University of Glasgow | School of Engineering

Digital Signal Processing

V.44

Bernd Porr

Contents

1 Acknowledgements

A big thanks to Fiona Baxter for typing up the handwritten lecture notes and turning them into LaTeX.

2 Introduction

This handout introduces you into the basic concepts of Digital Signal Processing. In contrast to the YouTube clips it focusses more on the theoretical aspects of it and its analytical derivations. The (video-) lectures cover more practical programming examples in Python and C++.

2.1 Suggested reading

- Digital Signal Processing by John G. Proakis and Dimitris K Manolakis
- Digital Signal Processing: System Analysis and Design by Paulo S. R. Diniz, Eduardo A. B. da Silva, and Sergio L. Netto

2.2 Advantages of digital signal processing

- flexible: reconfigurable in software!
- easy storage: numbers!
- cheaper than analogue design
- very reproducible results (virtually no drift)

2.3 Development boards

Buy a DSP development board and play with it. You get them either directly from a company or from a distributor such as Farnell or RS. Also general purpose boards such as the Raspberry PI and Arduino are great platforms for DSP.

3 Python

Python is a fully featured scripting language and it's very well thought through in its approach. It has a vast library of modules which means that only rarely you need implement low level functionality. It can be programmed

both in a quick and dirty approach to try out new ideas and at the same time it supports object oriented programming so that truly professional programs can be developed.

Python has a straightforward interface to call C and C++ functions so that you can also mix it with C/C++ to speed up processing. For example a digital filter class might have the design parts written in python and the actual fast filter routines implemented in C. All this is supported by the python package management so that you can also deploy mixed python / C code without any problems. For example the AI library by google "tensorflow" can be installed as a python package and in the background it uses the GPUs on your computer without even noticing.

Here, I just comment on certain aspects of Python which are important for DSP. Check out the full documentation of python if you are interested beyond DSP in it. In particular it has also very easy to use support for GUI programming.

3.1 How to use Python?

There are two ways to use Python. The one is interactive within the python console and then the other method is by running a python script.

Development environments such as anaconda have both the console and then scripting facility integrated under one roof.

3.1.1 The python console

The python console you can just start by typing "python" or "python3" (for version 3):

```
bp1@bp1-ThinkPad-X220:~$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

then you can type in commands one by one. Under Windows you might need to point the PATH variable to your python program or you let Anaconda do it for you.

3.1.2 Python scripts

Scripts are text files which contain python commands. One command per line and these are executed line by line.

Write your commands in any text editor and save it as a .py-file. However, it's recommended to use an editor which supports at least python's indentation which is a very important feature of python. We'll see that indentation in python defines the body of a function, the content of a loop or the code executed conditionally.

So, it's recommended for any serious work to install an editor which supports python such as Anaconda or emacs.

Once you have your script file saved just type "python myscript.py" and it will be executed.

3.2 Help

There is an abundance of online help available for Python. The official Python pages have an excellent tutorial (`https://docs.python.org/3/tutorial/`) and of course you find answers to all your questions on Stack Overflow.

If you need help for a function you can use the in built help function "help(thecommand)" to get help for "thecommand". Try out "help(print)" to get help for the "print" command.

3.3 Modules

The power of Python comes form thousands of modules which are shipped with python or can be installed via python's (or Anaconda's) package manager. The standard package manager of python is called "pip" which downloads the package for you and installs it. This is not just very convenient but also is safe because the download location is the official python repository.

In general you import modules with the "import" command. For example "import numpy as np" imports the module numpy into python and abbreviates it as "np" so that you'd write "np.sin(c)" to call the sine function in numpy.

The import command loads essentially a file called "numpy.py" so that you can write your own modules just by creating file with that name. See below in the "class" section for an example. There are numerous ways how to import modules but the "import as" way is by far the most popular.

Important modules are:

- numpy: Numerical operations for number crunching, matrix/array operations, trigonometric functions, etc. It offers fast numerical operations on arrays and matrices. It also has extensive Fourier Transform commands.

- pylab: Plotting functions (MATLAB style), statistics and data fitting functions
- scipy: Scientific computing package. In particular we are insterested in scipy.signal which contains all the signal processing functions and filter design commands. It also offers Fourier Transforms, data io routines, vector quantisation, linear algebra, optimisation tools and statistics.

3.4 Data types

Python has all the standard data types such as int, float, string, ... and a few more. You can check which type a variable has with: "type(variable)". Important for DSP are these types:

3.4.1 Numbers

Numbers are represented in a standard way, including complex numbers:

- `6E23`= $6 \cdot 10^{23}$
- `6+2j` is a complex number which is just generated by adding a "j" to the imaginary part.
- `complex(0,1)` creates 1j and is useful if you have variables which you want to combine into a complex number.

Note that python has implicit types which might make you trip over. If you write "a=1" then "a" is an integer whereas "a=1.0" is a floating point number. You can force a certain type by using the typecast operator, for example " a = int(b)" to create an integer.

3.4.2 Lists

There are numerous data types in Python which allow storing collections of data. For DSP we need: lists, tupels and arrays. To clear up the confusion from the start we are going through them one by one. First up: lists!

- `p = [1,9,77]` is a list with the components 1,2 and 77.
- `p[2]` returns the third element of the list (the index counts from 0). Note the index needs to be an integer so you might need to cast a floating point variables first into an integer, for example "p[int(a)]".

- `p[1:5]` extracts from the list all elements from index number 1 to index number 4 (!). The index 5 is excluded! This is called splicing. Check out the online docs for more examples.
- `p = range(1,4)` generates $1, 2, 3$ which is the same as a "for loop" in C: for(int i=1;i¡4;i++)
- `p.append(10)` adds a new element to the end of the list. You see that lists are essentially classes with methods (see below)! Check out the documentation for more options to manipulate lists.
- `len(p2)` provides the number of elements.

3.4.3 Arrays

This data structure is used for number crunching and is thus ideal for DSP. Python arrays are essentially C arrays with a thin wrapper around them.

Usually you'd create arrays with the numpy library (add `import numpy as np` to your script).

- `a = np.array([1, 2, 5])` which creates a numpy array from a standard Python list containing 1,2,5.
- `b = np.array([[6, 2], [9, 7]])` creates a two dimensional array.
- There are a lot of convenience functions in numpy to create arrays just containing zeros, ones or ascending numbers etc. Check out the numpy docs.

Since arrays are close relatives to C arrays they have an additional field which is called "dtype" which determines the type of the array elements, for example "int16" (a 16 bit wide integer).

```
>>> data
array([-12288, -12545, -12798, ...,    511,    513,    513], dtype=int16)
>>> data.dtype
dtype('int16')
>>> data.astype(float)
array([-12288., -12545., -12798., ...,    511.,    513.,    513.])
```

where the member function "astype" can convert from one array type to another, here from int16 to floating point.

3.4.4 Tupels

Tupels are similar to lists but they are rather used as "containers" to package up different variables into one variable. For example, you might want to put the sampling rate and the resolution of an ADC into a tuple as they belong together.

- `p = (1,9,'kk')` is a tupel with the two integer numbers 1,2 and 'kk'.
- The round brackets are often omitted: `p = 1,9,'kk'` which implicitly defines a tupel.
- `p[1]` returns the 2nd element of the tupel (the index counts from 0).
- Extracting a tupel into separate variables also works: `a,b,c = p` assigns 1 to a, 9 to b and 'kk' to c. This is very convenient for functions if you want to return more than one result. Just package them up into a tuple and return it.

3.5 Conditionals

Conditional operation is done in exactly the same way as in other languages:

```
if a > c:
    a = c
    c = c*2
print(c)
```

The indent of 4 characters indicates the conditionally executed commands. The same applies for do and while loops where the body has an indent. Here, print(c) is always executed because it doesn't have an indent.

3.6 Loops

Loops in python are always done with the help of iterators. This means that we progress through the elements of a list. Imagine you want to calculate the average of the array `y`.

```
avg = 0
for i in range(len(y)):
    avg = avg + y[i]
    print("momentary average=",avg)
print(avg/len(y))
```

The "for" statement iterates through all elements of an array. In this case we have created a list with the "range" command. Important: the *indent* indicates the commands which the loop iterates through. This is the general rule in python. There are also while/do loops available.

3.7 Functions

Function definitions can happen anywhere in the program or in a separate file. For example, the above average calculations can be defined as:

```
def myaverage(g):
    avg = 0
    for i in range(len(y)):
        avg = avg + y[i]
    return (avg/len(y))
```

and then called from the main program as `myaverage(y)`. The keyword "def" indicates the start of the function definition. Note again the 4 character indent for the function body and the colon after the function name. This is characteristic for python.

3.8 Classes

Classes contain functions and variables. They usually have a constructor which is a special function which is called when an instance of a class is created.

```
class MyAmazingClass:
    def __init__(self,coeff):
        # Your init code here.
        self.mylocalcoeff = coeff

    def doSomeProcessing(self,data):
        # we process data
        data = data * self.mylocalcoeff
        return data
```

(!) Note the special variable "self" which is *the first* argument of every function in a class. Variables which are part of the class are references by "self". For example the constructor argument "coeff" is saved in the class as "mylocalcoeff" and is then available throughout the lifetime of the class as "self.mylocalcoeff". When we call the method "doSomeProcessing" then

"self.mylocalcoeff" is multiplied with the "data" argument and the result is returned. Anything which hasn't got "self." in front of it will be forgotten after the function has terminated. In this way you distinguish which variables are kept for the lifetime of the class or are just used temporarily.

It's common practise to save a class (or a couple of them) in a separate file and create a module. For example, let's save our averaging class in "myamazingclass.py". If we want to use this class in the main program we type:

```
import myamazingclass
f = myamazingclass.MyAmazingClass(5)
a = f.doSomeProcessing(10)
b = f.doSomeProcessing(20)
```

which will give us 50 as a result for the variable "a" and 100 for the variable "b". The variable "f" is the instance of the class MyAmazingClass and the argument initialises self.mylocalcoeff with 5.

3.9 Mathematical functions

Just import the numpy library which contains all the mathematical functions and constants such as pi. `import numpy as np`. It has all the standard functions such as np.sin, np.cos, np.tan, np.exp, ...

Note that these functions also take arrays as arguments. For example, inputting the array x into the sine:

```
x = np.array(range(6))
y = np.sin(x);
```

yields again an array in y! All elements have been processed at the *same* time.

3.10 Plotting of Functions

Pylab has a vast amount of plotting functions which supports simple plotting with just two commands (plot and show) up to complex animations of 3D data. Plotting a sine wave can be achieved with:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0,2*np.pi,100)
y = np.sin(x)
plt.plot(x,y)
plt.show()
```

where I have used the numpy linspace command to create an array which runs from 0 to two pi with 100 samples.

If you want to create more plot-windows you can use the command `plt.figure`. Access the windows with the commands `plt.figure(1)`, `plt.figure(2)`.

It is also possible to combine plot windows into one with the command `subplot`.

How to make it look nice? Here is an example:

```
plt.xlabel('time (s)')
plt.ylabel('voltage (V)')
plt.title('That is an AC voltage over time!')
plt.grid(True)
```

Generally the best approach is to use an example and then customise it to your needs.

3.11 Importing data

3.11.1 Space/Tab/Comma separated data

Numpy has a very handy import function for data as long as the the elements in the data file are separated with spaces, tabs or commas. To load such data files just type in: `x = np.loadtxt('mydatafile.dat')`. This creates a two dimensional array which reflects exactly the structure in the datafile.

Thus, if you want to write software which creates Python readable data then just export the data as space/tab separated ASCII files.

3.11.2 WAV files

Scipy contains in its submodule io a subsubmodule wavfile which supports reading and writing wav files:

```
import scipy.io.wavfile as wavfile
fs,data = wavfile.read('kongas.wav');
```

fs,data is a tuple containing the sampling rate and then the data.

```
import scipy.io.wavfile as wavfile
wavfile.write('kongas_filt.wav',fs,y3);
```

whereas 'fs' is the sampling rate and y3 the data – usually normalised between -1 and +1 for floating point numbers.

4 Signal conversion

$$\text{Analogue} \rightarrow \text{A/D} \rightarrow \text{Digital processing} \rightarrow \text{D/A} \rightarrow \text{Analogue} \tag{1}$$

4.1 A/D conversion

1. **sampling**

$$\underbrace{X_a(nT)}_{\text{analogue signal}} \quad \equiv \quad \underbrace{x(n)}_{\text{discrete data}} \tag{2}$$

   - X_a: analogue signal
   - T: sampling interval, $\frac{1}{T} = F_s$ is the sampling rate
   - $x(n)$: discrete data

2. **quantisation:** Continuous value into discrete steps. $X(n) \rightarrow X_q(n)$
3. **Coding** $X_q(n) \rightarrow$ into binary sequence or integer numbers.

4.2 D/A Conversion

Back to analogue. The D/A converter needs to interpolate between the samples to get the original signal back without error! This is usually done by a simple low pass filter.

5 Sampling of Analogue Signals

The analogue signal X_a is sampled at intervals T to get the sampled signal $X(n)$.

$$X(n) = X_a(nT), \qquad -\infty < n < +\infty \tag{3}$$

5.1 Normalised frequency

Note that the sampling frequency

$$F_s = \frac{1}{T_s} \tag{4}$$

is lost and needs to be stored additionally so that the analogue signal can be reconstructed:

$$X_a(t) = X_a(nT) = X_a(\frac{n}{F_s}) \tag{5}$$

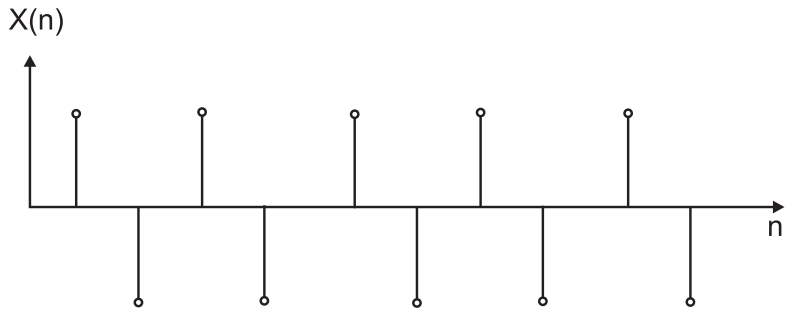

Figure 1: Maximum frequency of a sampled signal is half the normalised frequency.

