_s$ (in hertz). The corresponding sample period is $T_s = 1/F_s$ (in seconds). The discrete-time signal (also called digital signal) is $x[n] = x_a(nT_s)$. Generally a digital signal is also quantized in amplitude, but we do not discuss that here.

Sampling reduces information and gives rise to *aliasing*: different continuous-time signals result in the same discrete-time signal. Consider e.g., a complex sinusoid,

$$x_a(t) = e^{j\Omega t} = e^{j2\pi Ft}$$

After sampling, this becomes

$$x[n] = e^{j\Omega T_s n} = e^{j2\pi FT_s n} = e^{j2\pi fn} = e^{j(\omega n}$$

where $f = FT_s$ and $\omega = \Omega T_s = 2\pi FT_s$. Because of periodicity, the fundamental interval of values for $f$ is $[0,1]$ and for $\omega$ it is $[0,2\pi]$; signals with frequencies outside this interval are mapped to a signal with a frequency within this interval. See figure 7.1. Usually, we take fundamental intervals $f \in [-1/2, 1/2]$ and $\omega \in [-\pi, \pi]$.

More in general, if an analog signal with a spectrum $X_a(\Omega)$ is sampled, then the spectrum of the digital signal $x[n]$ is

$$X(\omega)|_{\omega=\Omega T_s} = \frac{1}{T_s}\sum_k X_a(\Omega - k\Omega_s), \qquad \Omega_s = 2\pi/T_s$$

As all digital signals, it is periodic in $\omega$ with period $2\pi$, this is obtained by summing up periodic shifts of $X_a(\Omega)$, shifted by $\Omega_s = 2\pi/T_s$ (or in terms of hertz by $F_s = 1/T_s$). The summation leads to aliasing.

This leads to the Nyquist condition: a signal should be sampled at at least twice its highest frequency to be represented uniquely in the fundamental interval, without aliasing. The Shannon theorem is saying that such signals can be converted back to an analog signal without loss of information.

### Aliasinging due to downsampling

To study aliasing in Matlab, we could play with the microphone and the A/D converter attached to the microphone input of the PC. But the ADC has a built-in anti-aliasing filter that will mask the effects. However, we can also observe aliasing if we downsample a digital signal. Downsampling means reducing the sample rate by throwing away a fraction of the samples.

- In Matlab, generate a high-frequency signal:

```
Fs = 48000; % sample rate
Fi = 7000; % high frequency
N = Fs/2;
```

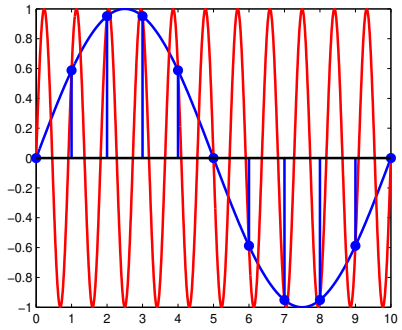



**Figure 7.1**. Aliasing. The red signal has frequency $\omega = 1.1 \cdot 2\pi$, the blue signal frequency $0.1 \cdot 2\pi$. If we sample only at integer times, samples of the red signal cannot be distinguished from samples of the blue signal.

```
n = [0:N-1]; % half a second of data
x = sin(2*pi*Fi/Fs*n);

soundsc(x,Fs);
```

Next, generate a new signal which consists of every second sample of $x[n]$:

```
Fs1 = Fs/2;          % = 24 kHz
x1 = x(1:2:end);     % downsample by 2

soundsc(x1,Fs1);
```

This signal is downsampled by a factor of 2, but sounds the same. Next, we downsample the original signal by a factor of 4:

```
Fs2 = Fs/4;         % = 12 kHz
x2 = x(1:4:end);

soundsc(x2,Fs2);
```

This time the signal sounds differently. Can you explain why?

- Make plots in time domain:

```
clf;
plot(n(1:100),x(1:100));
```

```
hold on;
plot(n(1:100),x(1:100),'.');
n2 = n(1:4:end);
plot(n2(1:25),x2(1:25),'ro');
```

- Make also plots in frequency domain. Take care that you use the correct frequency axis.

If we downsample by a factor of $M$, then the new sample frequency is $F_s' = F_s/M$. If the highest frequency in the signal is larger than half this frequency, then the normalized frequency $f_i = F_i/F_s'$ will be outside the interval $[-1/2, 1/2]$, or $\omega_i \in [-\pi, \pi]$. The frequency will be equivalent to a new frequency $\omega_i'$ within the interval, and it is obtained by adding/subtracting multiples of $2\pi$ to $\omega_i$, or $f_i'$ is obtained by adding/subtracting multiples of 1 to $f_i$. The corresponding 'analog' frequency $F_i' = f_i' F_s'$.

In the above case, we started with a sinusoid at $F_i = 7$ kHz. The new sample frequency was $F_s' = 12$ kHz, and $f_i = 7/12 = 0.58$, outside the interval $[-1/2, 1/2]$. The value within the interval is $f_i' = 0.58 - 1 = -0.41$, and the analog frequency is $F_i' = -5$ kHz. The component of the signal at $-7$ kHz is mapped to 5 kHz. Thus, the downsampled signal sounds like it is 5 kHz. This is the effect of aliasing.

- Let's generate a chirp signal (which has a continuously changing frequency),

```
Fs = 48000;
Fmin = 0; Fmax = 12000;
N = 2*Fs;
n = [0:N-1];
f = linspace(Fmin,Fmax,N)/Fs;
x = sin(2*pi*f.*n);

soundsc(x,Fs);
```

  Now, downsample this signal by a factor of 4,

```
Fs2 = Fs/4;
x2 = x(1:4:end);

soundsc(x2,Fs2);
```

- (report 25) Make a time-frequency plot of the original signal, and of the downsampled signal. Can you explain what you see?

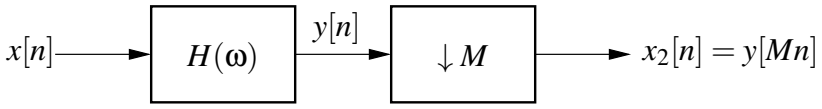



**Figure 7.2**. Filtering followed by downsampling.

### Anti-aliasing filter

In processing audio signals, it frequently occurs that we need to adjust the sample frequency. E.g., studio-quality audio is sampled at 48 kHz, whereas if we want to transmit this over a fixed-line telephone line, it has to be limited to a bandwidth of 4 kHz, i.e., the maximal sample rate is 8 kHz. The signal will have to be downsampled by a factor $M = 6$.