What is the frequency range of our digital signal? What is the maximum frequency in digital terms? Fig 1 shows that the max frequency is 0.5 because we need two samples to represent a "wave". The frequency range $0 \ldots 0.5$ is called the *normalised frequency* range. What is the relation between normalised frequency and sampling rate?

Let's have a look at the analog signal $X_a(t)$ with the frequency F:

$$X_a(t) = A\cos(2\pi F t) \tag{6}$$

and we sample it only at:

$$t = n/F_s \tag{7}$$

Then the digital signal is:

$$X_a(n) = A\cos(2\pi F/F_s n) \tag{8}$$

Now, we can define the normalised frequency as:

$$\text{Normalised frequency:} \qquad f = \frac{F}{F_s} \tag{9}$$

which has its max value at 0.5 which represents one period of a sine wave within two samples[1]

5.2 Nyquist frequency

Recall that the max value for the normalised frequency f is 0.5 and that:

$$X_a(n) = A\cos(2\pi n f) \tag{10}$$

[1]Note that the normalised frequency in Python's scipy is differently defined. In scipy it is annoyingly defined as $f_{\text{scipy}} = 2\frac{F}{F_s}$. So, in other words $f_{\text{scipy}} = 1$ is the Nyquist frequency instead of $f = 0.5$ as normally defined.

with n as an integer because we are sampling.

What happens above $f > 0.5$? Imagine $f = 1$

$$X_a(n) = \cos(2\pi n) = 1 \qquad f = 1 \tag{11}$$

which gives us DC or zero frequency. The same is true for $f = 2, 3, 4, \ldots$. We see that above $f = 0.5$ we never get higher frequencies. Instead they will always stay between $0 \ldots 0.5$ for the simple reason that it is not possible to represent higher frequencies. This will be discussed later in greater detail.

The ratio F/F_s must be lower than 0.5 to avoid ambiguity or in other words the maximum frequency in a signal must be lower than $\frac{1}{2}F_s$. This is the Nyquist frequency.

If there are higher frequencies in the signal then these frequencies are "folded down" into the frequency range of $0 \ldots \frac{1}{2}F_s$ and creating an alias of its original frequency in the so called "baseband" ($f = 0 \ldots 0.5$). As long as the alias is not overlapping with other signal components in the baseband this can be used to downmix a signal. This leads to the general definition of the sampling theorem which states that the bandwidth B of the input signal must be half of the sampling rate F_s:

$$B < \frac{1}{2}F_s \tag{12}$$

The frequency $\frac{1}{2}F_s$ is called the Nyquist frequency.

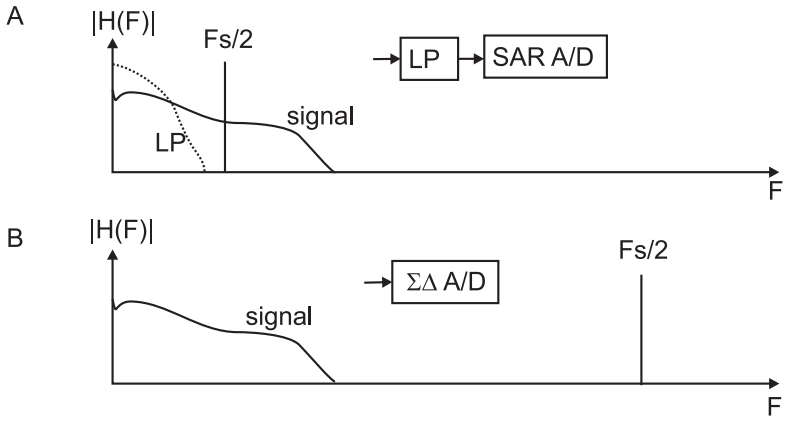

Figure 2: Anti alias filtering. A) with a lowpass filter. B) with a sigma delta converter with very high sampling rate.

What do we do if the signal contains frequencies above $\frac{1}{2}F_s$? There are two ways to tackle this problem: The classical way is to use a lowpass filter

(see Fig. 2A) which filters out all frequencies above the Nyquist frequency. However this might be difficult in applications with high resolution A/D converters. Alternatively one can use a much higher sampling rate to avoid aliasing. This is the idea of the sigma delta converter which operates at sampling rates hundred times higher than the Nyquist frequency.

5.3 Reconstruction of an analogue signal: Sampling theorem

Is it possible to reconstruct an analogue signal from a digital signal which contains only frequencies below the Nyquist frequency?

$$F_s > 2F_{\max} \tag{13}$$

where $F_{\max}$ is max frequency in the signal which we represent by sine waves:

$$x_a(t) = \sum_{i=1}^{n} A_i \cos(2\pi F_i t + \Theta_i) \tag{14}$$

The analogue signal $x_a(t)$ can be completely reconstructed if:

$$g(t) = \frac{\sin 2\pi Bt}{2\pi Bt} \tag{15}$$

with

$$B = F_{\max} \tag{16}$$

$$x_a(t) = \sum_{h=-a}^{a} x_a(\frac{n}{F_s}) g(t - \frac{h}{F_s}) \tag{17}$$

6 Quantisation of analogue signals

A/D converters have a certain resolution. For example, the MAX1271 has a resolution of 12 bits which means that it divides the input range into 4096 equal steps (see Fig 3).

$$\Delta = \text{quantisation step} = \frac{x_{\max} - x_{\min}}{L - 1} \tag{18}$$

where $x_{\max} - x_{\min}$ is the dynamic range in volt (for example $4.096V$) and L is the number of quantisation steps (for example, 4096). Δ is the quantisation step which defines minimal voltage change which is needed to see a change in the output of the quantiser. The operation of the quantiser can be written down as:

$$x_q(n) = Q[x(n)] \tag{19}$$

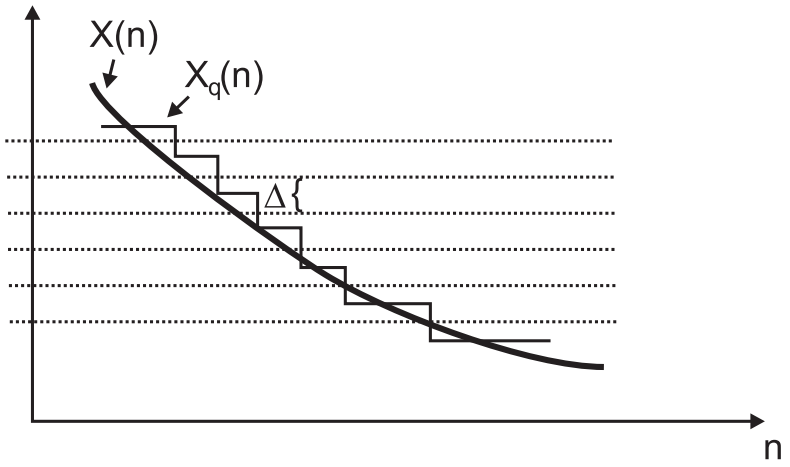

Figure 3: Δ is the quantisation step.

6.1 Quantisation error

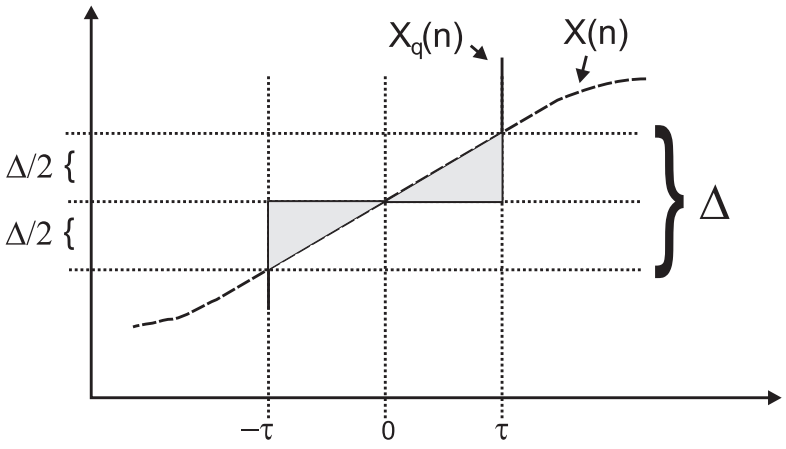

Figure 4: Illustration of the quantisation error. It is zero at $t = 0$ and increases to the edges of the sampling interval. Illustrated is the worst case scenario. This repeats in the next sampling interval and so forth.

Fig. 4 shows error produced by the quantiser in the worst case scenario. From that it can be seen that the maximum quantisation error is half the quantisation step:

$$-\frac{\Delta}{2} \leq e(n) \leq \frac{\Delta}{2} \tag{20}$$

The smaller the quantisation step Δ the lower the error!

What is the mean square error P_q?

$$P_q = \frac{1}{\tau}\int_0^\tau e_q^2(t)dt \tag{21}$$

$$\begin{aligned} P_q &= \frac{1}{\tau}\int_0^\tau \left(\frac{\Delta}{2\tau}\right)^2 t^2 dt & (22)\\ &= \frac{\Delta^2}{4\tau^3}\int_0^\tau t^2 dt & (23)\end{aligned}$$

$$P_q = \frac{\Delta^2}{12\tau^3}\tau^3 = \frac{\Delta^2}{12} \tag{24}$$

What is the relative error to a sine wave?

$$P_x = \frac{1}{T_p}\int_0^{T_p} (A\cos\Omega t)^2 dt = \frac{A^2}{2} \tag{25}$$

Ratio to signal power to noise:

$$\begin{aligned} \text{SQNR} &= \frac{P_x}{P_q} = \frac{A^2}{2}.\frac{12}{\Delta^2} & (26)\\ &= \frac{6A^2}{\Delta^2} & (27)\end{aligned}$$

This equation needs to be interpreted with care because increasing the amplitude of the input signal might lead to saturation if the input range of the A/D converter is exceeded.

7 Frequency representation of signals

Often we are more interested in the frequency representation of signals than the evolution in time.

We will use small letters for time domain representation and capital letters for the frequency representation, for example $X(F)$ and $x(t)$.

7.1 Continuous time and frequency

7.1.1 Periodic signals

Periodic signals can be composed of sine waves :

$$x(t) = \sum_{k=-\infty}^{\infty} c_k e^{j2\pi k F_1 t} \tag{28}$$

where F_1 is the principle or fundamental frequency and $k \neq 1$ are the harmonics with strength c_k. Usually c_1 is set to 1. This is a Fourier series with c_k as coefficients. For example an ECG has a fundamental frequency of about 1Hz (60 beats per minute). However, the harmonics give the ECG its characteristic peak like shape.

How do we find the coefficients c_k?

$$c_k = \frac{1}{T_p} \int_{T_p} x(t) e^{-j2\pi k F_1 t} dt \tag{29}$$

For simple cases there are analytical solutions for c_k, for example for square waves, triangle wave, etc.

What are the properties of c_k?

$$c_k = c_{-k}^* \qquad \Leftrightarrow \qquad x(t) \text{ is real} \tag{30}$$

or

$$c_k = \mid c_k \mid e^{j\theta_k} \tag{31}$$

$$c_{-k} = \mid c_k \mid e^{-j\theta_k} \tag{32}$$

Proof: with the handy equation...

$$\cos z = \frac{1}{2}\left(e^{zi} + e^{-zi}\right) \tag{33}$$

we get

$$x(t) = c_0 + \sum_{k=1}^{\infty} \mid c_k \mid e^{j\theta_k} e^{j2\pi k F_1 t} + \sum_{k=1}^{\infty} \mid c_k \mid e^{-j\theta_k} e^{-j2\pi k F_1 t} \tag{34}$$

$$= c_0 + 2\sum_{k=1}^{\infty} \mid c_k \mid \cos(2\pi k F_1 t + \theta_k) \tag{35}$$

How are the frequencies distributed? Let's have a look at the frequency spectrum of a periodic signal: $P_k = \mid c_k \mid^2$

- There are discrete frequency peaks
- Spacing of the peaks is $\frac{1}{T_1} = F_1$
- Only the positive frequencies are needed: $c_{-k} = c_k^*$

7.1.2 A-periodic signals

In case nothing is known about $X(t)$ we need to integrate over all frequencies instead of just the discrete frequencies.

$$X(F) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi Ft}dt \tag{36}$$

Consequently, the frequency spectrum $X(F)$ is continuous.

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(F)e^{j2\pi Ft}dF \tag{37}$$

7.2 Sampled time and/or frequency

7.2.1 Discrete time Fourier Transform (DTFT)

The signal $x(n)$ is discrete whereas the resulting frequency spectrum is considered as continuous (arbitrary signals).

- Analysis or direct transform:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n} \tag{38}$$

- Synthesis or inverse transform:

$$\begin{aligned} x(n) &= \frac{1}{2\pi}\int_{-\pi}^{\pi} X(\omega)e^{j\omega n}d\omega & (39)\\ &= \frac{1}{2\pi}\int_{-0.5}^{0.5} X(f)e^{j2\pi fn}dt & (40)\end{aligned}$$

  note the range here. f is the normalised frequency.

7.2.2 The effect of time domain sampling on the spectrum (Sampling theorem)

What effect has time domain sampling on the frequency spectrum[2]? Imagine we have an analog spectrum $X_a(F)$ from a continuous signal $x(t)$. We want to know how the spectrum $X(F)$ of the discrete signal $x(n)$ looks like.

$$X(F) \Leftrightarrow X_a(F) \tag{41}$$

[2]This derivation is loosely based on Proakis and Manolakis (1996)

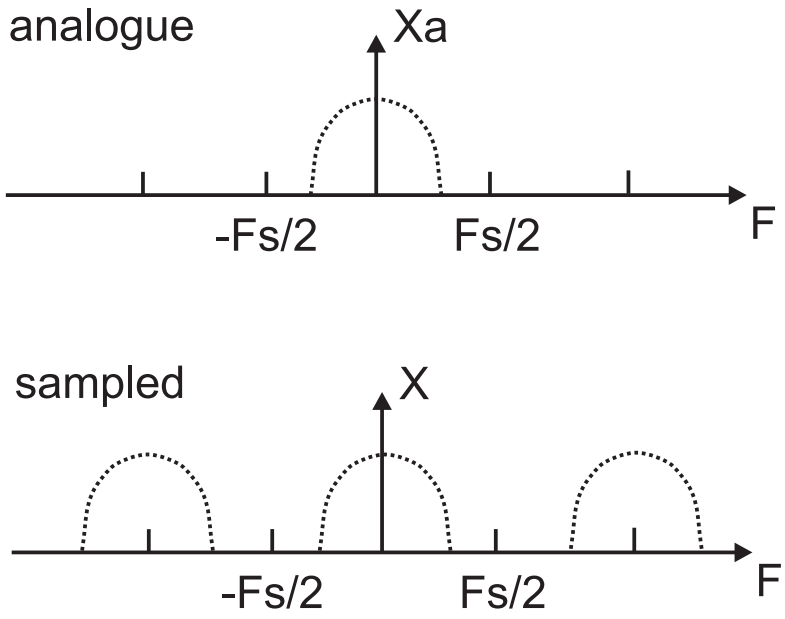

Figure 5: Effect of sampling on the spectrum.

The signal is represented by $x(n)$ so that we can equate the Fourier transforms of the sampled spectrum $X(F)$ and of the analogue spectrum $X_a(F)$.

$$\int_{-0.5}^{0.5} \underbrace{X(f)}_{\text{sampled}} e^{j2\pi fn} df = \int_{-\infty}^{+\infty} \underbrace{X_a(F)}_{\text{cont}} e^{j2\pi nF/F_s} dF \tag{42}$$

Obviously $X(f)$ must be different to accommodate the different integration ranges. The trick is now to divide the integral on the right hand side of Eq. 42 into chunks to make it compatible to the range on the left hand hand side.

Remember that the normalised frequency is $f = F/F_s$ which allows us to change the integration to analogue frequency on both sides:

$$\frac{1}{F_s} \int_{-F_s/2}^{F_s/2} X(\frac{F}{F_s}) e^{j2\pi nF/F_s} dF = \int_{-\infty}^{+\infty} X_a(F) e^{j2\pi nF/F_s} dF \tag{43}$$

and now we divide the right hand side into chunks of F_s which corresponds to the integration range on the left hand side.

$$\int_{-\infty}^{+\infty} X_a(F) e^{j2\pi n \frac{F}{F_s}} dF \quad = \quad \sum_{k=-\infty}^{\infty} \int_{-\frac{1}{2}F_s+kF_s}^{+\frac{1}{2}F_s+kF_s} X_a(F) e^{j2\pi n \frac{F}{F_s}} dF \tag{44}$$

$$= \sum_{k=-\infty}^{\infty} \int_{-\frac{1}{2}F_s}^{+\frac{1}{2}F_s} X_a(F - kF_s) \underbrace{e^{j2\pi n \frac{F}{F_s}}}_{kF_s \text{ omit.}} dF \quad (45)$$

$$= \int_{-\frac{1}{2}F_s}^{+\frac{1}{2}F_s} \underbrace{\sum_{k=-\infty}^{\infty} X_a(F - kF_s)}_{=X(F) \text{ of Eq. 43}} e^{j2\pi n \frac{F}{F_s}} dF \quad (46)$$

This gives us now an equation for the sampled spectrum:

$$X(F/F_s) = F_s \sum_{k=-\infty}^{\infty} X_a(F - kF_s) \quad (47)$$

$$X(f) = F_s \sum_{k=-\infty}^{\infty} X_a[(f - k)F_s] \quad (48)$$

This equation can now be interpreted. In order to get the sampled spectrum $X(F)$ we need to make copies of the analog spectrum $X_a(F)$ and place these copies at multiples of the sampling rate F_s (see Fig. 5). This illustrates also the *sampling theorem*: if the bandwidth of the spectrum is wider than $F_s/2$ then the copies of the analogue spectrum will overlap and reconstruction would be impossible. This is called aliasing. Note that it is not necessary bad that the spectrum of the analogue signal lies within the range of the so called "base band" $-F/2 \ldots F/2$. It can also lie in another frequency range further up, for example $-F/2 + 34 \ldots F/2 + 34$ as long as the *bandwidth* does not exceed $F_s/2$. If it is placed further up it will automatically show up in the baseband $-F/2 \ldots F/2$ which is called "fold down". This can be used for our purposes if we want to down mix a signal.

With the insight from these equations we can create a plot of how analogue frequencies map onto sampled frequencies. Fig 6 shows how the analogue frequencies F_{analogue} map on the normalised frequencies f_{sampled}. As long as the analogue frequencies are below $F_s/2$ the mapping is as usual as shown in Fig 6A. Between $F_s/2$ and F_s we have an inverse mapping: an increase in analogue frequency causes a decrease in frequencies. Then, from F_s we have again an increase in frequencies starting from DC. So, in general if we keep a bandlimited signal within one of these slopes (for example from $F_s \ldots F_s + 1/2F_s$ as shown in Fig 6B) then we can reproduce the signal.

This leads us to the generalised Nyquist theorem: if a bandpass filtered signal has a bandwidth of B then the minimum sampling frequency is $F_s = 2B$.

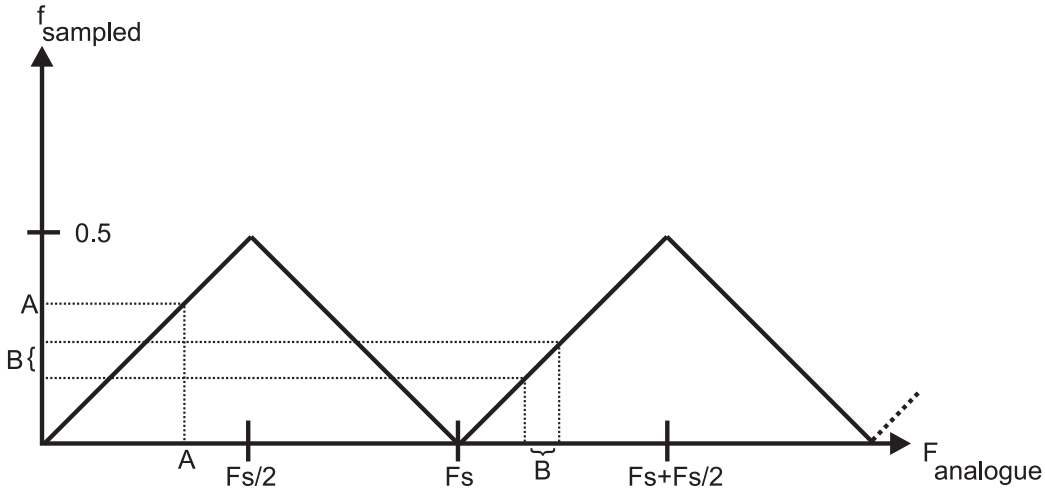

Figure 6: Mapping of frequencies in the analogue domain (F) and sampled domain (f). F_s is the sampling rate.

7.2.3 Discrete Fourier Transform (DFT)

So far we have only sampled in the time domain. However, on a digital computer the Fourier spectrum will always be a discrete spectrum.

The discrete Fourier Transform (DFT) is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} \qquad k = 0, 1, 2, \ldots, N-1 \tag{49}$$

where N is the number of samples in both the time and frequency domain.

The inverse discrete Fourier Transform (IDFT) is defined as:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j2\pi kn/N} \qquad n = 0, 1, 2, \ldots, N-1 \tag{50}$$

What is the effect in the timedomain of this discretisation[3]? We start with the continuous Fourier transform and discretise it into N samples in the frequency domain:

$$X\left(\frac{2\pi}{N}k\right) = \sum_{n=-\infty}^{\infty} x(n) e^{-j\frac{2\pi}{N}kn} \qquad k = 0, \ldots, N-1 \tag{51}$$

Let's subdivide the sum into chunks of length N:

$$X\left(\frac{2\pi}{N}k\right) = \sum_{l=-\infty}^{\infty} \sum_{n=lN}^{lN+N-1} x(n) e^{-j\frac{2\pi}{N}kn} \tag{52}$$

[3]This derivation is loosely based on Proakis and Manolakis (1996)

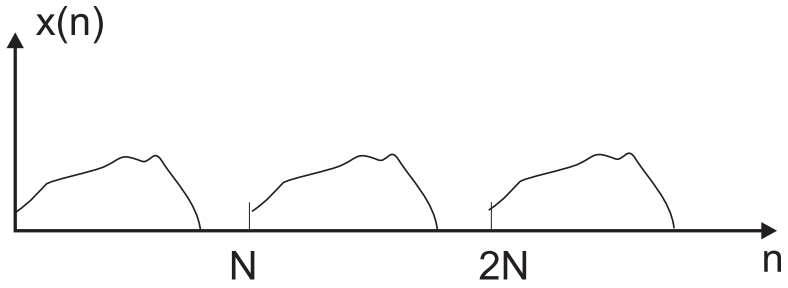

Figure 7: Effect of sampling in the frequency domain. The inverse in the time domain $x(n)$ contains repetitions of the original signal.

$$= \sum_{l=-\infty}^{\infty} \sum_{n=0}^{N-1} x(n-lN) e^{-j\frac{2\pi}{N}kn} \tag{53}$$

$$= \sum_{n=0}^{N-1} \underbrace{\sum_{l=-\infty}^{\infty} x(n-lN)}_{\text{Periodic repetition!}} \quad e^{-j\frac{2\pi}{N}kn} \tag{54}$$

We note the following:

- Ambiguity in the time domain
- The signal is repeated every N samples

Practically this repetition won't show up because the number of samples is limited to N in the inverse transform. However, for operations which shift signals in the frequency domain it is important to remember that we shift a periodic time series. If we shift it out at the end of the array we will get it back at the start of the array.

Fig. 8 shows an example of a DFT. It is important to know that the spectrum is mirrored around $N/2$. DC is represented by $X(0)$ and the Nyquist frequency $F_s/2$ is represented by $X(N/2)$. The mirroring occurs because the input signal $x(n)$ is *real*. This is important if one wants to modify the spectrum $X(F)$ by hand, for example to eliminate 50Hz noise. One needs to zero two elements of $X(F)$ to zero. This is illustrated in this python code:

```
import scipy as sp
yf=sp.fft(y)
# the sampling rate is 1kHz. We've got 2000 samples.
# midpoint at the ifft is 1000 which corresponds to 500Hz
# So, 100 corresponds to 50Hz
```

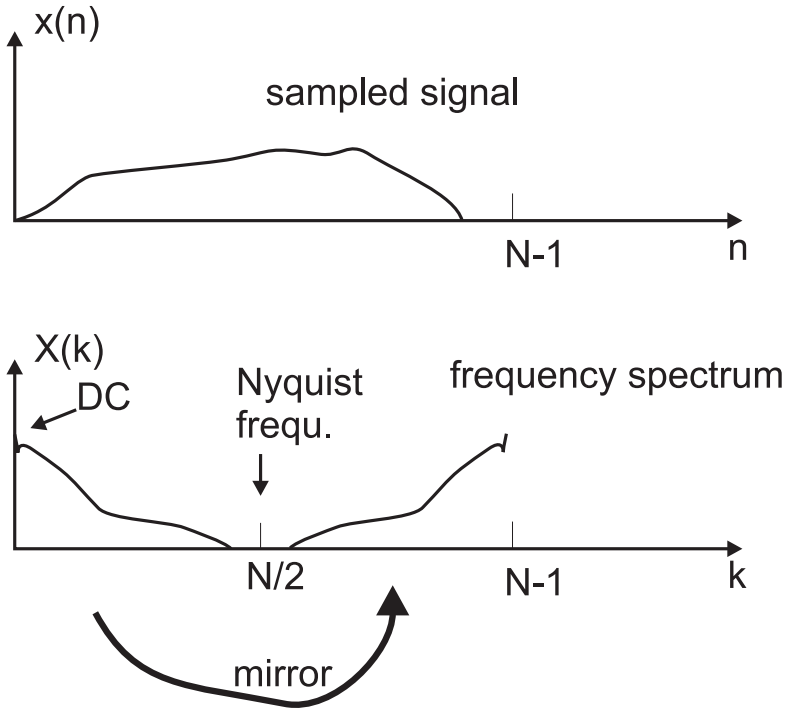

Figure 8: Example of a DFT. The sampled signal $x(n)$ is converted to the spectrum $X(k)$. Both have N samples. Note the redundancy: the spectrum is mirrored around $N/2$. The first value $X(0)$ is the DC value of the signal. However this is the only sample which is not mirrored.

```
yf[99:101+1]=0;
# and the mirror
yf[1899:1901+1]=0;
#
yi=sp.ifft(yf);
```

This filters out the 50Hz hum from the signal y with sampling rate 1000Hz. The signal `yi` should be real valued again or contain only very small complex numbers due to numerical errors.