Obviously there will be information loss. However, it is also important to avoid aliasing: higher frequencies should never be mapped to lower frequencies, because then the downsampled signal will sound very bad. For this, we need to apply a lowpass filter prior to downsampling. The lowpass filter should be such that the highest frequency in the signal after lowpass filtering is less than half the new sample rate. Such a lowpass filter is called an anti-aliasing filter.

Figure 7.2 shows the process of lowpass filtering followed by downsampling of a factor $M$. Figure 7.3 shows the effect of this process on a signal $x[n]$. The spectrum after downsampling becomes periodic with period determined by the new sampling frequency, which is a factor $M$ lower than the original sampling frequency. Without lowpass filtering, this leads to aliasing. With ideal filters, the lowpass filter should have a cut-off frequency $\omega_c = \pi/M$. With non-ideal filters, we need to allow for a transition band.

- Design a linear-phase lowpass filter $H(z)$ with cut-off frequency $\pi/M$. E.g., use `h = fir1(N,1/M)` with $N$ sufficiently large. This will have a damping of 6 dB at the cut-off frequency, which is perhaps ok for a quick design.

  Alternatively, you could make a design with the following specifications:

  | | |
  |---|---|
  | Sample frequency | $F_s = 48000$ Hz |
  | Cut-off frequency (-3dB) | $F_c = 3.5$ kHz |
  | Stopband frequency | $F_1 = 4$ kHz |
  | Stopband damping | 40 dB |

- Load the audio signal $x[n]$ from Brightspace `T4_ca.wav` (castagnettes). Its sample rate is 48 kHz. Play the signal on the loudspeaker.

- Construct a signal $x_1[n]$ which is downsampling $x[n]$ by a factor 6 by simply taking every 6th sample. The new sample frequency is $F_s' = 8$ kHz.

  Play this signal on the loudspeaker. Does it sound right?

- Apply the lowpass filter $H(z)$ to the original signal $x[n]$. Next, downsample by a factor 6, call the result $x_2[n]$.

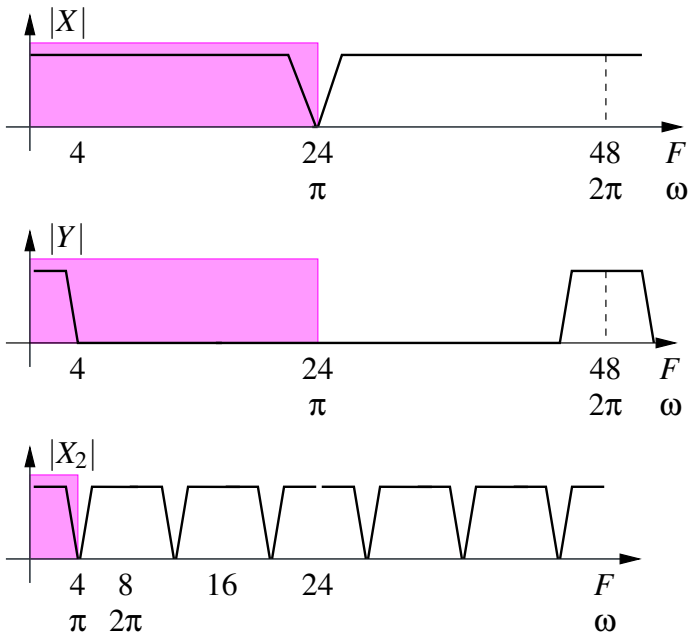



**Figure 7.3**. The effect of filtering followed by downsampling on the spectrum of a signal. The pink box denotes the fundamental interval $[0, \pi]$.

Play this signal on the loudspeaker. Does it sound right? (Save this signal for later use.)

- If you like, you can repeat with another signal, `T7_gs.wav` (german speech), where the effect of aliasing is also very noticeable, especially in the /ts/ sounds.

## 7.2 SIGNAL RECONSTRUCTION, UPSAMPLING AND INTERPOLATION

Shannon's sampling theorem says that it is possible to perfectly reconstruct an analog signal $x_a(t)$ from its samples $x[n] = x_a(nT_s)$, provided the Nyquist condition holds. This ideal D/A converter consists in theory of a device that replaces samples (numbers in a computer) $x[n]$ by continuous-time impulses $x[n]\delta(n-nT_s)$, followed by an ideal analog lowpass filter $H_r(\Omega)$. The effect on the spectrum is

$$X_a(\Omega) = X(\Omega T_s)H_r(\Omega)\,.$$

$X(\omega)$ is the spectrum of the digital signal. It is periodic with period $2\pi$. The spectrum of the impulse train is $X(\Omega T_s)$, it is the same periodic spectrum, now with analog frequency $\Omega = \omega/T_s$. The lowpass filter $H_r(\Omega)$ removes all the periodic duplicates in the spectrum. It should cut off at $\Omega_s/2$ so that only the interval $\Omega \in [-\Omega_s/2, \Omega_s/2]$ remains. This corresponds to $\omega \in [\pi, \pi]$.

In time-domain, the filter operation is a convolution

$$x_a(t) = \sum x[n]h_r(t-nT_s) \tag{7.1}$$

where $h_r(t)$ is the impulse response of $H_r(\Omega)$,

$$H_r(\Omega) = \begin{cases} T_s & |\Omega| \le \Omega_s/2 \\ 0 & |\Omega| > \Omega_s/2 \end{cases} \qquad \Leftrightarrow \qquad h_r(t) = \frac{\sin(\pi t/T_s)}{\pi t/T_s} =: \mathrm{sinc}(t/T_s)$$

Thus, (7.1) becomes

$$x_a(t) = \sum x[n]\mathrm{sinc}(\frac{t-nT_s}{T_s}) \tag{7.2}$$

which is a sum of sinc-functions, each properly delayed and scaled.

### Interpolation

Let's see how this equation works. For lack of a D/A converter, we will model $x_a(t)$ by a densely sampled digital signal.

- Generate 20 samples of some digital signal $x[n]$. For example,

```
x = randn(20,1);
```

or

```
omega_i = 0.4;
n = [0:19]';
x = sin(omega_i*n);
```

- Let's take $T_s = 1$ second and generate the sinc function $h_r(t)$ with a resolution of 10 samples/second on the interval $t \in [-5,5]$:

  ```
  t = -5 : 0.1 : 5;
  hr = sin(pi*t)./(pi*t);
  ```

  There is a problem for $t = 0$, the corresponding value should be $h_r(0) = 1$. It is easier to use the Matlab function `sinc(t)`.

  Make a plot of this signal, `plot(t,hr)`.