7.2.4 Properties of the DFT

- Periodicity:

$$x(n+N) = x(n) \tag{55}$$
$$X(k+N) = X(k) \tag{56}$$

- Symmetry: if $x(n)$ real:

$$x(n) \text{ is real} \Leftrightarrow X^*(k) = X(-k) = X(N-k) \tag{57}$$

  This is important when manipulating $X(k)$ by hand.

- Time Reversal:

$$x(n) \quad \leftrightarrow \quad X(k) \tag{58}$$
$$x(-n) \quad \leftrightarrow \quad X(N-k) \tag{59}$$

- *Circular* convolution:

$$X_1(k)X_2(k) \leftrightarrow x_1(n) * x_2(n) \tag{60}$$

  with

$$x_3(m) = \sum_{n=0}^{N-1} x_1(n)x_2(m-n) \tag{61}$$

More useful equations are at Proakis and Manolakis (1996, pp.415).

7.2.5 Problems with finite length DFTs

$$x_w(n) = x(n) \cdot w(n) \tag{62}$$

where x is the input signal and $w(n)$ represents a function which is 1 from $n = 0$ to $n = L - 1$ so that we have L samples from the signal $x(n)$.

To illustrate the effect of this finite sequence we introduce a sine wave $x(n) = \cos \omega_0 n$ which has just two peaks in a proper Fourier spectrum at $-\omega$ and $+\omega$.

The spectrum of the rectangular window with the width L is:

$$W(\omega) = \frac{sin(\omega L/2)}{sin(\omega/2)} e^{-j\omega(L-1)/2} \tag{63}$$

The resulting spectrum is then:

$$X_3(\omega) = X(\omega) * W(\omega) \tag{64}$$

Because the spectrum of $X(\omega)$ consists of just two delta functions the spectrum $X_3(\omega)$ contains the window spectrum $W(\omega)$ twice at $-\omega$ and $+\omega$ (see Fig. 9). This is called *leakage*. Solutions to solve the leakage problem? Use Windows with a narrower spectrum and with less ripples (see FIR filters).

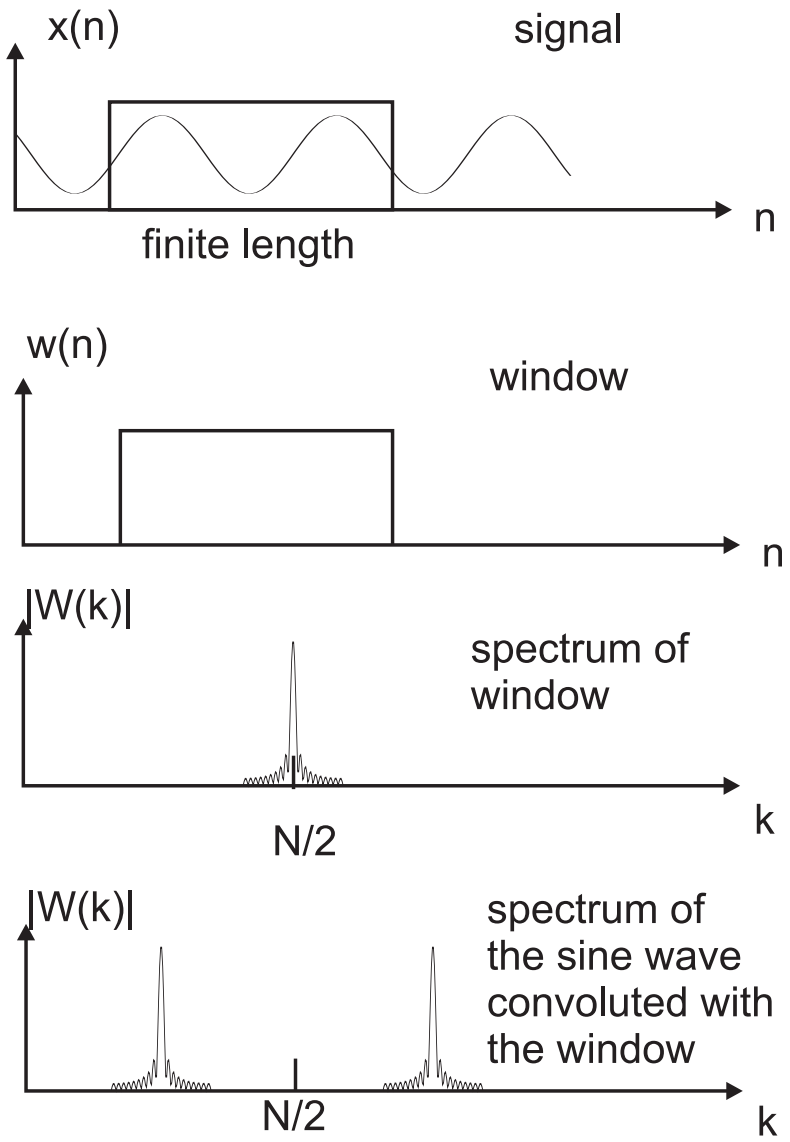

Figure 9: The effect of windowing on the DFT.

7.2.6 Fast Fourier Transform

We can rewrite the DFT (Eq. 49) in a slightly more compact form:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \tag{65}$$

with the constant:

$$W_N = e^{-j2\pi/N} \tag{66}$$

The problem with the DFT is that it needs N^2 multiplications. How can we reduce the number of multiplications? Idea: Let's divide the DFT in an odd and an even sequence:

$$x(2m) \tag{67}$$

$$x(2m+1), \qquad m = 0, ..., \frac{N}{2} - 1 \tag{68}$$

which gives us with the trick $W_N^{2mk} = W_{N/2}^{mk}$ because of the definition Eq. 66.

$$\begin{aligned} X(k) &= \sum_{m=0}^{N/2-1} x(2m) W_N^{2mk} + \sum_{m=0}^{N/2-1} x(2m+1) W_N^{k(2m+1)} && (69)\\ &= \sum_{m=0}^{N/2-1} x(2m) W_{N/2}^{mk} + W_N^k \sum_{m=0}^{N/2-1} x(2m+1) W_{N/2}^{mk} && (70)\\ &= F_e(k) + W_N^k F_o(k) && (71) \end{aligned}$$

F_e and F_o have both half the length of the original sequence and need only $(N/2)^2$ multiplication, so in total $2 \cdot (N/2)^2 = \frac{N^2}{2}$. Basically by dividing the sequence in even and odd parts we can reduce the number of multiplications by 2. Obviously, the next step is then to subdivide the two sequences $F_e(k)$ and $F_o(k)$ even further into something like $F_{ee}(k), F_{eo}(k), F_{oe}(k)$ and $F_{oo}(k)$.

In general the recipe for the calculation of the FFT is:

$$X_i(k) = X_{ie}(k) + W_L^k X_{io}(k) \tag{72}$$

W_L^k is the phase factor in front of the odd sequence. This is continued until we have only two point ($N = 2$) DFTs (see Eq. 65):

$$\text{DC:} \qquad X(0) = x(0) + \underbrace{W_2^0}_{1} x(1) = x(0) + x(1) \tag{73}$$

$$\text{Nyquist frequ.:} \qquad X(1) = x(0) + \underbrace{W_2^1}_{-1} x(1) = x(0) - x(1) \tag{74}$$

A two point DFT operates only on two samples which can represent only two frequencies: DC and the Nyquist frequency which makes the calculation trivial. Eq. 73 is an averager and Eq. 74 is basically a differentiator which gives the max output for the sequence $1, -1, 1, -1, \ldots$. Fig. 10 illustrates how to divide the initial sequence to arrive at 2 point DFTs. In other words the calculation of the full DFT is done by first calculating $N/2$ 2 point DFTs and recombining the results with the help of Eq. 72. This is sometimes called the "Butterfly" algorithm because the data flow diagram can be drawn as a

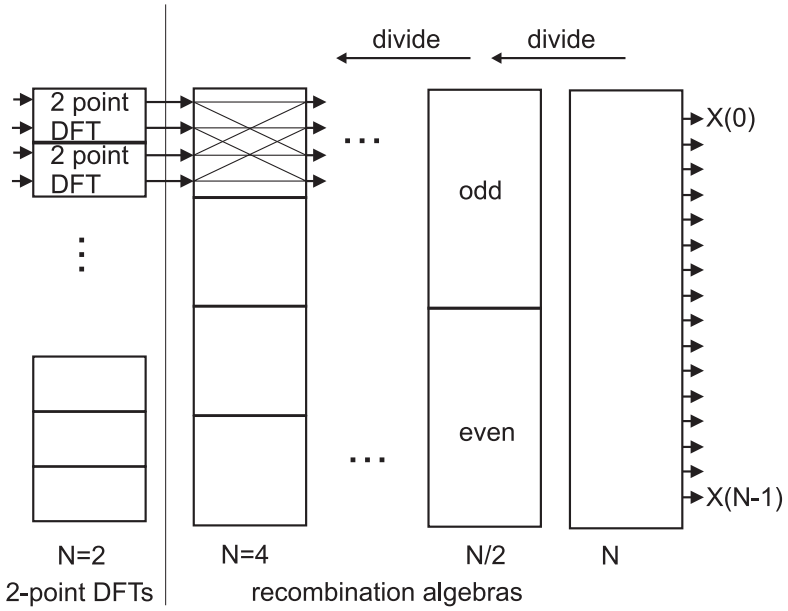

Figure 10: Illustration of the Fast Fourier Algorithm. The sequence of N samples is recursively divided into subsequences of odd and even samples.

butterfly. The number of complex multiplications reduces in this approach to $N \log_2 N$ which is actually the worst case scenario because many W_L^k usually turn out to be $1, -1, j, -j$ which are just sign inversions or swaps of real and imaginary parts. A clever implementation of this algorithm will be even faster.

In summary the idea behind the FFT algorithms is to divide the sequence into subsequences. Here we have presented the most popular radix 2 algorithm. The radix 4 is even more efficient and there are also algorithms for divisions into prime numbers and other rather exotic divisions. However, the main idea is always the same: subsample the data in a clever way so that the final DFT becomes trivial.

7.3 Software

In Teukolsky et al. (2007) you'll find highly optimised C code for Fourier transforms. Most Linux distros (Ubuntu, Suse, RedHat, ...) come with an excellent FFT library called `libfftw3`.

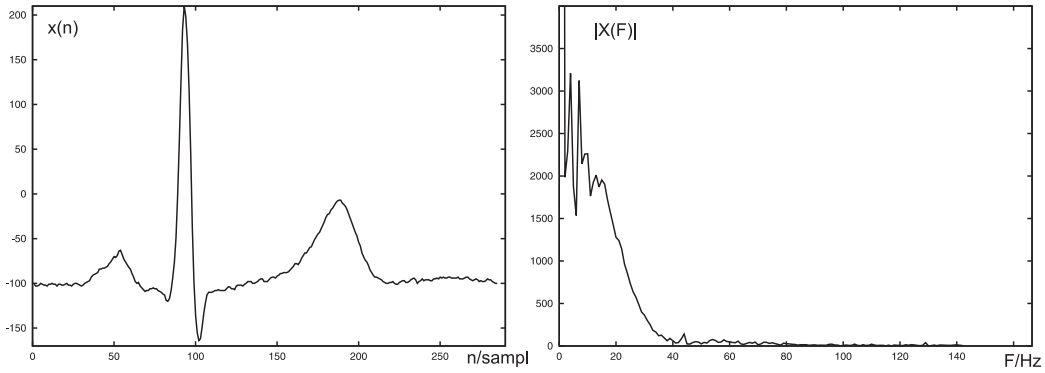

Figure 11: An ECG signal in the time- and frequency domain.

7.4 Visualisation

The Fourier spectrum is a complex function of frequency which cannot be displayed in a convenient way. Usually the amplitude or magnitude of the spectrum is of interest (see Fig. 11) which is calculated as the absolute value $|X(F)|$ of the Fourier spectrum X(F). In addition the phase might be of interest which is calculated as $\arg(X(F))$.

- Magnitude or Amplitude: $|X(F)|$
- Phase: $\arg(X(F))$

8 Causal Signal Processing

8.1 Motivation

Here are some randomly chosen reasons why we need causal signal processing:

- Fourier transform is not *real time*. We need the whole signal from the first to the last sample.
- Reaction time: in time critical systems (robotics) we want to react fast. as little delay as possible!
- We would like to "recycle" our analogue math.

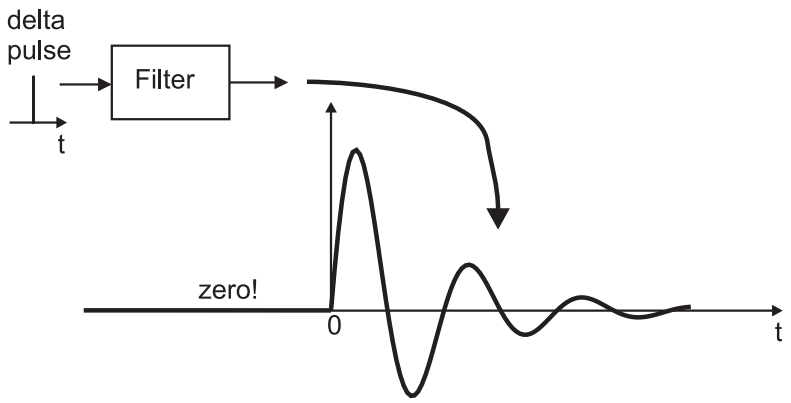

Figure 12: A causal system only *reacts* to it's input. Causal signals only evolve in positive time. Per definition the signal is zero for $t < 0$.

8.2 Causality

Fig. 12 illustrates the concept of causality. Causal systems cannot look into the future. They can only react to a certain input. Causal signals are kept zero for $t < 0$ per definition.

8.3 Convolution of Causal Signal

After having defined causality we can define the convolution:

$$y(t) = h(t) * x(t) = \int_{-\infty}^{\infty} h(t-\tau)x(\tau)d\tau y(n) = h(n) * x(n) = \sum_{n=-\infty}^{\infty} h(n)x(m-n) (75)$$

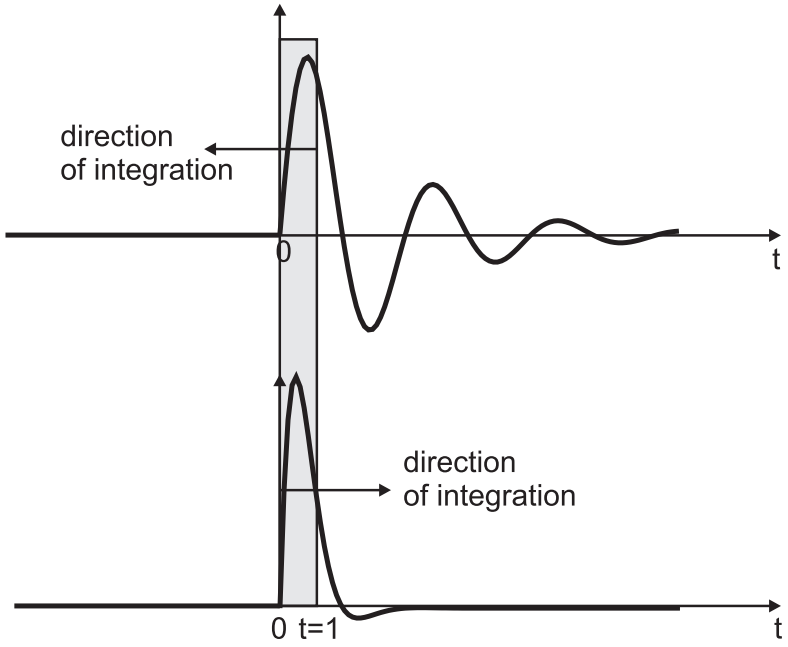

Figure 13: Illustration of the convolution. The shaded area shows the integration for $t = 1$.

Note the reversal of integration of the function h. This is characteristic of the convolution. At time $t = 0$ only the values at $h(0)$ and $x(0)$ are evaluated (see Fig. 13). Note that both functions are zero for $t < 0$ (causality!). At $t > 0$ the surface which is integrated grows as shown in Fig. 13 for $t = 1$.

What happens if $x(t) = \delta(t)$? Then Eq. 75:

$$y(t) = \int_{-\infty}^{\infty} h(t-\tau)\delta(\tau)d\tau = h(t) \tag{76}$$

provides us with the function $h(t)$ itself. This will be used later to determine the impulse response of the filter.

8.4 Laplace transform

The Fourier transform is not suitable for causal systems because it requires the whole signal from $-\infty < t < +\infty$. What we need is a transform which works with causal signals. This is the Laplace transform:

$$\mathbf{LT}(h(t)) = H(s) = \int_{0}^{\infty} h(t)e^{-st}dt \tag{77}$$

The Laplace transform has a couple of very useful properties:

- Integration:

$$\int f(\tau)d\tau \Leftrightarrow \frac{1}{s}F(s) \tag{78}$$

- Differentiation:

$$\frac{d}{dt}f(t) \Leftrightarrow sF(s) \tag{79}$$

- Shift:

$$f(t-T) \Leftrightarrow e^{-Ts}F(s) \tag{80}$$

Proof:

$$\begin{aligned}
\mathbf{LT}(h(t-T)) &= \int_0^\infty h\underbrace{(t-T)}_{causal}e^{-st}dt && (81)\\
&= \int_0^\infty h(t)e^{-s(t+T)}dt && (82)\\
&= \int_0^\infty h(t)e^{-st}e^{-sT}dt && (83)\\
&= e^{-sT}\underbrace{\int_0^\infty h(t)e^{-st}dt}_{H(s)} && (84)\\
&= e^{-sT}H(s) && (85)
\end{aligned}$$

- Convolution:

$$f(t) * g(t) \Leftrightarrow F(s)G(s) \tag{86}$$

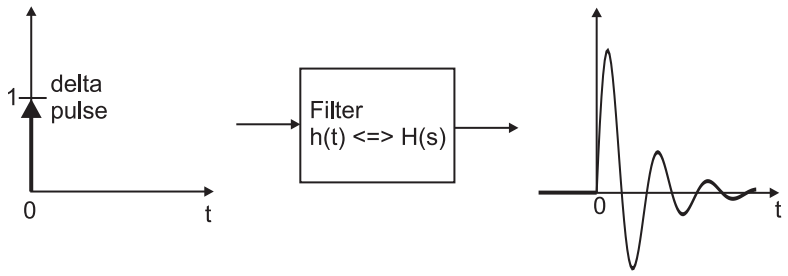

Figure 14: General idea of a filter and how to describe it: either with its impulse response or with it's Laplace transforms.

8.5 Filters

Fig. 14 presents a causal filter as a black box. We send in a signal and and we get a signal out of it. The operation of the filter is that of a convolution of the input signal or a multiplication in the Laplace space:

$$g(t) = h(t) * x(t) \Leftrightarrow Y(s) = H(s) \cdot X(s) \tag{87}$$

8.5.1 How to characterise filters?

1. **Impulse Response**

$$\begin{aligned} x(t) &= \delta(t) && \leftarrow \text{delta pulse} && (88)\\ h(t) &= y(t) && \leftarrow \text{impulse response} && (89)\\ & && && (90) \end{aligned}$$

   The filter is fully characterised by its impulse response $h(t)$

2. **Transfer function** The Laplace transform of the impulse response is called *Transfer Function*. With the argument $j\omega$ we get the frequency response of the filter. What does the frequency response tell us about the filter? The absolute value of the

$$|H(i\omega)| \tag{91}$$

   gives us the amplitude or magnitude for every frequency (compare the Fourier transform). The angle of the term $H(j\omega)$ gives us the phase shift:

$$\phi = \arg\left(H(i\omega)\right) \tag{92}$$

   of the filter. In this context the *group delay* can be defined as:

$$\tau_\omega = \frac{d\phi(\omega)}{d\omega} \tag{93}$$

   which is delay for a certain frequency ω. In many applications this should be kept constant for all frequencies.

8.6 The z-transform

The Laplace transform is for continuous causal signals but in DSP we have sampled signals. So, we need to investigate what happens if we feed a sampled signal:

$$x(t) = \sum_{n=0}^{\infty} x(n)\delta(t - nT) \qquad \text{Sampled signal} \tag{94}$$

into the Laplace transform:

$$X(s) = \sum_{n=0}^{\infty} x(n)e^{-snT} \tag{95}$$

$$= \int_0^{\infty} x(n)e^{-st}dt \tag{96}$$

$$= \sum_{n=0}^{\infty} x(n) \underbrace{(e^{-sT})^n}_{z^{-1}=e^{-sT}} \tag{97}$$

$$= \sum_{n=0}^{\infty} x(n)(z^{-1})^n \qquad \text{z-transform} \tag{98}$$

What is $e^{-sT} = z^{-1}$? It's a unit delay (see Eq. 80).

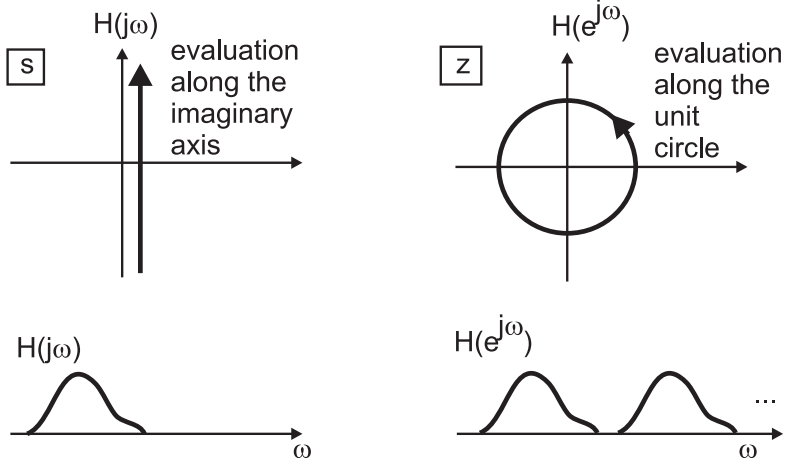

Figure 15: Comparing the calculation of the frequency response in the continuous case and sampled case.

8.7 Frequency response of a sampled filter

Reminder: for analogue Signals we had $H(s)$ with $s = j\omega$. Let's substitute s by z: $z^{-1} = e^{-sT}$ which gives us in general a mapping between the continuous domain and the sampled domain:

$$z = e^{sT} \tag{99}$$

With $s = j\omega$ this gives us $z = e^{j\omega}$ and therefore the frequency response of the digital filter is:

$$H(e^{j\omega}) \tag{100}$$

which then leads to the amplitude and phase response:

- Amplitude/Magnitude:

$$|H(e^{i\omega})| \tag{101}$$

- Phase

$$\arg H(e^{i\omega}) \tag{102}$$

Fig. 15 shows the difference between continuous and sampled signal. While in the continuous case the frequency response is evaluated along the imaginary axis, in the sampled case it happens along the unit circle which makes the response periodic! This is a subtle implication of the sampling theorem.

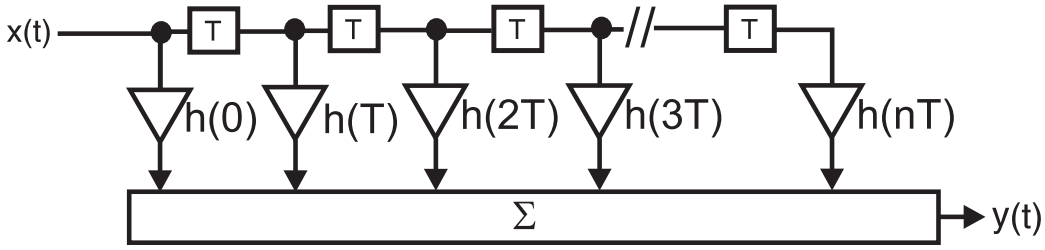

Figure 16: FIR filter

8.8 FIR Filter

What happens if we sample the impulse response $h(t)$ of an analogue filter? Let's find out:

$$h(t) = \sum_{n=0}^{\infty} h(nT)\delta(t - nT) \tag{103}$$

If we transform it to the Laplace space it looks like this:

$$H(s) = \sum_{n=0}^{\infty} h(nT)\underbrace{\left(e^{-sT}\right)}_{z^{-1}}^{n} \tag{104}$$

Remember that e^{-sT} has a very handy meaning: it is a delay by the unit time step (Eq. 80). Thus z^{-n} is a delay by n time steps. We can rewrite Eq. 104:

$$H(z) = \sum_{n=0}^{\infty} h(nT)(z^{-1})^n \tag{105}$$

This is the z-transform of the impulse response $h(t)$ of the filter.

We filter now the signal $X(z)$ with $H(z)$:

$$H(z)X(z) = \underbrace{\sum_{n=0}^{\infty} h(nT)z^{-n}}_{H(z)} X(z) \tag{106}$$

This sum is a direct recipe how to filter the signal $X(z)$. We only need the impulse response of the filter $h(nT)$ and we can set up a digital filter (see Fig. 16). Of course in practise this impulse response cannot run till infinity but only for a limited number of samples. The these are often called "taps". So for example a filter with 100 samples of $h(nT)$ has 100 "taps". This in turn then requires a delay line which can hold 100 samples.

8.8.1 FIR filter implementations

- C++: This is a simple example of a filter which stores the values in a simple linear buffer `bufferFIR` which stores the delayed values. The coefficients are stored in `coeffFIR`.

```
float filter(float value) {
  // shift
  for (int i=taps-1;i>0;i--)  {
    bufferFIR[i]=bufferFIR[i-1];
  }
  //store new value
  bufferFIR[0]=value;
  //calculate result
  for (int i=0;i<taps;i++)  {
    output +=bufferFIR[i]*coeffFIR[i];
  }
return output;
}
```

- Python: Here, the FIR filter is implemented as a class which receives the FIR filter coefficients in the constructor and then filters a signal sample by sample in the function filter:

```
class FIR_filter:
    def __init__(self,_coefficients):
        self.ntaps = len(_coefficients)
        self.coefficients = _coefficients
```

```
        self.buffer = np.zeros(self.ntaps)

    def filter(self,v):
        buffer = np.roll(buffer,1)
        self.buffer[0] = v
        return np.inner(self.buffer,self.coefficients)
```

which again processes the signal sample by sample. It uses the numpy "roll" command to shift the samples and then the inner product to calculate the weighted sum between the buffer and the coefficients.

If one wants to filter a whole array one can use Python's lfilter command:

```
import scipy.signal as signal
y = signal.lfilter(h,1,x)
```

This filters the signal `x` with the impulse response `h`. Note that this operation is on an array and thus a-causal.

More sophisticated code can be found in Teukolsky et al. (2007). This book is strongly recommended for any C programmer who needs efficient solutions.

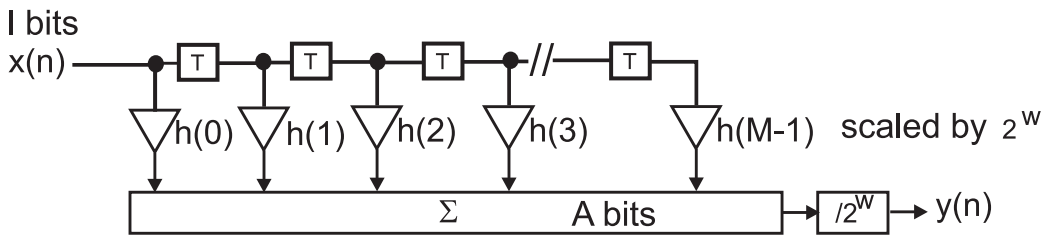

Figure 17: Fixed point FIR filter. The output signal is bit shifted to the right by w bits while the coefficients are scaled up by 2^w.