- (report 26) To implement (7.2), we replace $x[n]$ by an "impulse train", still in the digital domain but with a resolution of 10 samples/second. This is done by inserting 9 zeros in between every two samples of $x[n]$:

  ```
  xx = kron(x, [1; zeros(9,1)]);
  ```

  The Kronecker product replaces every sample $x[n]$ by a vector $x[n][1,0,\cdots,0]^T$; it was assumed that **x** is a column vector.

  Next, filter this signal by $h_r(t)$:

  ```
  xa = conv(xx,hr);
  ```

  Compute the correct time axis for this signal. It should start at $t = -5$ and have increments of 0.1.

  Make a plot of this signal. Also plot the original samples of $x[n]$. Do you see that $x_a(t)$ interpolates $x[n]$? What property of the sinc-function makes this happen?

  Your figure could look like the blue curve in Fig. 7.4, where the red curves are $x[n]h_r(t - nT_s)$.

- (report 27) Other interpolating functions (sampled to 10 samples like $h_r(t)$) are:

  ```
  h0 = ones(10,1);    % zero order hold, time axis is t0 = -0.5:0.1:0.4
  h1 = bartlett(21);  % linear interpolation, time axis t1 = -1:0.1:1
  ```

  Show the impulse responses of these filters. Make plots of the effect of these filters on $x[n]$ in time domain and frequency domain; compare to the effect of $h_r(t)$.

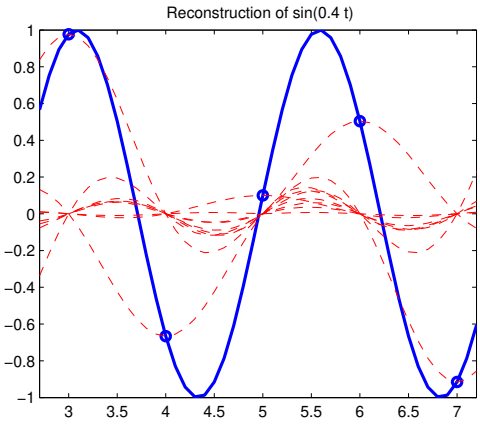



**Figure 7.4**. Reconstruction of a sampled signal using a sum of sinc functions.

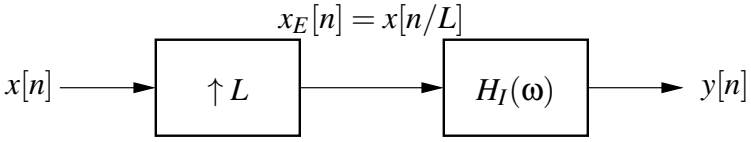



**Figure 7.5**. Upsampling by an integer factor $L$.

### Upsampling

In the previous exercise, we have effectively upsampled the original signal $x[n]$ by a factor of 10. We did this by inserting 9 zeros between each sample, followed by ideal lowpass filtering.

We can of course do this for any integer factor $L$. In this case, we insert $L-1$ zeros between every two samples of $x[n]$:

$$x_E[n] = \begin{cases} x[k], & n = kL \\ 0, & \text{elders} \end{cases}$$

The data rate of the "expanded" signal increases by a factor $L$. There is no loss of information. The effect on the spectrum is

$$X_E(z) = X(z^L), \qquad X_E(\omega) = X(\omega L).$$

This results in a compression of the spectrum by a factor $L$. However, if we express the spectrum in terms of $\Omega$ or $F$ (hertz), then nothing has changed. The only thing that changes is the relation between $\omega$ and $\Omega$, namely $2\pi$ maps to the new (higher) sample frequency $F_s' = LF_s$.

In the interval $[-\pi, \pi]$, upsampling results in $L-1$ additional copies of $X(\omega)$. We can remove the extra copies by lowpass filtering, i.e., a filter

$$H_I(\omega) = \begin{cases} L, & |\omega| < \pi/L \\ 0, & \pi/L < |\omega| < \pi \end{cases} \qquad \Leftrightarrow \qquad h_I[n] = \text{sinc}(\frac{n}{L}).$$

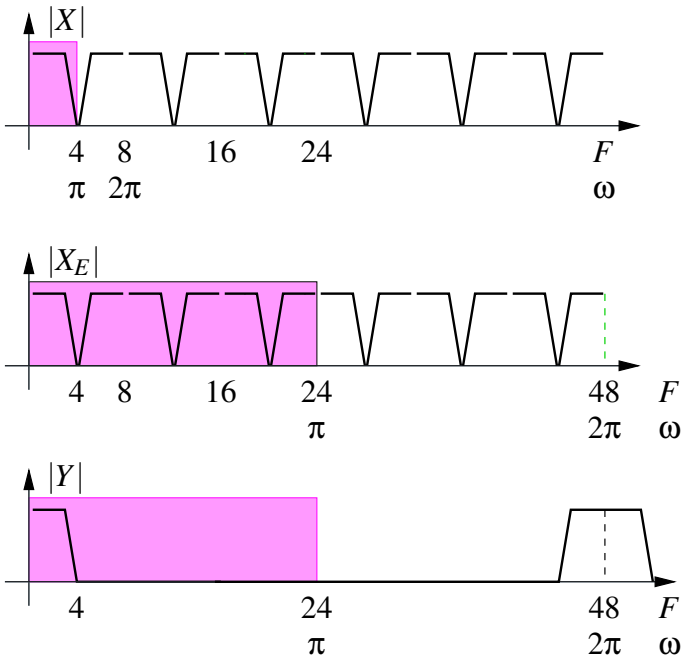



**Figure 7.6.** Effect of upsampling by a factor $L = 6$ followed by lowpass filtering on the spectrum of an audio signal.

The filter is called an "image reject" or interpolating filter. The effect of the filter in time domain is that the zeros that were inserted are replaced by nicely interpolated values, just as in the previous exercise. This is the same as first creating an analog signal (ideal D/A conversion), followed by sampling at the new sample rate.

Upsampling followed by lowpass filtering can be used to increase the sample rate of an audio signal, e.g., from 8 kHz to 48 kHz, as in figure 7.2. No information is being added, so the result is likely to sound the same.

- Implement upsampling by a factor $L = 6$. You can use

  ```
  xx = kron(x, [1; zeros(L-1,1)]);
  ```

  (assuming **x** is a column vector), and you need interpolation using a lowpass filter with stopband frequency $\pi/L$. E.g., use `fir1(N,1/L)` with $N$ sufficiently large.

- Apply to the 8 kHz signal you created out of the `T4_ca.wav` signal in Section 7.1. Listen to both the upsampled signal $x_E[n]$ and the interpolated (filtered) signal $y[n]$; use `soundsc` with sample rate `48e3`.

  The latter signal should sound like the 8 kHz signal, even if it is now played at 48 kHz.

## 7.3 ASSIGNMENT: IMAGE SCALING

An image can be regarded as a 2-D sampled signal, where samples are now called pixels. Many operations such as filtering and spectrum analysis (DFT) can equally well be applied to images. In many cases operations can be applied independently to rows and columns: we regard the image as a matrix $\mathbf{X}$, and apply matrix operations $\mathbf{A}$ and $\mathbf{B}$ to obtain $\mathbf{X}' = \mathbf{AXB}$. In the case of linear space-invariant filters, $\mathbf{A}$ and $\mathbf{B}$ are Toeplitz matrices that implement convolutions.

What is different is that causality does not play a role in image processing: all pixels are available, and we might easily "go back in space", whereas this is more difficult in time. On the other hand, edge effects are important: if we have an image of a certain size, then applying a filter (convolution) to the image, the convolution will take the pixels outside the specified region as zero (whereas in reality they are not zero but simply not specified), and the convolution will make the image bigger. Thus, if we apply a 2-D FIR filter with impulse response running from $-L$ to $L$ in each direction, then the image will be larger by $L$ pixels on all sides, but due to missing values, the borders are not reliable and the resulting image should actually be truncated by $2L$ pixels on each side. Keeping track of "valid pixels" is an important chore in image processing.

Our aim in this assignment is to scale (resample) an image. As before, if we extend the image, we need to apply interpolation, whereas if we shrink the image, we need to do anti-aliasing. Aliasing in images is noticeable in diagonal lines, which will appear jagged (stair-case effect) if you simply drop pixels, or in patterned (high-frequency) regions, which might show moiré effects when downsampled. This is why TV newsreaders do not wear pin-striped suits.

Matlab has a function for image scaling, `imresize`, which uses bilinear (first-order) or bicubic (second-order) interpolation, because these are FIR filters of limited orders. This will limit the edge effects at the expense of some aliasing. We will try to see if a higher-order lowpass filter will give better results.

Matlab has a list of demo images, see `help imdemos`. Let's work with `cameraman.tif`, which is a $256 \times 256$ often-used black-white test image:

```
X = imread('cameraman.tif');
imagesc(X);
X = double(X); % convert image from uint8 to double
```

The function `imagesc` shows the matrix $\mathbf{X}$ as an image (using scaling to keep the numbers in range). For a matrix containing integers, 0 corresponds to black and 255 to white in the default colormap; for a matrix containing doubles, 0 corresponds to black and 1 corresponds to white. The colormap can be set using `colormap('gray')`. The typecasting to 'double' is an annoyance that is needed for compatibility of multiplications.

The assignent is to take this image, and to scale it by a factor $3/2$ and by a factor $2/3$.

In the first case, you would first upsample by a factor $L = 3$, then apply an interpolating filter, followed by downsampling by a factor $M = 2$. The downsampling requires first an anti-aliasing filter. Since both filters are lowpass filters, they can be combined and we need only a single lowpass filter. The strongest

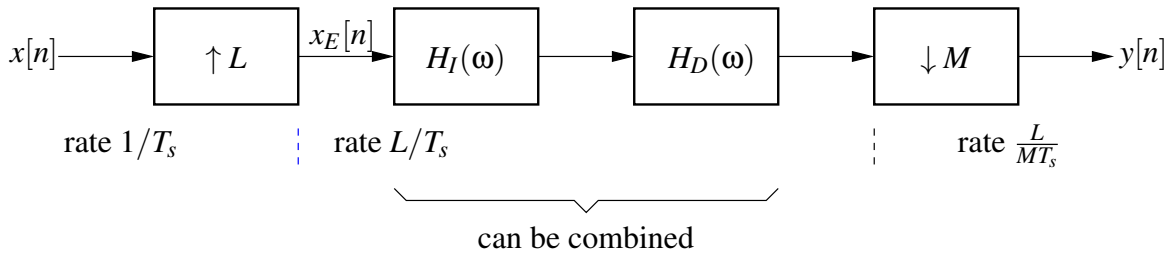



**Figure 7.7**. Sample rate conversion by a rational fraction $L/M$.

condition holds: the cut-off frequency is $\omega_c = \min(\pi/M, \pi/L)$, which in this case is $\omega_c = \pi/3$.

In the second case, we would first upsample by a factor $L = 2$, followed by lowpass filtering, then downsampling by a factor $M = 3$. The lowpass filter has the same cut-off frequency as before, $\omega_c = \pi/3$.

Upsampling by a factor $L$ on an image $\mathbf{X}$ can be conveniently implemented in Matlab by

```
E = zeros(L,L); E(1,1) = 1;
XE = kron(X, E);
```

The Kronecker product replaces every entry $x_{i,j}$ of $\mathbf{X}$ by a matrix $x_{i,j}\mathbf{E}$. The matrix $\mathbf{E}$ has one nonzero entry; formally that should be the entry in the center, but here the corner entry is taken as this works better with a causal lowpass filter (anyway, we have to keep track of the validity of the edges).

- Design a lowpass filter with $\omega_c = \pi/3$. We prefer this to be a linear-phase FIR filter (why?), with a filter order $N$ which is a multiple of $L$. The filter order should be small to limit the edge effects, although too small will not result in a good lowpass filter. E.g., use `fir1(N,1/3)` with $N = 6$, 9 or 12.

  Compare to a triangle-shape filter (`bartlett(7)`) that provides a simple linear interpolation between the nonzero samples.

- (report 28) Scale the image by a factor $L/M = 3/2$, following the scheme of Fig. 7.7, where the filter needs to be applied to all rows and then all columns of the image.

  Plot the result. Compare to the result of Matlab `imresize(M,3/2,'bilinear')`. Do you see differences?

- (report 29) Scale the image by a factor $L/M = 2/3$, following the scheme of Fig. 7.7.

  Plot the result. Compare to the result of Matlab `imresize(X,2/3,'bilinear')`. Do you see differences?

The cameraman picture is not of high quality, and artifacts in the original image are likely to dominate any differences between your method and the standard Matlab function. Other suitable B/W test images included in Matlab are e.g., `moon.tif`, `AT3_1m4_01.tif`.

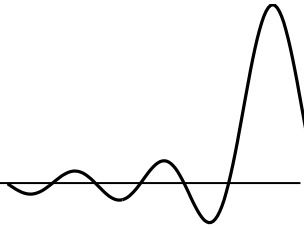
Matlab implements image scaling with an arbitrary scaling factor by reconstructing the analog "signal" $x_a(t)$ (in fact 2-D), using an appropriate interpolation function such as a linear, cubic or sinc function, but only evaluating this at the samples needed in the new image.