8.8.2 Fixed point FIR filters

These filters receive integer numbers as input, perform integer multiplications/additions and their outputs are integer as well. Thus, these filters do not require a floating point unit on a processor.

Fig. 17 shows a fixed point FIR filter. The input $x(n)$ is an integer variable with I bit integers, the accumulator is an integer variable with A bits and the output as well (usually the same as the input in terms of bit width).

In contrast to a floating point FIR filter we need to scale up the coefficients so that they use full the integer range to avoid quantisation errors. For example if the coefficients of $h(n)$ range from -0.75 and $+0.75$ and we have signed 16 bit integers then the scaling factor is $2^W, W = 15$.

However, the accumulator A which collects the data needs to have more bits because it receives scaled input values at I bits precision and these multiplied by factor 2^W. If we have M taps then the additional bits we need is $\log_2 M$. The total number of bits we need in the accumulator in the worst case are:

$$A = I + W + \log_2 M \tag{107}$$

However, this is the worst case scenario because if the gain of the FIR filter is below one then the summations by the M taps will only create *temporary* overflows because integer numbers are cyclic in their representation. In case the gain of the FIR filter is below one this can be relaxed:

$$A = I + W \tag{108}$$

The actual bitwidth of the accumulator is usually the next integer size available and also makes sure that in case the gain goes slightly over one in an unexpected case that the filter still works. For example if I has 16 bits the accumulator has probably 32 bits.

8.8.3 Constant group delay or linear phase filter

So far the FIR filter has no constant group delay which is defined by:

$$\tau_\omega = \frac{d\phi(\omega)}{d\omega} \tag{109}$$

This means that different frequencies arrive at the output of the filter earlier or later. This is not desirable. The group delay τ_ω should be constant for all frequencies ω so that all frequencies arrive at the same time at the output y of the filter.

A constant group delay can be achieved by restricting ourselves to the transfer function:

$$H(e^{i\omega}) = B(\omega)e^{-i\omega\tau+i\phi} \tag{110}$$

where $B(\omega)$ is real and the phase is only defined by the exponential. The phase of this transfer function is then trivially $\omega\tau + \phi$. The group delay is the derivative of this term which is constant.

Eq. 110 now imposes restrictions on the impulse response $h(t)$ of the filter. To show this we use the inverse Fourier transform of Eq. 110 to get

the impulse response:

$$h(n) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} H(e^{i\omega}) e^{i\omega n} d\omega \tag{111}$$

After some transformations we get:

$$h(n+\tau) = \frac{1}{2\pi} e^{i\phi} b(n) \tag{112}$$

where $b(n)$ represents the Fourier coefficients of $B(\omega)$. Since B is real the coefficients $b(n)$ have the property $b(n) = b^*(-n)$. Thus we get a second impulse response:

$$h(n+\tau) = \frac{1}{2\pi} e^{i\phi} b^*(-n) \tag{113}$$

Now we can eliminate b by equating Eq. 112 and Eq. 113 which yields:

$$h(n+\tau) = e^{2i\phi} n^*(-n+\tau) \tag{114}$$

With the shift τ we have the chance to make the filter "more" causal. We can shift the impulse response in positive time to get $h(n)$ zero for $n < 0$. In a practical application we shift the impulse response by half the number of delays. If we have a filter with M taps we have to delay by $\tau = M/2$.

The factor $e^{2i\phi}$ restricts the values of ϕ because the impulse response must be real. This gives us the final FIR design rule:

$$h(n + M/2) = (-1)^k h(-n + M/2) \tag{115}$$

where $k = 0$ or 1. This means that the filter is either symmetric or antisymmetric and the impulse response has to be delayed by $\tau = M/2$.

8.8.4 Window functions

So far we still have a infinite number of coefficients for for the FIR filter because there's no guarantee that the impulse response becomes zero after $M/2$ delays. Thus, we have to find a way to truncate the response without distorting the filter response.

The standard technique is to multiply the coefficients with a window function which becomes and stays zero at a certain coefficient $n > N$ so that Eq. 106 need not to run to infinity:

$$H(z)X(z) = \sum_{n=0}^{N} \underbrace{h(nT)w(nT)}\, z^{-n} X(z) \tag{116}$$

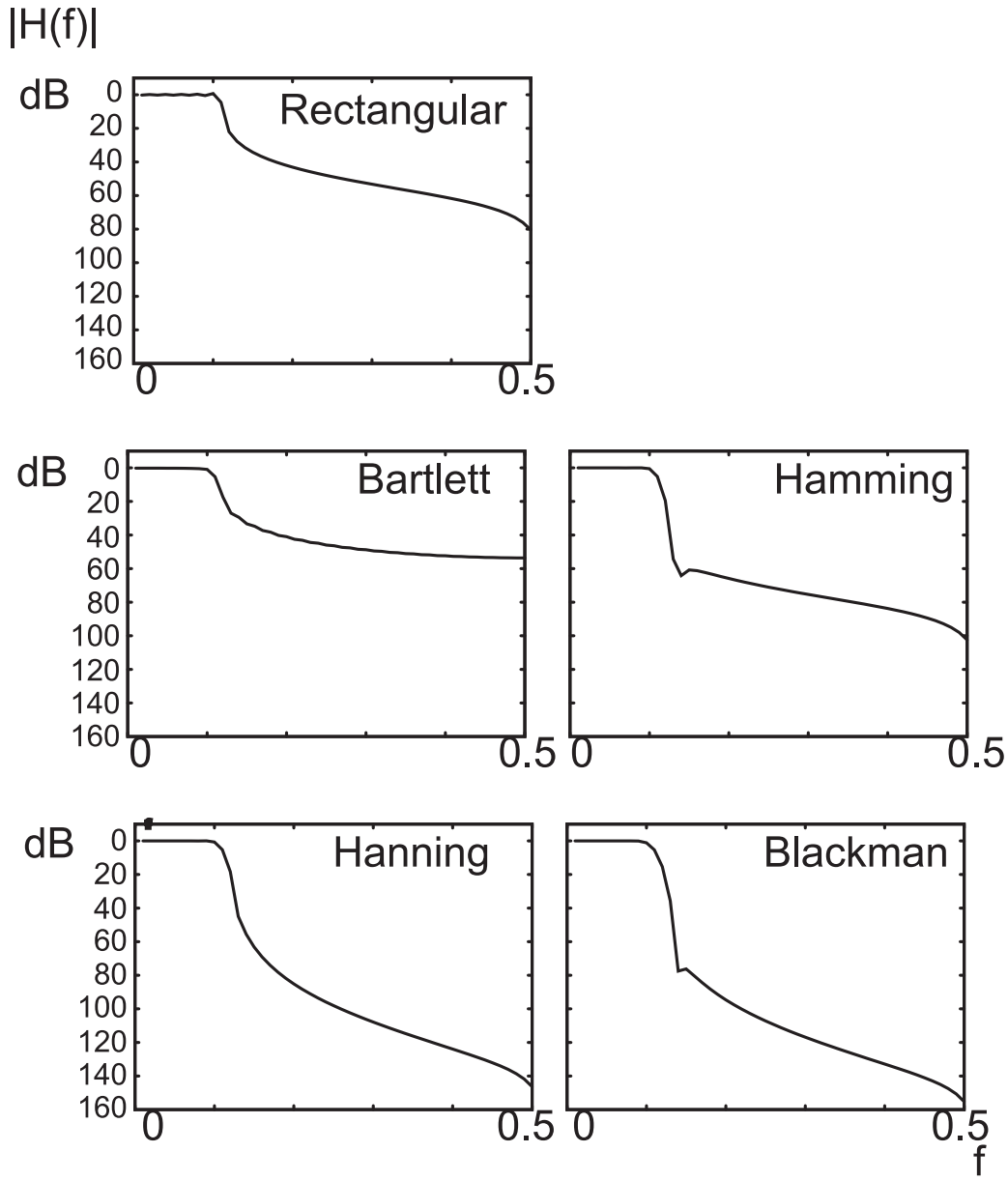

Figure 18: Different window functions applied to a low pass filter (Eq. 128) with cutoff at $f = 0.1$ and 100 taps.

1. Rectangular window: truncating the impulse response. Problem: we get ripples in the frequency-response. The stop-band damping is poor

2. Triangular or Bartlett window: greatly improved stop-band attenuation.

3. Hanning and Hamming window: standard windows in many applications.

$$w(n) = \alpha - (1-\alpha)\cos\left(\frac{2\pi n}{M}\right) \tag{117}$$

   - Hamming: $\alpha = 0.54$
   - Hanning: $\alpha = 0.5$

4. Blackman window:

$$w(n) = 0.42 + 0.5\cos\left(\frac{2\pi n}{M}\right) + 0.08\cos\left(\frac{4\pi n}{M}\right) \tag{118}$$

5. Kaiser window: control over stop- and passband. No closed form equation available.

To illustrate how window functions influence the frequency response we have taken an impulse response of a lowpass filter ($f_c = 0.1$) and applied different window functions to it (Fig. 18).

Note that the higher the damping the wider the transition from pass- to stopband. This can be seen when comparing the Blackman window with the Hamming window (Fig. 18). For the lowpass filter this seems to be quite similar. However, for a bandstop filter the wider transition width might lead actually to very poor stopband damping. In such a case a Hamming window might be a better choice.

8.8.5 Python code: impulse response from the inverse DFT - The frequency sampling method

Imagine that we want to remove $50Hz$ from a signal with sampling rate of $1kHz$. We define an FIR filter with 100 taps. The midpoint $N/2 = 50$ corresponds to $500Hz$. The $50Hz$ correspond to index 5.

```
f_resp=np.ones(100)
# note we need to add "+1" to the end because the end of
# the range is not included.
f_resp[4:6+1]=0
f_resp[94:96+1]=0
hc=np.fft.ifft(f_resp)
h=np.real(hc)
# this is from index 0 to index 49 on the left
# and on the right hand side from index 50 to index 99
h_shift[0:50]=h[50:100]
h_shift[50:100]=h[0:50]
h_wind=h_shift*hamming(100)
```

To get a nice symmetric impulse response we need to shift the inverse around 50 samples.

8.8.6 FIR filter design from ideal frequency response – The analytical way

For many cases the impulse response can be calculated analytically. The idea is always the same: define a function with the ideal frequency response

$$|H(e^{j\omega})| = \underbrace{B(e^{j\omega})}_{\text{real}} \tag{119}$$

and perform an inverse Fourier transform to get the impulse response $h(n)$.

We demonstrate this with a lowpass filter:

$$|H(e^{j\omega})| = \begin{cases} 1 & \text{for } |\omega| \leq \omega_c \\ 0 & \text{for } \omega_c < |\omega| \leq \pi \end{cases} \tag{120}$$

Use the inverse Fourier transform to get the impulse response:

$$\begin{aligned} h(n) &= \frac{1}{2\pi}\int_{-\pi}^{\pi} H(e^{j\omega})e^{j\omega n}d\omega && (121)\\ &= \frac{1}{2\pi}\int_{-\omega_c}^{+\omega_c} e^{j\omega n}d\omega && (122)\\ &= \frac{1}{2\pi}\left[\frac{1}{jn}e^{j\omega n}\right]_{-\omega_c}^{+\omega_c} && (123)\\ &= \frac{1}{2\pi jn}\left(e^{j\omega_c n} - e^{-j\omega_c n}\right) && (124) \end{aligned}$$

With these handy equations:

$$\begin{aligned} \sin z &= \frac{1}{2j}(e^{zj} - e^{-zj}) && (125)\\ \cos z &= \frac{1}{2}(e^{zj} + e^{-zj}) && (126) \end{aligned}$$

we get for the filter:

$$h(n) = \begin{cases} \frac{1}{\pi n}\sin\omega_c n & \text{for } n \neq 0 \\ \frac{\omega_c}{\pi} & \text{for } n = 0 \end{cases} \tag{127}$$

This response is a-causal! However we know that we can shift the response by any number of samples to make it causal! And we have to window the

response to get rid of any remaining negative contribution and to improve the frequency response.

Highpass, bandstop and bandpass can be calculated in exactly the same way and lead to the following ideal filter characteristics:

- **Lowpass** with cutoff frequency $\omega_c = 2\pi f_c$:

$$h(n) = \begin{cases} \frac{\omega_c}{\pi} & \text{for } n = 0 \\ \frac{1}{\pi n}\sin(\omega_c n) & \text{for } n \neq 0 \end{cases} \tag{128}$$

- **Highpass** with the cutoff frequency $\omega_c = 2\pi f_c$:

$$h(n) = \begin{cases} 1 - \frac{\omega_c}{\pi} & \text{for } n = 0 \\ -\frac{1}{\pi n}\sin(\omega_c n) & \text{for } n \neq 0 \end{cases} \tag{129}$$

- **Bandpass** with the passband frequencies $\omega_{1,2} = 2\pi f_{1,2}$:

$$h(n) = \begin{cases} \frac{\omega_2 - \omega_1}{\pi} & \text{for } n = 0 \\ \frac{1}{\pi n}(\sin(\omega_2 n) - \sin(\omega_1 n)) & \text{for } n \neq 0 \end{cases} \tag{130}$$

- **Bandstop** with the notch frequencies $\omega_{1,2} = 2\pi f_{1,2}$:

$$h(n) = \begin{cases} 1 - \frac{\omega_2 - \omega_1}{\pi} & \text{for } n = 0 \\ \frac{1}{\pi n}(\sin(\omega_1 n) - \sin(\omega_2 n)) & \text{for } n \neq 0 \end{cases} \tag{131}$$

See Diniz (2002, p.195) for more impulse responses.

Here is an example code for a bandstop filter which fills the array `h` with the analytically calculated impulse response:

```
f1 = 45.0/fs
f2 = 55.0/fs
n = np.arange(-200,200)
h = (1/(n*np.pi))*(np.sin(f1*2*np.pi*n)-np.sin(f2*2*np.pi*n))
h[200] = 1+(f1*2*np.pi-f2*2*np.pi)/np.pi;
h = h * np.hamming(400)
```

After the function has been calculated it is windowed so that the cut off is smooth. It's not very elegant as it's causes a division by zero first and then the coeffecient at 200 is fixed. Do it better!

8.8.7 Design steps for FIR filters

Here are the design steps for an M tap FIR filter.

1. Get yourself an impulse response for your filter:
   (a) Create a frequency response "by hand" just by filling an array with the desired frequency response. Then, perform an inverse Fourier transform (see section 8.8.5).
   (b) Define a frequency response analytically. Do an inverse Fourier transform (see section 8.8.6).
   (c) Use an analogue circuit, get its impulse response and use these as filter coefficients.
   (d) Dream up directly an impulse response (for example, averagers, differentiators, etc)
2. Mirror the impulse response (if not already symmetrical)
3. Window the impulse response from an infinite number of samples to M samples.
4. Move the impulse response to positive time so that it becomes causal (move it $M/2$ steps to the right).

8.8.8 FIR filter design with Python's high level functions

The "firwin" command generates the impulse response of a filter with m taps and also applies a default window function. For example:

```
from scipy.signal import firwin
h = firwin(m,2*f)
```

generates a lowpass FIR filter with the normalised frequency f. Note the factor two because in scipy the normalisation is *not* the sampling rate but the *nyquist frequency*. To be compatible with the math here the easiest approach is to multiply the frequency by 2. With this command one can also design high pass, bandstop and bandpass filters. Type `help(firwin)` which provides examples for all filter types and which parameters need to be set.

8.9 Signal Detection

How can I detect a certain event in a signal? With a correlator. Definition of a correlator?

$$e(t) = \int_0^t \underbrace{s(\tau)}_{\text{signal}} \quad \underbrace{r(\tau)}_{\text{template}} \quad d(\tau) \tag{132}$$

How to build a correlator with a filter? Definition of filtering?

$$e(t) = \int_0^\infty s(\tau)h(t-\tau)d\tau \tag{133}$$

NB. h - integration runs backwards. However we used an int forward! $h(t) := r(T-\tau)$, only valid for $0 \ldots T$.

$$\begin{aligned} e(t) &= \int_0^T s(\tau)r\left(T-(t-\tau)\right)d\tau & (134)\\ &= \int_0^T s(\tau)r(T-t+\tau)d\tau & (135) \end{aligned}$$

for $t := T$ we get:

$$e(T) = \int_0^\infty s(\tau)r(\tau)d\tau \tag{136}$$

$$\underbrace{h(t) := r(T-t)}_{matched \quad filter!} \tag{137}$$

In order to design a detector we just create an impulse response h by reversing the template r in time and constructing an FIR filter with it.

How to improve the matching process? Square the output of the filter!

8.10 IIR Filter

IIR filter stands for Infinite Impulse Response. Such filters are implemented as recursive filters. We will see that impulse responses from exponentials can easily be implemented as a recursive filter.

8.10.1 Introduction

We'll try to find a recursive version of a discrete filter. To demonstrate how this works we take a very simple analogue filter and sample it. This example can then be generalised to more complex situations.

We define a first order filter which can be implemented, for example, as a simple RC network:

$$h(t) = e^{-bt} \tag{138}$$

where its Laplace transform is a simple fraction:

$$H(s) = \frac{1}{s+b} \tag{139}$$

Now we sample Eq. 138:

$$h(t) = \sum_{n=0}^{\infty} e^{-bnT} \cdot \delta(t - nT) \tag{140}$$

and perform a Laplace transform on it:

$$H(s) = \sum_{n=0}^{\infty} e^{-bnT} \underbrace{e^{-nsT}}_{z^{-1^n}} \tag{141}$$

which turns into a z-transform:

$$\begin{aligned} H(z) &= \sum_{n=0}^{\infty} e^{-bnT} z^{-n} && (142) \\ &= \sum_{n=0}^{\infty} \left(e^{-bT} z^{-1}\right)^n && (143) \\ &= \frac{1}{1 - e^{-bT} z^{-1}} && (144) \end{aligned}$$

Consequently the analogue transfer function $H(s)$ transfers into $H(z)$ with the following recipe:

$$H(s) = \frac{1}{s+b} \qquad \Leftrightarrow \qquad H(z) = \frac{1}{1 - e^{-bT} z^{-1}} \tag{145}$$

Thus, if you have the poles of an analogue system and you want to have the poles in the z-plane you can transfer them with:T

$$z_\infty = e^{s_\infty T} \tag{146}$$

his also gives us a stability criterion. In the Laplace domain the poles have to be in the left half plane (imaginary value negative). This means that in the sampled domain the poles have to lie within the unit circle.

The same rule can be applied to zeros

$$z_0 = e^{s_0 T} \tag{147}$$

together with Eq. 146 this is called "The matched z-transform method". For example $H(s) = s$ turns into $H(z) = 1 - z^{-1} e^{0T} = 1 - z^{-1}$ which is basically a DC filter.

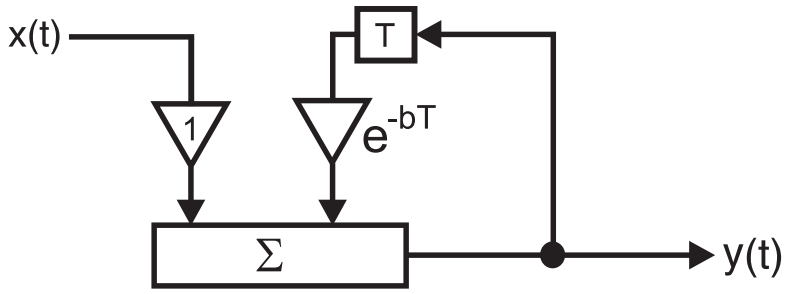

Figure 19: IIR filter

8.10.2 Determining the data-flow diagram of an IIR filter

To get a practical implementation of Eq. 144 we have to see it with its input- and output-signals:

$$\begin{aligned} Y(z) &= X(z)H(z) && (148)\\ &= X(z)\frac{1}{1-e^{-bT}z^{-1}} && (149)\\ &= Y(z)z^{-1}e^{-bT} + X(z) && (150) \end{aligned}$$

We have to recall that z^{-1} is the delay by T. With that information we directly have a difference equation in the temporal domain:

$$y(nT) = y([n-1]T)e^{-bT} + x(nT) \tag{151}$$

This means that the output signal $y(nT)$ is calculated by adding the weighted and delayed output signal $y([n-1]T)$ to the input signal $x(nT)$. How this actually works is shown in Fig. 19. The attention is drawn to the *sign inversion* of the weighting factor e^{-bT} in contrast to the transfer function Eq. 145 where it is $-e^{-bT}$. In general the recursive coefficients change sign when they are taken from the transfer function.

8.10.3 General form of filters

A filter with forward and feedback components can be represented as:

$$H(z) = \frac{\sum_{k=0}^{r} B_k z^{-k}}{1 + \sum_{l=1}^{m} A_i z^{-l}} \tag{152}$$

where B_k are the FIR coefficients and A_l the recursive coefficients. Note the signs of the recursive coefficients are *inverted* in the actual implementation of the filter. This can be seen when the function $H(z)$ is actually multiplied

with an input signal to obtain the output signal (see Eq. 151 and Fig. 19). The "1" in the denominator represents actually the output of the filter. If this factor is not one then the output will be scaled by that factor. However, usually this is kept one.

In Python filtering is performed with the command:

```
import scipy.signal as signal
Y = signal.lfilter(B,A,X)
```

where B are the FIR coefficients, A the IIR coefficients and X is the input. For a pure FIR filter we just have:

```
Y = signal.lfilter(B,1,X)
```

The "1" represents the output.

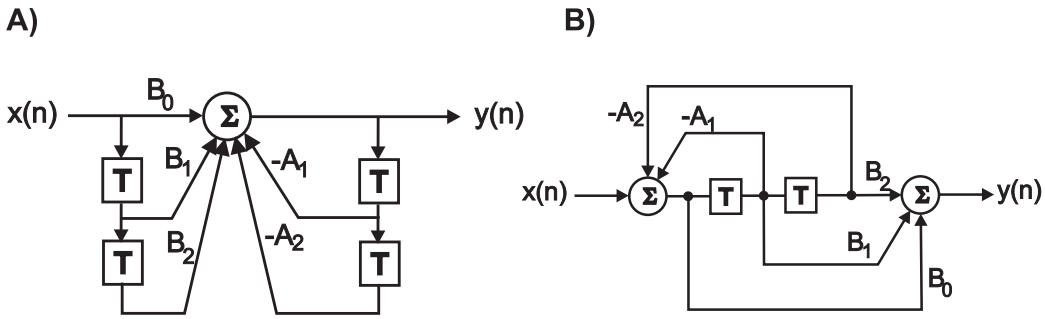

Figure 20: A) Direct Form I filter which one accumulator and B) Direct Form II filter with two accumlators.

8.10.4 IIR filter topologies

In real applications one would create a *chain of 2nd order IIR filters*. This has numerous advantages:

1. The optimisation can be done within these simple structures, for example omitting array operations completely and coding the 2nd order filters in assembly while keeping the higher level operations in C or C++.

2. Control of stability issues: high order recursive systems might become unstable and it is very hard to predict when this might happen. On the other hand a chain of 2nd order filters can be made stable with ease. One can focus on stability in 2nd order systems which is well understood and manageable.

3. Complex conjugate pole pairs are generated naturally in analogue filter design and directly translate to 2nd order IIR structures. Analogue design is usually described as 2nd order structures (=LCR) so that any transform from analogue to digital just needs to be of 2nd order!

Fig. 20 shows the most popular filter topologies: Direct Form I and II. Because of the linear operation of the filter one is allowed to do the FIR and IIR operations in different orders. In Direct From I we have one accumulator and two delay lines whereas in the Direct Form II we have two accumulators and one delay line. Only the Direct Form I is suitable for integer operations.

A Python class of a direct form II filter can be implemented with a few lines:

```
class IIR_filter:
    def __init__(self,_num,_den):
        self.numerator = _num
        self.denominator = _den
        self.buffer1 = 0
        self.buffer2 = 0

    def filter(self,v):
        input=0.0
        output=0.0
        input=v
        output=(self.numerator[1]*self.buffer1)
        input=input-(self.denominator[1]*self.buffer1)
        output=output+(self.numerator[2]*self.buffer2)
        input=input-(self.denominator[2]*self.buffer2)
        output=output+input*self.numerator[0]
        self.buffer2=self.buffer1
        self.buffer1=input
        return output
```

Here, the two delay steps are represented by two variables `buffer1` and `buffer2`.

In order to achive higher order filters one can then just chain these 2nd order filters. In Python this can be achieved by storing these in an array of instances of this class.

8.10.5 Fixed point IIR filters

Fig. 21 shows the implementation of a fixed point IIR filter. It's a Direct Form I filter with one accumulator so that temporary overflows can be com-

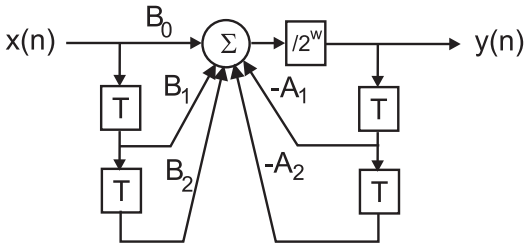

Figure 21: Direct Form I filter with fixed point arithmetic.

pensated. The original floating point IIR coefficients are scaled up by factor 2^w so that they max out the range of the integer coefficients. After the addition operation they are shifted back by w bits to their original values.

For example if the largest IIR coefficient is 1.9 and we use 16 bit signed numbers (max number is 32767) then one could multiply all coefficients with 2^{14}.

Then one needs to assess the maximum value in the accumulator which is much more difficult than for FIR filters. For example, resonators can generate high output values with even small input values so that it's advisable to have a large overhead in the accumulator. For example if the input signal is 16 bit and the scaling factor is 14 bits then the signal will certainly occupy $16+14 = 30$ bits. With an accumulator of 32 bits that gives only a headroom of 2 bits so the output can only be 4 times larger than the input. A 64 bit accumulator is a safe bet.

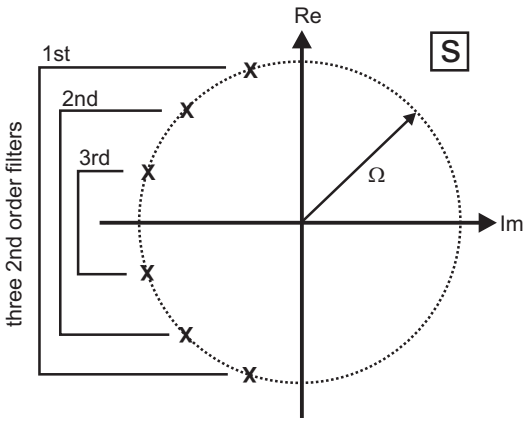

Figure 22: Pole placements of the Butterworth filter. Its analogue cutoff frequency is Ω. This filter can be implemented as a chain of 2nd order IIR filters (biquads) by using the complex conjugate pairs for the different 2nd order IIR filters.