---

The end! Hope you enjoyed the prakticum; please give feedback. See you at EPO4.

*Appendix* **A**

# DECONVOLUTION IN TIME DOMAIN



Channel estimation starts with understanding the inversion of a transfer function. Suppose we have a filter $X(z)$, and wish to have a filter $G(z)$ that "undoes" this filter. In the $z$-domain, that is easy enough: the inverse filter is $G(z) = 1/X(z)$. It is known as an equalizer, and applying $G(z)$ to undo a convolution by $X(z)$ is called deconvolution.

Suppose that $X(z)$ is a rational filter,

$$X(z) = \frac{B(z)}{A(z)}$$

Then the zeros of $X(z)$ become the poles of $G(z)$. Obviously, we require $G(z)$ to be a stable causal filter. Thus, the zeros of $X(z)$ should lie within the unit circle. A transfer function which satisfies this requirement is called *minimum phase*. In this case, the corresponding impulse response $g[n]$ is causal and

$$X(z)G(z) = 1 \qquad \Leftrightarrow \qquad x[n] * g[n] = \delta[n]$$

If $X(z)$ is not minimum phase, a stable causal inverse does not exist. However, very often it is sufficient if $x[n] * g[n] = \delta[n-K]$ for some integer $K > 0$: a possible delay is OK. This allows to approximate functions that do not have a stable causal inverse.

Only if $X(z)$ is an allpole function will $G(z)$ be an FIR filter. Thus, in general $G(z)$ will not be FIR. However, it is possible to approximate stable causal transfer functions by a FIR filter, by just using a large number of taps of its impulse response.

- Consider $X(z) = 1 - \frac{1}{2}z^{-1}$. What is the impulse response $x[n]$? What is the location of the zero of $X(z)$? Is this a minimum-phase transfer function?

  What is the inverse $G(z)$? Is it stable? What is $g[n]$?

- Idem for $X(z) = 1 - 2z^{-1}$.

### Inversion of a transfer function via matrix inversion

Assume that $X(z)$ has a stable causal inverse $G(z)$. If we know the time domain sequence (impulse response) $x[n]$, how can we compute $g[n]$? We write the convolution $x[n] * g[n] = \delta[n]$ in matrix form, for

$n \geq 0$:

$$x[n] * g[n] = \delta[n] \quad \Leftrightarrow \quad \sum_{k=0}^{\infty} x[n-k]g[k] = \delta[n]$$

$$\Leftrightarrow \quad \begin{bmatrix} \boxed{x[0]} & & & & \mathbf{0} \\ x[1] & x[0] & & & \\ x[2] & x[1] & x[0] & & \\ x[3] & x[2] & x[1] & x[0] & \\ x[4] & x[3] & x[2] & x[1] & x[0] \\ \vdots & \vdots & & & \ddots \end{bmatrix} \begin{bmatrix} \boxed{g[0]} \\ g[1] \\ g[2] \\ g[3] \\ g[4] \\ \vdots \end{bmatrix} = \begin{bmatrix} \boxed{1} \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \end{bmatrix}. \tag{A.1}$$

(The square box indicates the location of the $(0,0)$ entry of the matrix.) The matrix has a Toeplitz form: it is constant along diagonals. Such matrices often occur when you write down the equations for linear time-invariant systems.

These are infinite-size matrices, and not suitable for computations. If we approximate $G(z)$ by a finite length FIR filter, of length $L$, then we obtain the equations

$$x[n] * g[n] = \delta[n] \quad \Leftrightarrow \quad \sum_{k=0}^{L-1} x[n-k]g[k] = \delta[n]$$

$$\Leftrightarrow \quad \begin{bmatrix} \boxed{x[0]} & & & \mathbf{0} \\ x[1] & \ddots & & \\ \vdots & \ddots & x[0] & \\ x[L-1] & \ddots & x[1] & x[0] \\ x[L] & \ddots & x[2] & x[1] \\ \vdots & \ddots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \boxed{g[0]} \\ g[1] \\ \vdots \\ g[L-1] \end{bmatrix} = \begin{bmatrix} \boxed{1} \\ 0 \\ 0 \\ 0 \\ \vdots \end{bmatrix} \tag{A.2}$$

Now the matrix has an infinite number of rows, but only $L$ columns. Let us truncate the matrix to be square $L \times L$:

$$\begin{bmatrix} \boxed{x[0]} & & & \mathbf{0} \\ x[1] & \ddots & & \\ \vdots & \ddots & x[0] & \\ x[L-1] & \cdots & x[1] & x[0] \end{bmatrix} \begin{bmatrix} \boxed{g[0]} \\ g[1] \\ \vdots \\ g[L-1] \end{bmatrix} = \begin{bmatrix} \boxed{1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \Leftrightarrow \quad \mathbf{X}\mathbf{g} = \mathbf{e}_1$$

Since it is square, we can invert the matrix (e.g., using back substitution), assuming $x[0] \neq 0$. Note that only $L$ samples of $x[n]$ are used. If a stable causal equalizer of length $L$ exists, (and $x[0] \neq 0$) we will find it in this way, but we will not know from these equations if such an equalizer exists or not. In general, we find only an approximation.

*Example 1:* Let $X(z) = 1 - \frac{1}{2}z^{-1}$: a minimum-phase function. We expect $G(z) = 1 + \frac{1}{2}z^{-1} + \frac{1}{4}z^{-2} + \frac{1}{8}z^{-3} + \cdots$. In matrices:

$$\mathbf{X} = \begin{bmatrix} 1 & & & \mathbf{0} \\ -0.5 & 1 & & \\ & -0.5 & 1 & \\ \mathbf{0} & & -0.5 & 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{X}^{-1} = \begin{bmatrix} 1 & & & \mathbf{0} \\ 0.5 & 1 & & \\ 0.25 & 0.5 & 1 & \\ 0.125 & 0.25 & 0.5 & 1 \end{bmatrix}$$

The equalizer $\mathbf{g}$ (truncated to $L$ samples) corresponds to the first column of $\mathbf{X}^{-1}$.

*Example 2:* Let $X(z) = 1 - 2z^{-1}$: a non-minimum phase function. A causal expansion of $1/X(z)$ gives $G(z) = 1 + 2z^{-1} + 4z^{-2} + 8z^{-3} + \cdots$: this is an unstable transfer function. In matrices:

$$\mathbf{X} = \begin{bmatrix} 1 & & & \mathbf{0} \\ -2 & 1 & & \\ & -2 & 1 & \\ \mathbf{0} & & -2 & 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{X}^{-1} = \begin{bmatrix} 1 & & & \mathbf{0} \\ 2 & 1 & & \\ 4 & 2 & 1 & \\ 8 & 4 & 2 & 1 \end{bmatrix}$$

For larger matrices, we will quickly run into numerical problems.

If $X(z)$ is not minimum phase (as in the above example), then $G(z)$ is not stable, and we do not have that $g[n] \to 0$ for large $n$. Truncation to length $L$ will not be accurate. The problem stems from the fact that in (A.1), we forced $g[n]$ to be causal. A better solution is to start computing the sequence at $g[-K]$, for some $K > 0$. This allows to model the unstable part of $g[n]$ as a stable but anti-causal part, which is approximated by $K$ FIR coefficients. The equations to solve are

$$\begin{bmatrix} x[0] & & & \mathbf{0} \\ x[1] & \ddots & & \\ \vdots & \ddots & x[0] & \\ x[L-1] & \cdots & x[1] & x[0] \end{bmatrix} \begin{bmatrix} g[-K] \\ \vdots \\ \boxed{g[0]} \\ \vdots \\ g[-K+L-1] \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \boxed{1} \\ 0 \end{bmatrix}$$

where the matrix is $L \times L$, and the vector on the right hand side has $L$ entries and the '1' entry appears on the $K$th position.

*Example 3:* If $X(z) = z^{-1}$, then we would like to obtain $G(z) = z$, but this is anti-causal. Choosing $K = 1$ and $L = 4$, we obtain the matrix equation

$$\begin{bmatrix} 0 & & & \mathbf{0} \\ 1 & 0 & & \\ & 1 & 0 & \\ \mathbf{0} & & 1 & 0 \end{bmatrix} \begin{bmatrix} g[-1] \\ g[0] \\ g[1] \\ g[2] \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} g[-1] \\ g[0] \\ g[1] \\ g[2] \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Note that the matrix $\mathbf{X}$ is not invertible, but that there still is a valid solution to the system of equations.

As seen in the above example, it is in general not a good idea to reduce the system of equations to a square $L \times L$ matrix. A better solution is to truncate the infinite matrix in (A.2) to a "tall" $N \times L$ matrix:

$$\begin{bmatrix} x[0] & & & \mathbf{0} \\ x[1] & x[0] & & \\ \vdots & x[1] & \ddots & \\ x[N_x-1] & \ddots & \ddots & x[0] \\ 0 & x[N_x-1] & \ddots & x[1] \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \cdots & 0 & x[N_x-1] \end{bmatrix} \begin{bmatrix} g[-K] \\ \vdots \\ \boxed{g[0]} \\ \vdots \\ g[-K+L-1] \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \boxed{1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad \Leftrightarrow \qquad \mathbf{Xg} = \mathbf{e}_K \quad \text{(A.3)}$$

In the above, we assumed that $\mathbf{x}$ has finite length $N_x$. In that case, we can take $N = N_x + L - 1$; if we take a larger $N$, we just introduce rows that are zero.

To obtain the equalizer coefficients $\mathbf{g}$, we need to solve $\mathbf{Xg} = \mathbf{e}_K$. Since $\mathbf{X}$ is a tall matrix, we cannot use the usual inverse. However, if its columns are linearly independent (it has full column rank), there exists a left inverse $\mathbf{X}^\dagger$ such that $\mathbf{X}^\dagger\mathbf{X} = \mathbf{I}$, namely

$$\mathbf{X}^\dagger = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

Here, $\mathbf{X}^T\mathbf{X}$ is square $L \times L$, and invertible if the columns of $\mathbf{X}$ are linearly independent. Note that $\mathbf{X}^T\mathbf{X}$ may be invertible even if $x[0] = 0$.

Using this technique, we can obtain good FIR approximations for the inverse of most filters $X(z)$, specified via $N_x$ samples of their impulse response $x[n]$. Design choices are $L$ and $K$. Generally, $L$ should be large enough so that the truncation of $g[n]$ to $L$ coefficients is accurate. Looking at the equation $\mathbf{g} = \mathbf{X}^\dagger\mathbf{e}_K$, we see that the role of $K$ is to select the $K$th column of $\mathbf{X}^\dagger$.

*Example 4:* Again consider $X(z) = 1 - 2z^{-1}$. We aim for an anticausal but "stable" solution:

$$G(z) = \frac{1}{1-2z^{-1}} = \frac{-\frac{1}{2}z}{1-\frac{1}{2}z} = -\frac{1}{2}z(1 + \frac{1}{2}z + \frac{1}{4}z^2 + \cdots)$$

In matrices ($N = 5, L = 4$),

$$\mathbf{X} = \begin{bmatrix} 1 & & & \mathbf{0} \\ -2 & 1 & & \\ & -2 & 1 & \\ & & -2 & 1 \\ \mathbf{0} & & & -2 \end{bmatrix} \quad \Rightarrow \quad \mathbf{X}^\dagger = \begin{bmatrix} 0.2493 & -0.3754 & -0.1877 & -0.0938 & -0.0469 \\ 0.1232 & 0.0616 & -0.4692 & -0.2346 & -0.1173 \\ 0.0587 & 0.0293 & 0.0147 & -0.4927 & -0.2463 \\ 0.0235 & 0.0117 & 0.0059 & 0.0029 & -0.4985 \end{bmatrix}$$

For these small sizes, it is not easy to see a structure in $\mathbf{X}^\dagger$. For larger $N$ and $L$, it is seen that the lower triangular part vanishes, and the upper triangular part exhibits the expected geometric sequence $[\cdots, -\frac{1}{8}, -\frac{1}{4}, -\frac{1}{2}, \boxed{0}]^T$. The result to which we converge is also obtained if

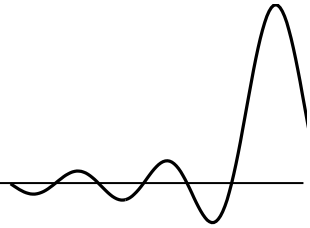
we drop the top row of $\mathbf{X}$:

$$\mathbf{X} = \begin{bmatrix} -2 & 1 & & \mathbf{0} \\ & -2 & 1 & \\ & & -2 & 1 \\ \mathbf{0} & & & -2 \end{bmatrix} \quad \Rightarrow \quad \mathbf{X}^{-1} = \begin{bmatrix} -\frac{1}{2} & -\frac{1}{4} & -\frac{1}{8} & -\frac{1}{16} \\ & -\frac{1}{2} & -\frac{1}{4} & -\frac{1}{8} \\ & & -\frac{1}{2} & -\frac{1}{4} \\ & & & -\frac{1}{2} \end{bmatrix}$$

Conclusions: (1) in theory, we should invert an infinite-size matrix $\mathbf{X}$, (2) in practice, we truncate $\mathbf{X}$ to a finite size; we need to take $\mathbf{X}$ "tall", with $N$ and $L$ sufficiently large, to obtain good results; (3) there are various options for the equalizers, corresponding to the various columns of $\mathbf{X}^{\dagger}$. (Their choice is beyond the scope of this tutorial.)

*Appendix* ***B***

# THE SVD, MATRIX INVERSION AND THE CONDITION NUMBER



**The SVD**

The singular value decomposition should have been presented in the Linear Algebra course (but probably was not). Here is a brief summary. For a given matrix $\mathbf{X}$ of size $m \times n$, where we assume $m > n$, the SVD is defined by

$$\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T, \qquad \mathbf{U} = [\mathbf{u}_1 \cdots \mathbf{u}_m], \quad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \\ \mathbf{0} & \cdots & \cdots & \mathbf{0} \end{bmatrix}, \quad \mathbf{V} = [\mathbf{v}_1 \cdots \mathbf{v}_n]$$

where $\mathbf{U} : m \times m$ and $\mathbf{V} : n \times n$ are orthogonal matrices, and $\boldsymbol{\Sigma}$ is a diagonal matrix of size $m \times n$ containing the singular values in descending order. Note that $\boldsymbol{\Sigma}$ has a block of $m - n$ "zero" rows at the bottom.

The Matlab command to compute this decomposition is

```
[U,S,V] = svd(X)
```

Since $\boldsymbol{\Sigma}$ has $m - n$ rows with zeros, and often $m$ can be very large, it is inefficient to keep so many columns of $\mathbf{U}$ that are anyway not used (they are multiplied by the zeros). Thus, we can also define the "economy-size" SVD, where

$$\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T, \qquad \mathbf{U} = [\mathbf{u}_1 \cdots \mathbf{u}_n], \quad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix}, \quad \mathbf{V} = [\mathbf{v}_1 \cdots \mathbf{v}_n]$$

where $\mathbf{U} : m \times n$ is a tall matrix of the same size as $\mathbf{X}$, and $\mathbf{V}$ and $\boldsymbol{\Sigma}$ are $n \times n$. Note that $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ but $\mathbf{U}\mathbf{U}^T \neq \mathbf{I}$ because it is an $m \times m$ matrix of rank $n$.

The Matlab command to compute this decomposition is

```
[U,S,V] = svd(X,0)
```

where the `0` option asks for the "economy size" SVD. From now on, we will assume this decomposition.

The singular values give important information on the dominant directions in the column span and row span of $\mathbf{X}$. This is seen by writing out the matrix equations, which gives

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{u}_1\sigma_1\mathbf{v}_1^T + \mathbf{u}_2\sigma_2\mathbf{v}_2^T + \cdots + \mathbf{u}_n\sigma_n\mathbf{v}_n^T . \tag{B.1}$$

Each term of the form $\mathbf{u}_k\sigma_k\mathbf{v}_k^T$ is a rank-1 matrix. If $\sigma_1$ is large, then the corresponding component $\mathbf{u}_1\mathbf{v}_1^T$ is dominantly present in $\mathbf{X}$, and $\mathbf{u}_1$ is the dominant direction in the column span of $\mathbf{X}$. In fact, $\mathbf{u}_1\sigma_1\mathbf{v}_1^T$ is the best rank-1 approximation of $\mathbf{X}$ (in the Least Squares sense).

If $\sigma_n$ is zero, then one dimension is missing in the matrix: it is rank deficient by order 1. In general, if $\mathbf{X}$ is of rank $d$, then only $d$ singular values $\sigma_1, \cdots, \sigma_d$ are nonzero. This is also seen from (B.1) because it will then consist of the sum of $d$ rank-1 components. The best rank-$d$ approximation of a matrix $\mathbf{X}$ is obtained by setting $\sigma_{d+1} = \cdots = \sigma_n = 0$.

If $\mathbf{X}$ is full column rank, then the left inverse of $\mathbf{X}$ is $\mathbf{X}^\dagger = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$. Inserting $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, we obtain that this can also be written as

$$\mathbf{X} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T \tag{B.2}$$

which is a slightly more general expression. Essentially, we are inverting the singular values here. We can easily verify that $\mathbf{X}^\dagger\mathbf{X} = \mathbf{I}$ and

$$\mathbf{X}\mathbf{X}^\dagger = \mathbf{U}\mathbf{U}^T$$

If we define $\mathbf{P} = \mathbf{U}\mathbf{U}^T$ then we see that $\mathbf{P}$ is an orthogonal projection, because $\mathbf{P}\mathbf{P} = \mathbf{P}$ and $\mathbf{P}^T = \mathbf{P}$. It is a projection onto the column span of $\mathbf{X}$.

### Connection to the eigenvalue decomposition

If we take the SVD of $\mathbf{X}$ and "square" it to $\mathbf{X}^T\mathbf{X}$, we obtain

$$\mathbf{X}^T\mathbf{X} = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T$$

Matrix $\mathbf{X}^T\mathbf{X}$ is a symmetric matrix; the decomposition is recognized as the eigenvalue decomposition of the symmetric matrix $\mathbf{X}^T\mathbf{X}$, where the eigenvalues are given by the entries of $\mathbf{\Sigma}^2$, and the eigenvectors by the columns of $\mathbf{V}$.

Similarly, $\mathbf{X}\mathbf{X}^T$ has eigenvalue decomposition

$$\mathbf{X}\mathbf{X}^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T$$

The point of the SVD is that it gives similar information as we obtain from an eigenvalue decomposition, but (1) it is applicable to any matrix (e.g., non-square matrices), and (2) it always exists, whereas the eigenvalue decomposition only exists for "regular" matrices. Also, there are numerically very robust algorithms to compute the decomposition.

### Rank reduction using the SVD

Equation (B.2) shows that, when we invert a matrix $\mathbf{X}$, the smallest singular values of $\mathbf{X}$ become the largest singular values of $\mathbf{X}^\dagger$. Sometimes, if a matrix is almost rank deficient ($\sigma_n$ is very small), that small component will dominate the inverse, which can give rise to numerical problems, as we will see later. In that case, we propose to first approximate $\mathbf{X}$ to a lower rank $d$, by setting all singular values below a certain threshold $\varepsilon$ equal to zero:

$$\hat{\mathbf{X}} = \mathbf{u}_1\sigma_1\mathbf{v}_1^T + \mathbf{u}_2\sigma_2\mathbf{v}_2^T + \cdots + \mathbf{u}_d\sigma_d\mathbf{v}_d^T$$

which we can write as

$$\hat{\mathbf{X}} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\hat{\mathbf{V}}^T, \qquad \hat{\mathbf{U}} = [\mathbf{u}_1 \cdots \mathbf{u}_d], \quad \hat{\mathbf{\Sigma}} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_d \end{bmatrix}, \quad \hat{\mathbf{V}} = [\mathbf{v}_1 \cdots \mathbf{v}_d].$$

This is called the Truncated SVD.

The corresponding approximate inverse is

$$\hat{\mathbf{X}}^\dagger = \hat{\mathbf{V}}\hat{\mathbf{\Sigma}}^{-1}\hat{\mathbf{U}}^T.$$

This is called the Moore-Penrose pseudo-inverse of $\hat{\mathbf{X}}$. It satisfies the projection properties:

$$\hat{\mathbf{X}}\hat{\mathbf{X}}^\dagger = \hat{\mathbf{U}}\hat{\mathbf{U}}^T = \mathbf{P}_c, \qquad \hat{\mathbf{X}}^\dagger\hat{\mathbf{X}} = \hat{\mathbf{V}}\hat{\mathbf{V}}^T = \mathbf{P}_r$$

where $\mathbf{P}_c$ is a projection onto the dominant column span of $\mathbf{X}$, and $\mathbf{P}_r$ a projection onto the dominant row span. The largest singular value of the pseudo-inverse is $\sigma_d^{-1}$.

This pseudo-inverse is commonly used if we are not sure if a matrix is full rank. Typically, we compare the singular values of $\mathbf{X}$ to a threshold ($\varepsilon$) and replace them by 0 if they are below the threshold, leading to $\hat{\mathbf{X}}$. Next, we compute $\hat{\mathbf{X}}^\dagger$ by inverting the non-zero singular values.

### Matrix norms

The norm of a vector $\mathbf{x}$ is

$$\|\mathbf{x}\| = \sqrt{\sum_i |x_i|^2} = \sqrt{\mathbf{x}^T\mathbf{x}}.$$

This is the vector 2-norm. For a matrix $\mathbf{X}$, we can define several norms. The "Frobenius norm" is based on the sum-square of all entries:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i,j} |x_{ij}|^2}.$$

However, we will sometimes also need to use other norms. The "induced 2 norm" or matrix 2-norm $\|\mathbf{X}\|_2$, but usually written simply as $\|\mathbf{X}\|$, measures how much a matrix can increase the 2-norm of a vector $\mathbf{v}$:

$$\|\mathbf{X}\| = \max_{\mathbf{v}} \frac{\|\mathbf{X}\mathbf{v}\|}{\|\mathbf{v}\|} \tag{B.3}$$

Without loss of generality, we may normalize the vectors $\mathbf{v}$ such that $\|\mathbf{v}\| = 1$. We can also insert the SVD. We then obtain

$$\|\mathbf{X}\|^2 = \max_{\|\mathbf{v}\|=1} \|\mathbf{X}\mathbf{v}\|^2 = \max_{\|\mathbf{v}\|=1} \mathbf{v}^T(\mathbf{X}^T\mathbf{X})\mathbf{v} = \max_{\|\mathbf{v}\|=1} \mathbf{v}^T(\mathbf{V}\boldsymbol{\Sigma}^2\mathbf{V}^T)\mathbf{v}.$$

From this we can deduce that the vector $\mathbf{v}$ that maximizes the norm is given by $\mathbf{v} = \mathbf{v}_1$, the dominant right singular vector. The matrix 2-norm of $\mathbf{X}$ is then seen to be equal to $\sigma_1$.

Similarly, the matrix 2-norm of $\mathbf{X}^\dagger$ is given by $\sigma_n^{-1}$.

An important property that follows from the definition of the norm (B.3) is

$$\|\mathbf{X}\mathbf{v}\| \leq \|\mathbf{X}\|\|\mathbf{v}\| \qquad \forall \mathbf{v}$$

where the maximum is only achieved for $\mathbf{v} = \alpha\mathbf{v}_1$.

### The condition number

The condition number of $\mathbf{X}$ is defined by

$$c(\mathbf{X}) := \frac{\sigma_1}{\sigma_n}$$

Thus, we always have $c(\mathbf{X}) \geq 1$. If it is large, then $\mathbf{X}$ is hard to invert (and $\mathbf{X}^\dagger$ is sensitive to small changes). The smallest condition number for a matrix is $c = 1$, which is achieved for an orthogonal matrix.

### Interpretation of the condition number

When we compute the inverse of a matrix, its condition number is very important. Indeed, the condition number gives the relative sensitivity of the solution of a linear systems of equations. Let us suppose that we wish to solve a system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, where we take $\mathbf{A} : n \times n$ square. We have

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Now, if we perturb the data vector $\mathbf{b}$ by a noise vector $\mathbf{e}$, we obtain

$$\mathbf{b}' = \mathbf{b} + \mathbf{e} \quad \Rightarrow \quad \mathbf{x}' = \mathbf{x} + \mathbf{A}^{-1}\mathbf{e}$$

Define $\sigma_1 = \| \mathbf{A} \|, \quad \sigma_n^{-1} = \| \mathbf{A}^{-1} \|$, and use $\|\mathbf{Ax}\| \leq \|\mathbf{A}\|\|\mathbf{x}\|$. Then

$$\begin{array}{rcl} \| \mathbf{A}^{-1}\mathbf{e} \| & \leq & \sigma_n^{-1} \| \mathbf{e} \| \\ \| \mathbf{b} \| & \leq & \sigma_1 \| \mathbf{x} \| \\ \dfrac{\|\mathbf{x}' - \mathbf{x}\|}{\|\mathbf{x}\|} & \leq & \sigma_n^{-1} \dfrac{\| \mathbf{e} \|}{\|\mathbf{x}\|} \leq \sigma_n^{-1}\sigma_1 \dfrac{\| \mathbf{e} \|}{\|\mathbf{b}\|} \end{array}$$

This measures the relative change in the solution vector $\mathbf{x}$, and shows that any error in $\mathbf{b}$ is potentially magnified by a factor equal to the condition number.

If a matrix has a poor condition number, the usual strategy is not to invert it directly, but to do a rank reduction to rank $d$, and compute the pseudo-inverse as shown before.