8.10.6 Filter design from analogue filters

Traditionally digital filters were designed from analogue filters by transforming from the continuous domain to the sampled domain. This has historical reasons because before digital signal processing became popular most designs were analogue and used Transfer functions in the Laplace domain. Some popular analogue transfer functions are:

- **Butterworth:** All poles lie on the left half plane equally distributed on a half circle with radius Ω which is shown in Fig. 22.
  - monotonic
  - only poles, no zeros
  - standard filter for many applications
  - no constant group delay!
- **Chebyshev Filters:**

$$|H(\Omega)|^2 = \frac{1}{1 - \varepsilon^2 T_N(\Omega/\Omega_p)} \tag{153}$$

  where T = Chebyslev polynomials.

- **Bessel Filter:**
  - Constant Group Delay
  - Shallow transition from stop- to passband.

These analogue transfer functions need to be transformed in the sampled domain. This could be done by the impulse invariance / matched z-transform method but the problem with these methods is that they map frequencies 1:1 between the digital and analogue domain. Remember: in the sampled domain there is no infinite frequency but only $F_s/2$. It ends at the Nyquist frequency which means that we never get more damping than at $F_s/2$. This is especially a problem for lowpass filters where damping increases the higher the frequency.

The solution is to map **all** analogue frequencies from $0 \ldots \infty$ to $0 \ldots 0.5$ (normalised frequency) in a non-linear way:

$$-\infty < \Omega < \infty \Rightarrow -\pi \leq \omega \leq \pi \tag{154}$$

This is called *Bilinear Transformation*:

$$s = \frac{2}{T} \quad \frac{z-1}{z+1} \tag{155}$$

Let's test if the mapping works. In the analogue domain the frequency is given as $s = j\Omega$ and in the sampled domain as $z = e^{j\omega}$.

$$j\Omega = \frac{2}{T}\left[\frac{e^{j\omega}-1}{e^{j\omega}+1}\right] = \frac{2}{T} j \tan\frac{\omega}{2} \tag{156}$$

Note, that the bilinear transform is a **non**linear mapping of the frequency:

$$\Omega = \frac{2}{T} tan \frac{\omega}{2} \tag{157}$$

So, the cut-off frequency of our analogue filter is changed by the bilinear transformation. Consequently, we need a pre-warp of the analogue filter frequency:

$$\Omega_c = \frac{2}{T} tan \frac{\omega_c}{2} \tag{158}$$

where T is the sampling interval.

The general design steps are:

1. Choose the cut-off frequency of your digital filter ω_c.
2. Pre-warp $\omega_c \rightarrow \Omega_c$ with Eq. 158
3. Choose your favourite analogue lowpass filter $H(s)$, for example Butterworth.
4. Replace all s in the analogue transfer function $H(s)$ by $\frac{2}{T}\frac{z-1}{z+1}$ to obtain the digital filter $H(z)$
5. Change the transfer function $H(z)$ so that it contains only negative powers of z ($z^{-1}, z^{-2}, \ldots$) which can be interpreted as delay lines.
6. Build your IIR filter!

For filter orders higher than two one needs to develop a different strategy because the bilinear transform is a real *pain* to calculate for anything above order two. Nobody wants to transform high order analogue transfer functions $H(s)$ to the $H(z)$ domain. However, there is an important property of all analogue transfer functions: they generate complex conjugate pole pairs (plus one real pole if of odd order) which suggest a chain of 2nd order IIR filters straight away (see Fig. 22). Remember that a complex conjugate pole pair creates a 2nd order IIR filter with with two delay steps. A real pole is a 1st order IIR filter with one delay but is often also implemented as a 2nd order filter where the coefficients of the 2nd delay are kept zero.

The design strategy is thus to split up the analogue transfer function $H(s)$ in a chain of 2nd order filters $H(s) = H_1(s)H_2(s)H_3(s)\ldots$ and then to apply the bilinear transform on every 2nd order term separately. Using this strategy you only need to calculate the bilinear transform once for a 2nd order system (or if the order is odd then also for a 1st order one) but then there is no need to do any more painful bilinear transforms. This is standard practise in IIR filter design.

8.11 The role of poles and zeros

Transfer functions contain poles and zeros. To gain a deeper understanding of the transfer functions we need to understand how poles and zeros shape the frequency response of $H(z)$. The position of the poles also determines the stability of the filter which is important for real world applications.

We are going to explore the roles of poles and zeros first with an instructional example which leads to a 2nd order bandstop filter.

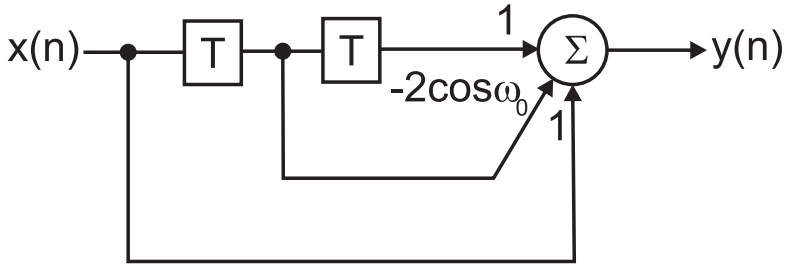

Figure 23: A two tap FIR stopband filter for the frequency ω_0.

8.11.1 Zeros

Let's look at the transfer function:

$$\begin{aligned} H(z) &= (1 - e^{j\omega_0} z^{-1})(1 - e^{-j\omega_0} z^{-1}) && (159)\\ &= 1 - z^{-1} e^{j\omega - 0} + z^{-2} && (160)\\ &= 1 - z^{-1}(e^{j\omega_0} + e^{-j\omega_0}) + z^{-2} && (161)\\ &= 1 - z^{-1} 2\cos\omega_0 + z^{-2} && (162) \end{aligned}$$

which leads to the data flow diagram shown in Fig 23.

$$H(e^{j\omega}) = \underbrace{(1 - e^{j\omega_0} e^{-j\omega})(1 - e^{-j\omega_0} e^{-j\omega})}_{2 \text{ zeros}} \quad (163)$$

The zeroes at $e^{j\omega_0}$ and $e^{-j\omega_0}$ eliminate the frequencies ω_0 and $-\omega_0$.

A special case is $\omega_0 = 0$ which gives us:

$$\begin{aligned} H(z) &= (1 - e^0 z^{-1})(1 - e^0 z^{-1}) && (164) \\ &= 1 - 2z^{-1} + z^{-2} && (165) \end{aligned}$$

a DC filter.

In summary: zeros eliminate frequencies (and change phase). That's the idea of an FIR filter where loads of zeros (or loads of taps) knock out the frequencies in the stopband.

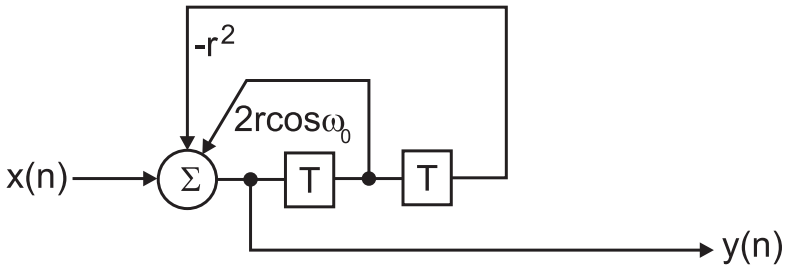

Figure 24: A two tap IIR resonator for the frequency ω_0 and with the amplification r.

8.11.2 Poles

While zeros knock out frequencies, poles amplify frequencies. Let's investigate complex conjugate poles:

$$H(z) = \underbrace{\frac{1}{(1 - re^{j\omega_0} z^{-1})(1 - re^{-j\omega_0} z^{-1})}}_{2poles!} \quad (166)$$

which are characterised by their resonance frequency ω_0 and the amplification $0 < r < 1$.

In order to obtain a data flow diagram we need to get powers of z^{-1} because they represent delays.

$$H(z) = \frac{1}{1 - 2r\cos(\omega_0)z^{-1} + r^2 z^{-2}} \quad (167)$$

which we multiply with the input signal $Y(z)$:

$$\begin{aligned} Y(z) &= X(z)\frac{1}{1 - 2r\cos(\omega)z^{-1} + r^2 z^{-2}} && (168) \\ X(z) &= Y(z) - Y(z)2r\cos(\omega_0)z^{-1} + Y(z)r^2 z^{-2} && (169) \\ Y(z) &= X(z) + z^{-1}Y(z)2r\cos(\omega_0) - z^{-2}Y(z)r^2 && (170) \end{aligned}$$

This gives us a second order recursive filter (IIR) which is shown in Fig 24. These complex conjugate poles generate a resonance at $\pm\omega_0$ where the amplitude is determined by r.

8.11.3 Stability

A transfer function $H(z)$ is only stable if the poles lie inside the unit circle. This is equivalent to the analog case where the poles of $H(s)$ have to lie on the left hand side of the complex plane (see Eq. 146). Looking at Eq. 166 it becomes clear that r determines the radius of the two complex conjugate poles. If $r > 1$ then the filter becomes unstable. The same applies to poles on the real axis. Their the real values have to stay within the range $-1 \ldots +1$.

In summary: poles generate resonances and amplify frequencies. The amplification is strongest the closer the poles move towards the unit circle. The poles need to stay within the unit circle to guarantee stability.

Note that in real implementations the coefficients of the filters are limited in precision. This means that a filter might work perfectly in python but will fail on a DSP with its limited precision. You can simlate this by forcing a certain datatype in python or you write it properly in C.

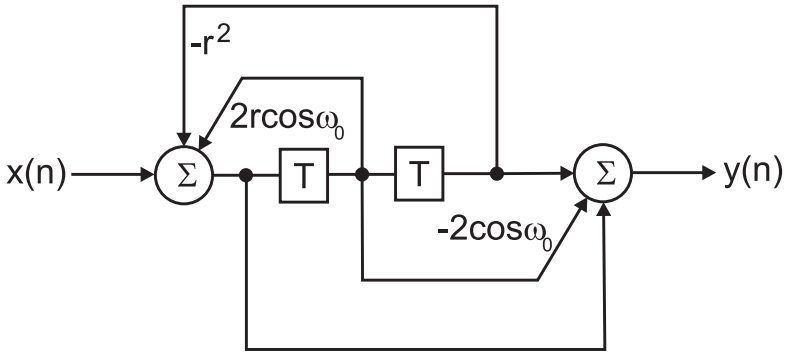

Figure 25: A two tap IIR bandstop filter with tuneable stopband width r for the frequency ω_0.

8.11.4 Design of an IIR notch filter

Now we combine the filters from the last two sections:

$$H(z) = \frac{1 - 2\cos(\omega_0)z^{-1} + z^{-2}}{1 - 2r\cos(\omega_0)z^{-1} + r^2 z^{-2}} \tag{171}$$

This gives us a notch filter where its width is tunable with the help of $r < 1$. The closer r goes towards 1 the more narrow is the frequency response. This

filter has two poles and two zeros. The zeros sit on the unit circle and eliminate the frequencies $\pm\omega_0$ while the poles sit within the unit circle and generate a resonance around $\pm\omega_0$. As long as $r < 1$ this resonance will not go towards infinity at $\pm\omega_0$ so that the zeros will always eliminate the frequencies $\pm\omega_0$.

A C++ implementation of such a filter is quite compact and requires no loops:

```
float Iirnotch::filter(float value) {
  float input=0.0;
  float output=0.0;
  // a little bit cryptic but optimized for speed
  input=value;
  output=(numerator[1]*buffer[1]);
  input=input-(denominator[1]*buffer[1]);
  output=output+(numerator[2]*buffer[2]);
  input=input-(denominator[2]*buffer[2]);
  output=output+input*numerator[0];
  buffer[2]=buffer[1];
  buffer[1]=input;
  return output;
}
```

This filter is part of the program `comedirecord` and is used to filter out 50Hz noise.

8.11.5 Identifying filters from their poles and zeroes

Proakis and Manolakis (1996, pp.333) has an excellent section about this topic and we refer the reader to have a look. As a rule of thumb a digital lowpass filter has poles where their real parts are positive and a highpass filter has poles with negative real part of the complex poles. In both cases they reside within the unit circle to guarantee stability.

9 Limitations / outlook

So far the coefficients have been constant in the filters. However, in many situations these coefficients need to change while the filter is operating. For example, an ECG with broadband muscle noise requires an adaptive approach because the noise level is unknown. The noise and the ECG overlap in their spectra. We need a lowpass filter which does a tradeoff between loosing the signal and filtering out most of the noise: The problem is that the

signal to noise ratio is constantly changing. A fixed cut off is not a good option. We need to change the cut off all the time. One example is a Kalman filter. The idea is to maximise the *predictability* of the filtered signal. A lowpass filter increases the predictability of a signal because it smoothes the signal.

$$H(z) = \frac{b}{1 - az^{-1}} \tag{172}$$

The parameter a determines the cutoff frequency. Frequency Response :

$$|H(e^{j\omega})| = |\frac{b}{1 - ae^{-j\omega}}| \tag{173}$$

Let's re-interpret our low-pass filter in the time domain:

$$\underbrace{y(n)}_{\text{actual estimate}} = a(n) \underbrace{y(n-1)}_{\text{previous estimate}} + b(n) \underbrace{x(n)}_{\text{current data sample}} \tag{174}$$

We would like to have the best estimate for $y(n)$

$$p(n) = E[(y(k) - y_{real}(k))^2] \tag{175}$$

We need to minimise p which gives us equations for a and b which implements a Kalman filter.

Transmission line equalisation also requires an adaptive approach where the coefficients of the filter are determined by a known training sequence. The coefficients are changed until the desired training sequence appears at the output of the filter.

References

Diniz, P. S. R. (2002). *Digital Signal Processing*. Cambridge university press, Cambridge.

Proakis, J. G. and Manolakis, D. G. (1996). *Digital Signal Processing*. Prentice-Hall, New Jersey.

Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition.