COMP.SGN.100 Introduction to Signal Processing

Sari Peltonen
Signal Processing
Computing Sciences
Tampere University

Preface

This lecture handout is meant for the course ”COMP.SGN.100 Introduction to Signal Processing” lectured at Tampere University. It is a rough English translation of the handout for the equivalent Finnish course ”COMP.SGN.100 Signaalinkäsittelyn perusteet” written by Heikki Huttunen. Additionally, a brief introduction to Matlab is included in the beginning of this handout.

After the Matlab introduction the handout provides the basics of digital signal processing, such as sampling theorem, discrete signals and systems, convolution operator, Fourier transform, FFT algorithm, z-transform, transfer function, filter design and multirate. Furthermore, we consider applications of signal processing algorithms in pattern recognition and machine learning.

The aim is to discover the basics of linear systems and digital signal processing and to obtain an overview of some applications. After the course the student understands what it is like to work with signal processing and what type of problems it can be applied to.

August 2020
Sari Peltonen
sari.peltonen@tuni.fi

Contents

Chapter 1

Matlab

We consider here briefly the very basics of Matlab to have a common tool to realize the signal processing tasks on this course. If you already master the basics of Matlab, then you can skip Chapter 1 and move directly to Chapter 2.

Matlab (**mat**rix **lab**oratory) has become a standard in scientific computing. The development of Matlab started in the 1970's as an easier alternative to Fortran. Since then, it has grown as a popular environment in various areas of science and technology. It is a proprietary programming language licensed by MathWorks.

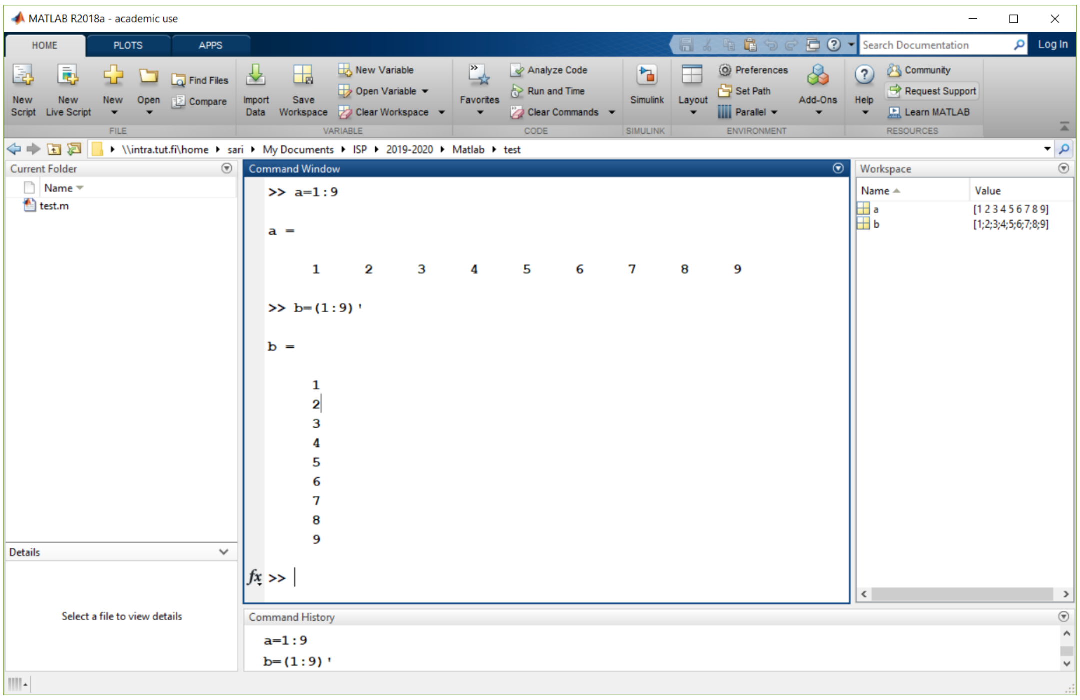
Tampere University has an extensive license allowing, e.g., installation on students' own computers.

Open source alternatives to Matlab are Python, R, Octave and Scilab. Some pros and cons of Matlab are listed in the table below.

Pros	Cons
Extremely efficient linear algebra routines	Built on top of Matlab interpreter, thus (can be) slower than compiled languages
Easy to use	
Large number of toolboxes	Closed source, requires license
Rapid development and prototyping	License is expensive for non-academics
Great visualization tools	Some parts of syntax

1.1 Matlab Basics

Matlab desktop shown below consists of (left to right): file browser, file details, command window, command history and variable explorer.



```
>> a=1:9

a =

     1     2     3     4     5     6     7     8     9

>> b=(1:9)'

b =

     1
     2
     3
     4
     5
     6
     7
     8
     9

fx >> 
```

Name	Value
a	[1 2 3 4 5 6 7 8 9]
b	[1;2;3;4;5;6;7;8;9]

```
a=1:9
b=(1:9)'
```

After starting Matlab, we can enter the commands directly to the command window. The commands are interpreted and Matlab responds with the results. For example, a variable is defined with the assignment operator (=):

```
>> x = 1

x =

     1
```

As a result, the variable explorer in the right hand side of the desktop shows your new variable.

For each command, help is there to refresh your memory:

```
>> help inv
 inv    Matrix inverse.
    inv(X) is the inverse of the square matrix X.
    A warning message is printed if X is badly scaled or
    nearly singular.

    See also slash, pinv, cond, condest, lsqnonneg, lscov.

    Reference page in Help browser
       doc inv
```

A browser based view of help can be opened by `doc inv`. Note that the content is different and more verbose, so this is a matter of taste.

Vector is the basic building block for storing data. Typical definitions either list the values in block parentheses, or with the colon operator.

```
>> a = [1,2,3,4]

a =

     1     2     3     4

>> b = 1:4

b =

     1     2     3     4
```

By default, the colon operator defines a row vector. Usually we wish to use column vectors instead, which is done as follows.

```
>> a = [1;2;3;4] % ';' is the column separator

a =

     1
     2
     3
     4

>> b = (1:4)'

b =

     1
     2
     3
     4
```

It is also possible to define the step between elements:

```
>> a = 1:0.5:4 % Increment the elements by 0.5 steps

a =

  Columns 1 through 5

    1.0000    1.5000    2.0000    2.5000    3.0000

  Columns 6 through 7

    3.5000    4.0000

>> b = 4:-1:1 % Increment is now negative

b =

     4     3     2     1
```

A matrix is defined similarly; either by specifying the values manually, or using special functions.

```
>> A = [1,2;3,4] % Write elements row by row separated by semicolons

A =

     1     2
     3     4

>> I = eye(3) % eye is a function for the identity matrix.

I =

     1     0     0
     0     1     0
     0     0     1
```

Matrix multiplication is straightforward:

```
>> A = [1,2;3,4]; % Semicolon at the end tells Matlab not to display
    any output
>> b = [5;6];
>> A*b

ans =

    17
    39
```

Also matrix inversion has its own command.

```
>> inv(A)

ans =

   -2.0000    1.0000
    1.5000   -0.5000

>> inv(A)*A

ans =

    1.0000         0
    0.0000    1.0000
```

Indexing of vectors uses the colon notation, too. Below, we extract selected items from the vector `1:10`.

```
>> x = 1:10;
>> everySecond = x(1:2:10)

everySecond =

     1     3     5     7     9

>> everyThird = x(1:3:end)

everyThird =

     1     4     7    10
```

Note the use of `end` if you do not want to calculate what is the last index.

Also matrices can be indexed similarly (often called *slicing*). Below we request for items on the rows 2,3,4 and columns 1,3,4 (shown in red). Note, that with matrices, the first index specifies the row; not the "x-coordinate". This order is called "Fortran style" or "column major" while the alternative is "C style" or "row major".

```
>> M = reshape(1:36, 6, 6)'

M =

     1     2     3     4     5     6
     7     8     9    10    11    12
    13    14    15    16    17    18
    19    20    21    22    23    24
    25    26    27    28    29    30
    31    32    33    34    35    36

>> items = M(2:4, [1,3,4])

items =

     7     9    10
    13    15    16
    19    21    22
```

To specify only column or row indices (not both), use the colon alone. Now we wish to extract rows 5 and 6 completely. `M(5:end, :)` reads "give me rows 5 and 6 and all columns".

```
>> M = reshape(1:36, 6, 6)'

M =

     1     2     3     4     5     6
     7     8     9    10    11    12
    13    14    15    16    17    18
    19    20    21    22    23    24
    25    26    27    28    29    30
    31    32    33    34    35    36

>> items = M(5:end, :)

items =

    25    26    27    28    29    30
    31    32    33    34    35    36
```

The indexing can also appear on the left hand side of the assignment. Here we change the values within the block on rows 5-6 and columns 2-4.

```
>> M = reshape(1:36, 6, 6)'

M =

     1     2     3     4     5     6
     7     8     9    10    11    12
    13    14    15    16    17    18
    19    20    21    22    23    24
    25    26    27    28    29    30
    31    32    33    34    35    36

>> M(5:6, 2:4) = [-1, -2, -3; -4, -5, -6]

M =

     1     2     3     4     5     6
     7     8     9    10    11    12
    13    14    15    16    17    18
    19    20    21    22    23    24
    25    -1    -2    -3    29    30
    31    -4    -5    -6    35    36
```

One can also assign a fixed value to selected coordinates. This includes also the empty matrix. Here, we assign the empty matrix in place of 4 rows, essentially removing those. Next, we assign a fixed scalar -10 to 4 first entries of the last row.

```
>> M = reshape(1:36, 6, 6)'

M =
     1     2     3     4     5     6
     7     8     9    10    11    12
    13    14    15    16    17    18
    19    20    21    22    23    24
    25    26    27    28    29    30
    31    32    33    34    35    36

>> M(2:5, :) = []

M =
     1     2     3     4     5     6
    31    32    33    34    35    36

>> M(end, 1:4) = -10

M =
     1     2     3     4     5     6
   -10   -10   -10   -10    35    36
```

There are several functions that generate often appearing matrices. Usually the input defines the size (columns and rows). If only one parameter is given (e.g., `rand(10)`), the result is a square matrix. A common error is to forget the singleton dimension: `x = rand(10000)` instead of `x = rand(10000, 1)`. This will create a 10000 × 10000 matrix that probably makes the computer stuck.

```
>> I = eye(3)

I =

     1     0     0
     0     1     0
     0     0     1

>> x = zeros(1,5)

x =

     0     0     0     0     0

>> y = ones(1,5)

y =

     1     1     1     1     1

>> r = rand(1,5)

r =

    0.1203    0.6255    0.3466    0.3346    0.5746
```

Matlab is designed for linear algebra, so matrix operations are very handy. Here we take the dot product (inner product) between two vectors. Most arithmetic operations have also their elementwise versions:

- `a.*b`: elementwise multiplication,
- `a./b`: elementwise division,
- `a.^b`: elementwise exponentiation (`2.^b` or `b.^2`).

```
>> a = [1,2,3,4], b = [-1,-2,-3,-4]

a =

     1     2     3     4


b =

    -1    -2    -3    -4

>> a*b' % Equivalent to "dot(a,b)"

ans =

   -30

>> a.*b % Elementwise multiplication

ans =

    -1    -4    -9   -16

>> 2.^b % Elementwise power

ans =

    0.5000    0.2500    0.1250    0.0625
```

Matlab includes a huge variety of mathematical functions. Typically one uses vector operations. This is much faster than for loop.

```
>> x = 0:0.5:pi;
>> y = exp(-i * x)

y =
  Columns 1 through 3

   1.0000 + 0.0000i   0.8776 - 0.4794i   0.5403 - 0.8415i

  Columns 4 through 6

   0.0707 - 0.9975i  -0.4161 - 0.9093i  -0.8011 - 0.5985i

  Column 7

  -0.9900 - 0.1411i
```

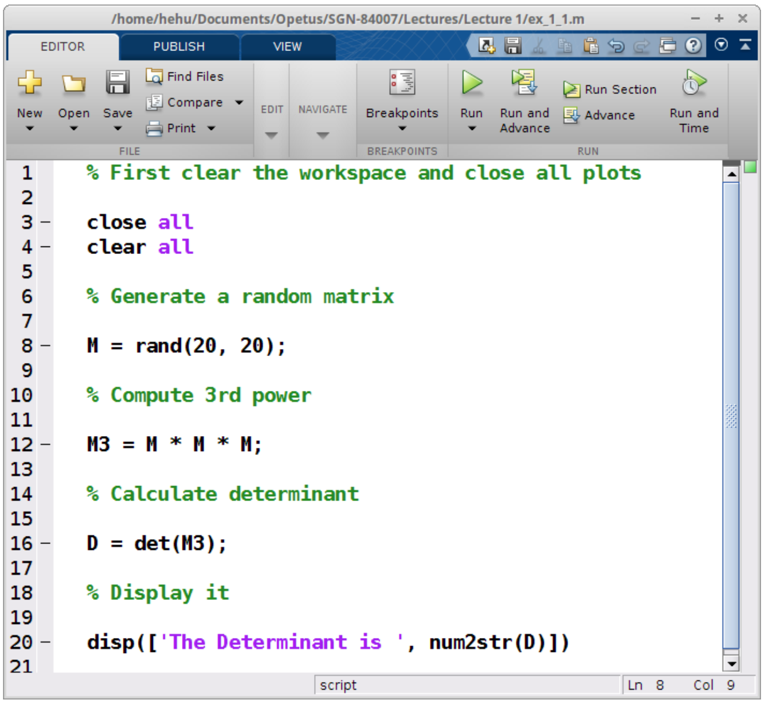
For programs longer than a few lines, it is not a good idea to write the code to the command line. Matlab editor is the IDE for this purpose:

```
>> edit myprogram.m
```

This opens an editor window for file `myprogram.m`.

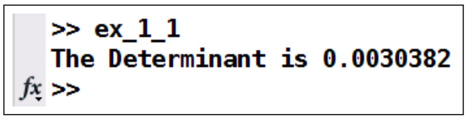
In the exercises, all solutions are written into a script file in the editor. This allows easier examination by the assistant; otherwise the command history is full of trial and error and the results cannot be repeated.

Example: Suppose the task is to generate a random 20×20 matrix, and to compute the determinant of its 3rd power. In this case, we would enter the following lines to the editor.



```
% First clear the workspace and close all plots

close all
clear all

% Generate a random matrix

M = rand(20, 20);

% Compute 3rd power

M3 = M * M * M;

% Calculate determinant

D = det(M3);

% Display it

disp(['The Determinant is ', num2str(D)])

```

The program runs conveniently by clicking the "Run" button at the top. Alternatively, use shortcut key $\boxed{\text{F5}}$. The output appears at the console (not the editor).



```
>> ex_1_1
The Determinant is 0.0030382
fx >>
```

For loop is used for iterating a block of code a fixed number of times. For example, the block below evaluates the expressions $(-2)^1, (-2)^2, (-2)^3, (-2)^4, (-2)^5$. For loops are slower than vector computations. The same result could be achieved by `a = (-2).^(1:5)`, which is **faster**.

```
a = zeros(1,5);

for k = 1:5
    a(k) = (-2)^k;
end
```

Looping over non-numeric arrays is most easily done via their indices.

```
languages = {'Matlab', 'C++', 'Python'};

for idx = 1:length(languages)
    message = sprintf('I love %s',...
        languages{idx});
    disp(message)
end
```

```
I love Matlab
I love C++
I love Python
```

The syntax of the if statement is unsurprising. No parentheses, colons or other separators are needed.

```
values = rand(1,100);

smallvals = [];
bigvals = [];
middlevals = [];

for v = values

    if v < 0.05
        smallvals(end+1) = v;
    elseif v > 0.95
        bigvals(end+1) = v;
    else
        middlevals(end+1) = v;
    end

end
```

```
3 small, 4 big and 93 middle values
```

Note the use of the more general for loop, where `v` loops over the values, not their indices (however, not recommended in Matlab). The same effect could be reached like this:

```
for i = 1:length(values)
    if values(i) < 0.05
      ...
```

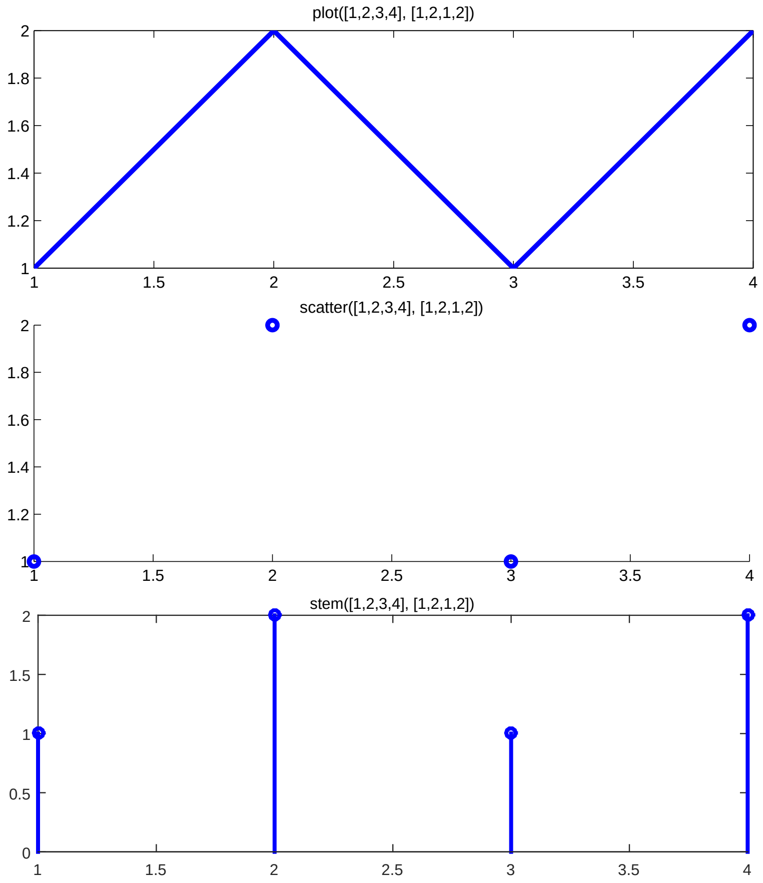
1.2 2D Plotting

All plotting commands have similar interface:

- y-coordinates: `plot(y)`, and
- x- and y-coordinates: `plot(x,y)`.

Most commonly used plotting commands include the following:

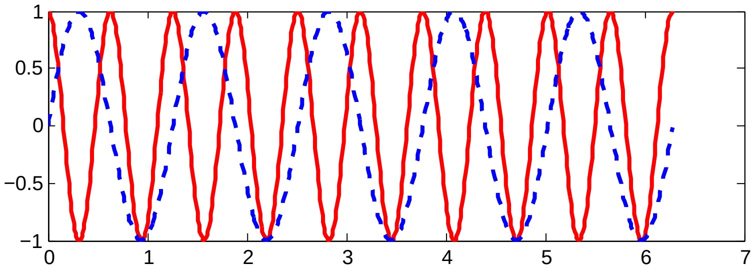
- `plot`: Draw a line plot joining individual data points. Typical for plotting continuous curves or long time series.
- `scatter`: Draw only the points. Same as `plot(x, y, 'o')`.
- `stem`: Draw points with stems. Typical for plotting discrete time series.



Often one wants to draw several functions into the same plot. There are two strategies:

- Enter more than one vector to the plot (stem, scatter) command.
- Use the `hold` command and the previous plot is not overwritten.

```
% Define the functions to plot
x = 0 : 0.01 : 2*pi;
y1 = cos(10 * x);
y2 = sin(5 * x);

% Plot with single command:
plot(x, y1, 'r-', x, y2, 'b--');

% Alternatively:
plot(x, y1, 'r-');
hold('on')
plot(x, y2, 'b--');
hold('off')
```



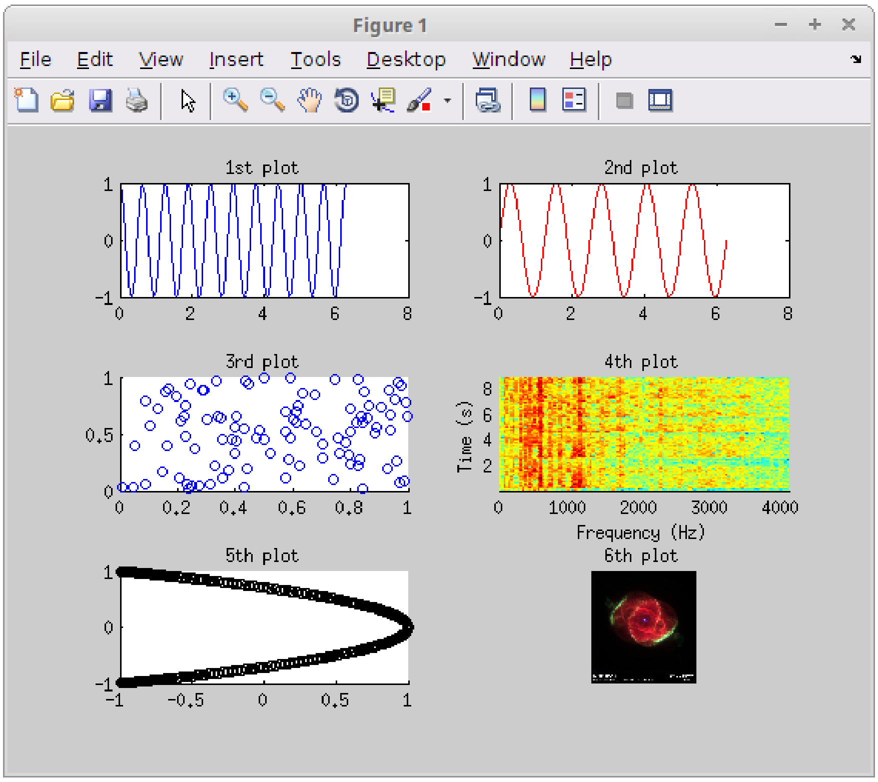
In order to separate the curves, we define styles as the third argument to each plot (e.g., 'b--'). There are many different plot style strings (`help plot`):

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

By default, there is one plot per figure. However, one may want to draw several axes on the same plot, and arrange plot sizes better. Command `subplot` defines many axes into the same window.

```
% 6 subplots arranged in a 3x2 grid
% The last argument defines where to plot

subplot(3, 2, 1);                % Plot #1
plot(x, y1);
title('1st plot');
subplot(3, 2, 2);                % Plot #2
plot(x, y2, 'r-');
title('2nd plot');
subplot(3, 2, 3);                % Plot #3
scatter(rand(100,1), rand(100,1));
title('3rd plot');
subplot(3, 2, 4);                % Plot #4
[X, Fs] = audioread('handel.ogg');
spectrogram(X, 512, 256, 256, Fs)
title('4th plot');
subplot(3, 2, 5);                % Plot #5
scatter(y1, y2, 'k');
title('5th plot');
subplot(3, 2, 6);                % Plot #6
img = imread('ngc6543a.jpg');
imshow(img);
title('6th plot');
```

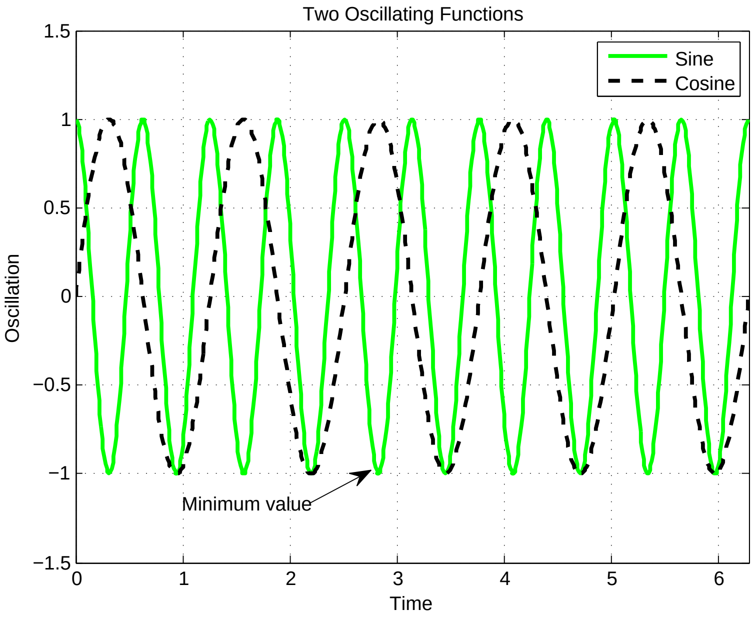
The result of this code is shown below.



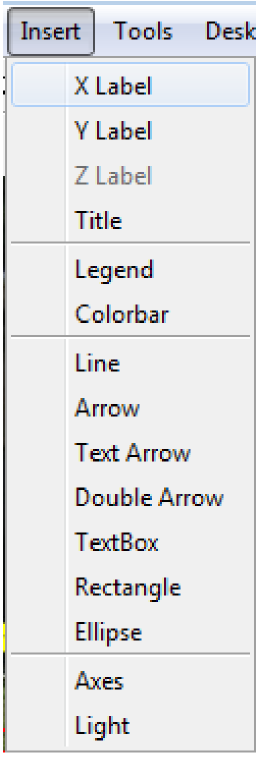
The subplots were marked with a title on top of each plot. There are many other annotation tools to add text, arrows, lines, legends, etc. on the plot:

- `title`: text above plot,
- `xlabel`: text for x-axis,
- `ylabel`: text for y-axis,
- `legend`: names for curves,
- `grid`: grid on and off, and
- `annotation`: arrows, lines, text, etc.

```
plot(x, y1, 'g-', x, y2, 'k--')
axis([0, 2*pi, -1.5, 1.5]);
legend({'Sine', 'Cosine'});
xlabel('Time');
ylabel('Oscillation');
title('Two Oscillating Functions');
annotation('textarrow', [0.4, 0.47], [0.2, 0.25], 'String', 'Minimum
    value');
grid('on');
```

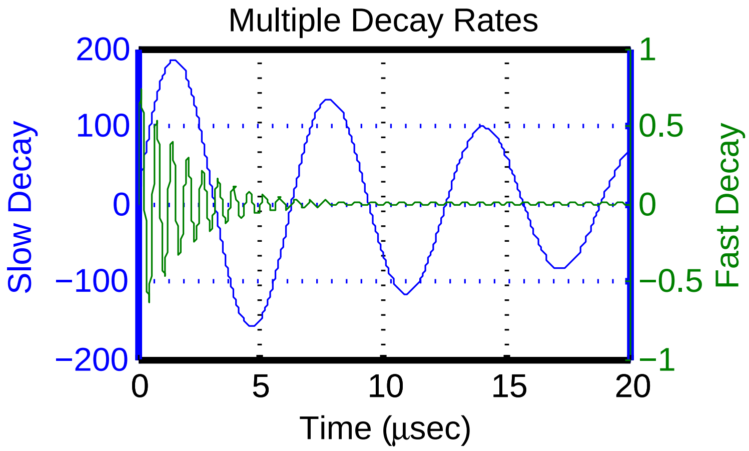


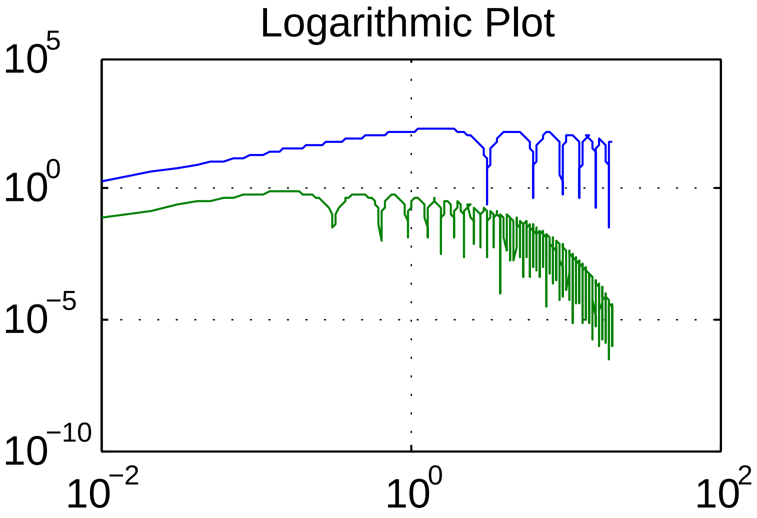
Note: Annotations can also be inserted interactively from figure window.

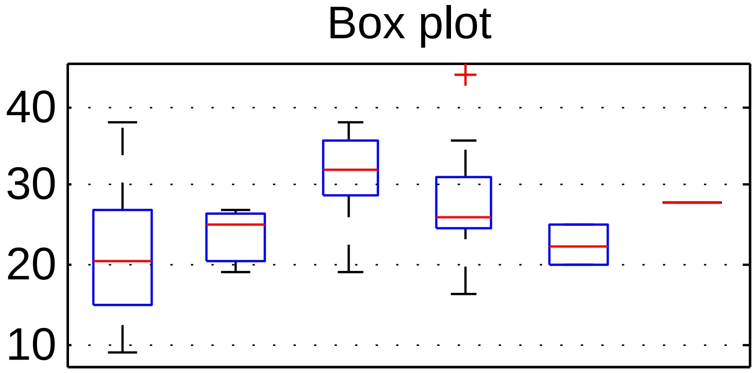


Other 2D visualization tools include the following:

- `plotyy`: Add two y-axes on the same plot.
- `semilogx, semilogy, loglog`: Logarithmic plots.
- `pie, histogram, bar`: Pie charts, etc.
- `polar, rose, compass`: Angular plots.
- `errorbar, boxplot`: Confidence intervals.







1.3 Data Import and Export

Matlab has its own file format for storing data and variables. However, not all applications produce Matlab compatible file formats. In such cases, a low level simple data format may be required as an intermediate step (such as csv). Matlab has several utilities for importing data from widely used formats. Matlab's own file format can easily store and recover the entire workspace. The default format is binary file, which may contain any Matlab data type including their names. When loaded, the values get their original names (this may in fact be confusing).

```
% Save the entire workspace

>> X = [1;2;3];
>> save data.mat % Save all variables
>> clear all     % Clear everything
>> load data.mat % Load all variables
>> disp(X)
     1
     2
     3
```

Variants of the save command allow saving in ascii format. However, the ascii file does not store variable names.

```
% Save data into txt format.
% Note: The variable names are not stored,
% so recovery may be difficult.

>> save data.txt -ascii
>> type data.txt

   1.0000000e+00
   2.0000000e+00
   3.0000000e+00
```

```
% Save only variables X and Y:

>> save data.mat X Y
```

```
>> X = [1,2,3;4,5,6];
>> csvwrite('data.csv', X) % Save to CSV
>> type data.csv

1,2,3
4,5,6
>> X2 = csvread('data.csv'); % Load CSV
```

```
>> type CarPrices.csv

Model;Year;Price
Ford;2009;9500
Peugeot;2008;8000
Toyota;2011;11000

>> X = csvread('CarPrices.csv');
Error using dlmread (line 138)
Mismatch between file and format string.
```

The CSV format (Comma Separated Values) is one of the most widely used generic data container formats. Commands `csvread` and `csvwrite` are typically enough for numeric data input and output. However, CSV stored from, e.g., Excel may contain non-numeric columns and rows. Such more complicated files can be read by `textscan`.

```
% Open the file for reading:
>> fid = fopen('CarPrices.csv', 'r');

% Scan the contents.
% Omit 1 line from beginning.
% Values are separated by ';'

>> C = textscan(fid, '%s%d%d',...
>>              'HeaderLines', 1,...
>>              'Delimiter', ';');

% The result is a 'cell',
% each item containing one column;

>> disp(C{1})
    'Ford'
    'Peugeot'
    'Toyota'

>> A = horzcat(C{2:3}); % Cell to matrix
>> disp(A)
        2009        9500
        2008        8000
        2011       11000

>> fclose(fid); % Close file
```

textscan allows specifying the data types at each row. This is done using a *format string* similar to printf in C. In the attached example, the format string %s%d%d states that each row contains a string (%s) and two integers (%d) separated by ';'. Other usual format specifiers include

- Floating point number: %f,
- 8-bit integer: %d8 (to save space).

```
>> [num,txt,raw] = xlsread('CarPrices.xlsx')

num =

        2009        9500
        2008        8000
        2011       11000


txt =
    'Model'       'Year'     'Price'
    'Ford'        ''         ''
    'Peugeot'     ''         ''
    'Toyota'      ''         ''


raw =

    'Model'       'Year'     'Price'
    'Ford'        [2009]     [ 9500]
    'Peugeot'     [2008]     [ 8000]
    'Toyota'      [2011]     [11000]
```

Excel files are well supported. Basic commands: `xlsread` and `xlswrite`. The xls reader returns three outputs:

- `num`: A matrix with all numerical values found.
- `txt`: A cell array with all textual values found.
- `raw`: A cell array with all values found.

```
>> image = imread('ngc6543a.jpg');
>> size(image)

ans =

   650   600     3
```

```
>> [audio, Fs] = audioread('handel.mp3');
>> size(audio)

ans =

   73113     1
```

```
>> movie = VideoReader('freqResponse.mp4');
>> videoFrames = read(movie);
>> size(videoFrames)

ans =

    900    1200     3     98

% Dimensions are:
% height, width, color channels, frames
```

Matlab is often used for image and audio processing. These can be conveniently loaded to Matlab:

- `imread` reads almost any image format.
- `VideoReader` reads movies in all formats that have a codec installed.
- `audioread` reads WAV, FLAC, MP3, MP4 and OGG audio formats.

Other supported formats include:

- HDF5 (Hierarchical Data Format): large datasets,
- NetCDF Files (Network Common Data Form): scientific data,
- CDF (Common Data Format): scientific data,
- XML (Extensible Markup Language): structured input and output,
- DICOM (Digital Imaging and Communications in Medicine): medical images,
- HDR (High Dynamic Range): images,
- Webcam: video from laptop camera, and
- Other devices: See Data Acquisition Toolbox.

1.4 Programming in Matlab

When writing any program code longer than a few dozen lines, you will need functions. Next we will study how functions are implemented in Matlab.

```
function [out1, out2] = myFunction(input1,...
                                   input2,...
                                   input3)
%
% Example definition of a function in Matlab.
%
% Usage:
%
% [out1, out2] = myFunction(in1, in2, in3)
%
% Function calculates the sum of the three
% inputs and the product of the three inputs
% and returns the results.

out1 = input1 + input2 + input3;
out2 = input1 * input2 * input3;
```

```
>> [a,b] = myFunction(5,6,7)

a =

    18

b =

   210
```

Matlab functions are written into individual *.m files. The file name has to match the function name, e.g., the attached example has to be in a file called myFunction.m. The function definition starts by keyword function and the header defines the inputs and outputs. In our example, there are 3 inputs and 2 outputs.

```
function [out1, out2] = myFunction(input1,...
                                   input2,...
                                   input3)
%
% Example definition of a function in Matlab.
%
% Usage:
%
% [out1, out2] = myFunction(in1, in2, in3)
%
% Function calculates the sum of the three
% inputs and the product of the three inputs
% and returns the results.

out1 = input1 + input2 + input3;
out2 = input1 * input2 * input3;
```

```
>> help myFunction
  Example definition of a function in Matlab.

  Usage:

  [out1, out2] = myFunction(in1, in2, in3)

  Function calculates the sum of the three
  inputs and the product of the three inputs
  and returns the results.
```

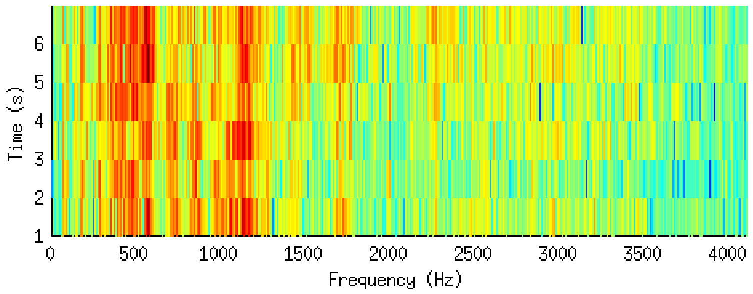
Below the header is the description of using the function. This message is shown when help myFunction is called. Below the help message, the actual computation is done. The return values are determined based on the header: Whatever values are in out1 and out2 at exit, will be automatically returned.

```
function result = myFunction(in1, in2, varargin)
% Usage:
%
% result = myFunction(in1, in2, [in3])
%
% Function calculates the sum of the two first
% inputs optionally multiplies the sum with
% the third.

if nargin == 3
    in3 = varargin{1};
    result = (in1 + in2) * in3;
else
    result = in1 + in2;
end
```

```
>> res = myFunction(5,6,7)

res =
    77

>> res = myFunction(5,6)

res =
    11
```

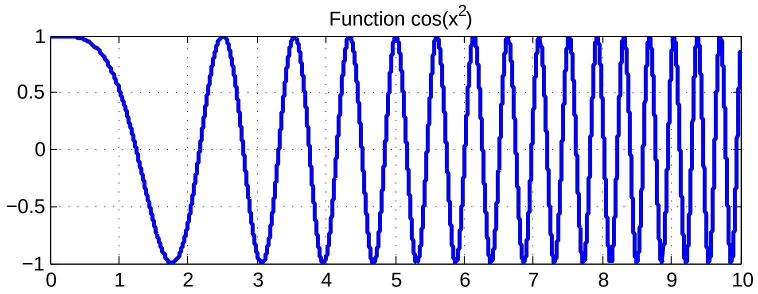
It is possible to overload the function such that it can accept variable number of arguments. This is done with the keyword `varargin` as the last item of the definition. After this, the variables `nargin` contains the number of arguments when called. `varargin` will be a cell array of the extra arguments.

```
% I want to calculate the spectrogram
% of a signal with:
%
%        spectrogram(X,WINDOW,NOVERLAP,NFFT,Fs)
%
% However, I only want to specify X and Fs.

[x, Fs] = audioread('handel.ogg');
spectrogram(x, [], [], [], Fs);
```



The varargin structure allows user to omit some optional arguments. Sometimes you may want to set the last argument and omit some middle ones. In this case, the empty matrix can be given in place of omitted arguments. *Example:* Search for the Root of a Function

```
% Define a function handle for our target:

>> func = @(x) cos(x.^2);

% Now we can call the function as usual.
% Let's plot it.

>> x = 0:0.01:10;
>> plot(x, func(x));
```



Let's implement a longer function, which searches for the root of a given function. We will use *bisection search* algorithm, which starts at two sides of the root and halves the range at each iteration. Question: How to pass the target function to our program? Answer: We use a function handle defined by '@'. The bisection algorithm works as follows.
Iterate until finished:

1. Assume x_1 and x_2 are at opposite sides of the root, such that $f(x_1)$ and $f(x_2)$ have different signs.

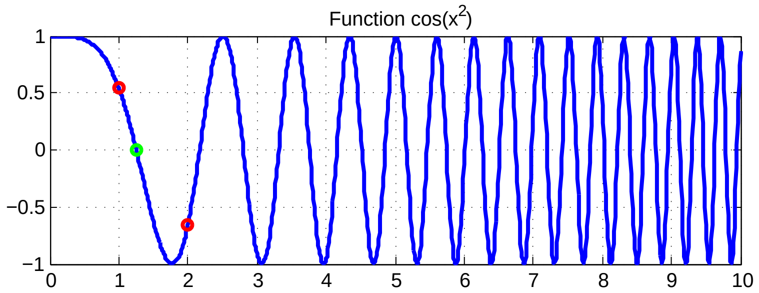
2. Check the sign at the center: $x_0 = (x_1 + x_2)/2$:

   - If $f(x_1)$ and $f(x_0)$ have different signs, then root is in $[x_1, x_0]$. Set $x_2 = x_0$.
   - If $f(x_2)$ and $f(x_0)$ have different signs, then root is in $[x_0, x_2]$. Set $x_1 = x_0$.

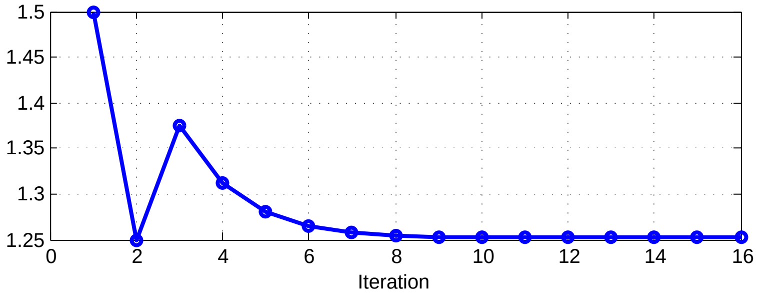
3. Go to step 1.

```
function x = searchRoot(f, x1, x2)

% Find the root of function f using the bisection method.
%
% Usage:
%
% x = searchRoot(f, x1, x2)
%
% x1 and x2 have to be such that
%    f(x1)*f(x2) < 0

half = (x1+x2) / 2;

% Assert that f has different sign in x1 and x2

if sign(f(x1)) == sign(f(x2))
    x = half;
    return
end
```

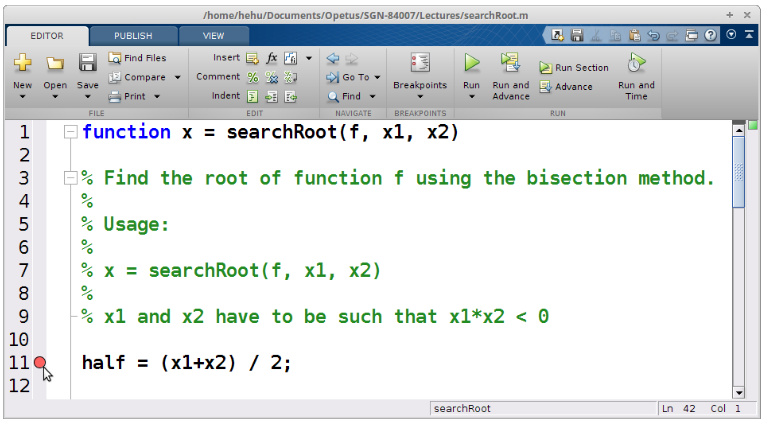
```
% Check if we're already close enough to the root

if abs(f(half)) < 1e-5
    x = half;
    return
end

% Make sure x1 < x2

if x1 > x2
    tmp = x1;
    x1 = x2;
    x2 = tmp;
end

% Otherwise, check the sign at half and recurse

if sign(f(half)) == sign(f(x1))
    x = searchRoot(f, half, x2);
else
    x = searchRoot(f, x1, half);
end
```



```
func = @(x) cos(x.^2);
x0 = searchRoot(func, 1, 2);
```



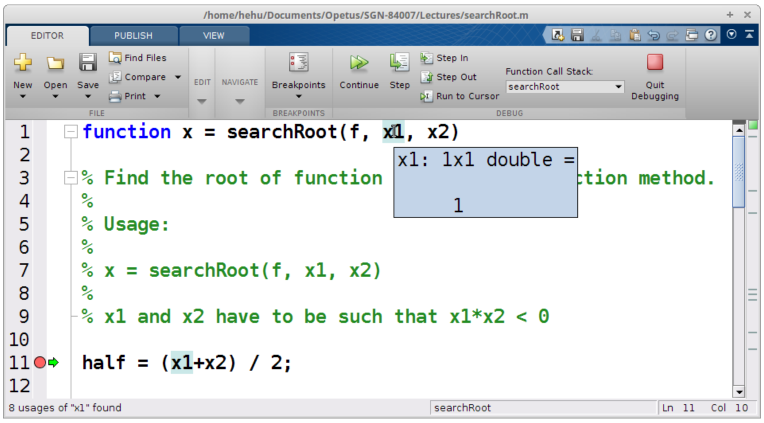
For our function, it seems that locations $x_1 = 1$ and $x_2 = 2$ are good starting points. The function iterates for 16 rounds and reaches $x_0 = 1.2533$ for which $f(x0) = 0.00000747$. Debugging helps in finding bugs in the code. Matlab editor allows setting breakpoints, where execution stops. Breakpoint is set by pressing F12 when on the active line.



```
function x = searchRoot(f, x1, x2)

% Find the root of function f using the bisection method.
%
% Usage:
%
% x = searchRoot(f, x1, x2)
%
% x1 and x2 have to be such that x1*x2 < 0

half = (x1+x2) / 2;

```

When executed, the program flow will stop at breakpoint. Moving mouse over variables will show their values. The Matlab prompt is also available. Shortcuts: F5 = run forward,

$\boxed{\text{F10}}$ = step to next line, $\boxed{\text{F11}}$ = enter inside function, and $\boxed{\text{shift}}+\boxed{\text{F5}}$ = stop.



```
function x = searchRoot(f, x1, x2)

% Find the root of function               tion method.
%
% Usage:
%
% x = searchRoot(f, x1, x2)
%
% x1 and x2 have to be such that x1*x2 < 0

half = (x1+x2) / 2;

```

```
x1: 1x1 double =

     1
```

Chapter 2

Digital Signal Processing

Digital signal processing has become one of the important fields in the modern technology. It closely supports at least telecommunication technology, measurement technology and information technology. The birth of digital signal processing (DSP) is considered to date in the 1960s and 1970s, when the computers started to be available. After those days it has been successfully applied in numerous areas; from medical PET imaging to the CD player and GSM phone.

The applications of DSP are numerous, however the most important basic methods have remained the same over the years. In this course, we will discuss

- the most important basic concepts,
- some of the most important methods, and
- example applications.

After covering the basics of linear systems, we will familiarize ourselves with a typical signal processing problem: how to remove certain frequencies from a given signal.

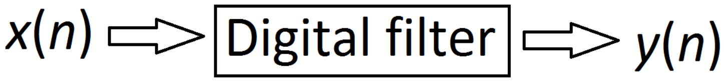
Future signal processing courses will then give you a closer look at signal processing methods and applications.

2.1 What Does Signal Processing Mean?

A typical DSP application includes the following steps:

1. The so-called A/D (analog-to-digital) converter converts the received (continuous-time) analog signal to digital and discrete-time.
2. Then the discrete-time digital signal is modified by some system (e.g., computer). This step is called *filtering.* The purpose of the filtering is to convert the input signal to a form which is more useful for the application. This may mean for example:
    - Removal of the noise in the signal so that the actual signal remains as good as possible.
    - Separation of interesting features from the signal.

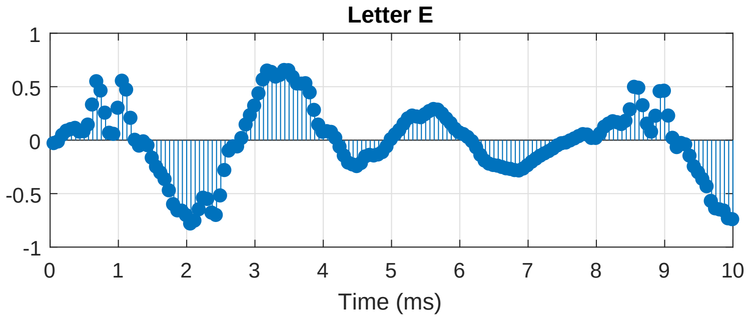
Traditionally, the filters have been linear due to easier implementation and analysis, but nonlinear filters have also been studied.



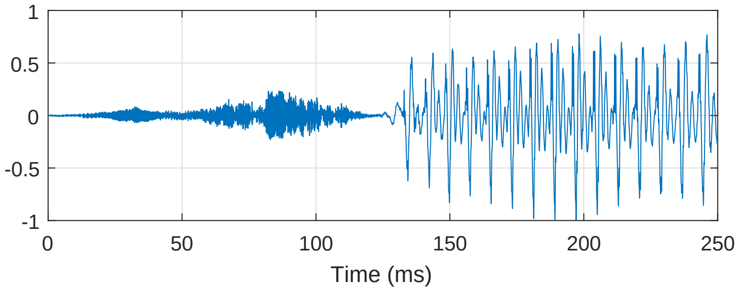
3. After filtering, the signal is converted back to analog using D/A converter.

In this course, we will mainly focus on step 2: filter design.

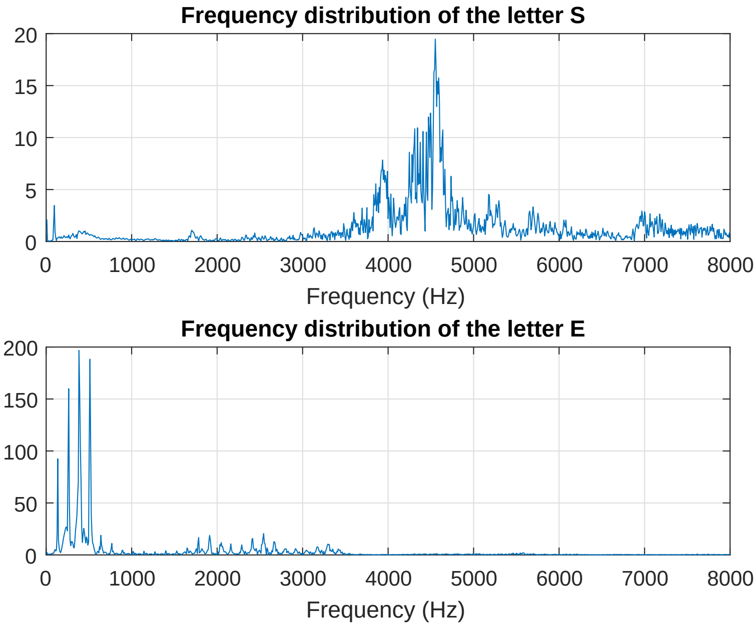
The signal to be processed may include, for example, sound, speech, pulse, brain curve, earthquake, stock exchange or any measurable time series. The figure below shows a 10 ms sample of a speech signal. The microphone converts the small fluctuations of the air pressure into electronic format, from which the computer converts them into digital format by storing the instantaneous numeric values of the voltage every 1/16000 seconds. In this case, *sampling rate* is 16000 hertz.



The following figure shows a longer sample of the same signal. This part of the signal contains the first two letters of the Finnish word ”seitsemän”. The letter S is located approximately in 125 ms from the start, and the next 125 ms contain the letter E. The difference between the consonant S and the vowel E can be observed well: there is a clear up-down vibration pattern for the voiced vowel, while for the consonant there is not such a clear vibrational pattern.



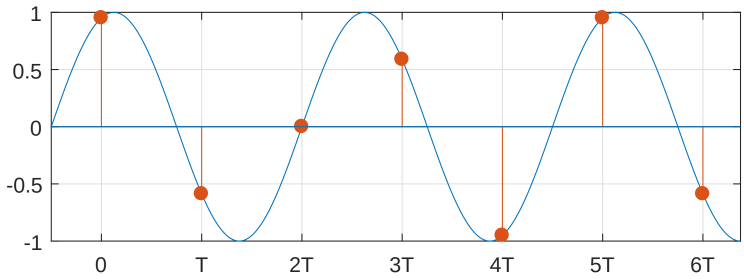
For many different types of signals, it is natural to think that they are composed of individual frequencies (individual sinusoidal signals in an appropriate ratio). For example, audio signals are easiest to understand and analyze using their frequency distribution. As we will later see, the frequency distribution can be calculated using *Fourier transform*. The frequency distributions of the letters S and E are shown in the figures below. The figures clearly show that the frequencies the letter S contains spread over a fairly wide area including also high frequencies, whereas for the letter E there are four separate low frequencies (the highest peaks) dominating the distribution.



2.2 Sampling Theorem

In the previous example, the sampling rate was 16000 hertz, i.e. 16 kHz. The higher the sampling rate, the more accurately the sampled signal represents the original analog signal. However, higher sampling rate requires more storage space, so the frequency should not be raised too high. How then to figure out the adequate sampling rate for a given signal? This question is answered by *the sampling theorem*.

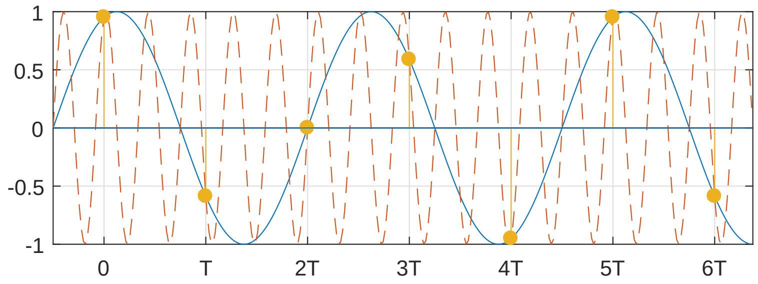
When a continuous-time signal is sampled, the samples are taken at times 0, T, $2T$, $3T, \ldots$ and the values of the signal are stored only at these instants of time (red circles in the figure below).



If we denote the continuous-time signal by $x_c(t)$, where $t \in \mathbf{R}$, then the result of the sampling will be a sequence $x(n)$ for which $x(n) = x_c(nT)$ $(n = 0, 1, 2, \ldots)$. The constant T is *sampling period* and indicates the number of seconds between the two consecutive samples.

Often the same thing is expressed by telling how many samples are taken per second. The name of this quantity is *sampling rate* or *sampling frequency* and it is the inverse of the constant T, $F_s = 1/T$. If the sampling rate is too small (and thus the sampling period T is too large) occurs *aliasing*.

Aliasing is illustrated in the following figure. The two sinusoidal signals in the figure have the same sample values due to the sampling rate being too low for the high-frequency signal (dashed line). When converting the sampled signal back to analog form, the result is the low-frequency sinusoidal signal (solid line).

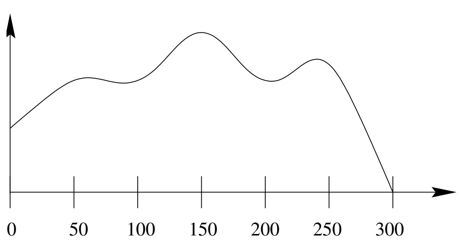


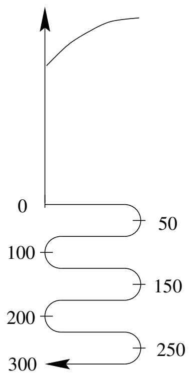
When considering sufficient sampling frequency for the sinusoidal signal of the previous example, it seems that two samples per cycle might suffice. In this case, we would store the maximum and minimum value of the signal and using these values we could then interpolate the other values. Naturally, any higher frequency is equally good. *The sampling theorem* tells us that our guess is correct:

> *A continuous-time signal can be exactly reproduced if it is sampled at a frequency F_s, where F_s is at least twice the maximum frequency in the signal.*

If the previous condition is not valid, the frequencies above $F_s/2$ must be cut off by some analog system to prevent aliasing. The frequency $F_s/2$ is called Nyquist frequency or folding frequency.

Therefore, a signal with the maximum frequency of 300 kHz requires a system with the sampling rate of at least 600 kHz. Signal spectrum indicates how much the signal contains each frequency. Below is the spectrum of a continuous signal having frequencies up to 300 kHz (left). If the signal is sampled at 100 kHz, then the frequencies above 50 kHz are summed with the lower frequencies according to the folded x-axis (right).





In the sampled result, the spectrum is sum of frequency components:

- 25 kHz is sum of: 25, 75, 125, 175, 225 and 275 kHz.
- 10 kHz is sum of: 10, 90, 110, 190, 210 and 290 kHz.

The original signal can no longer be reproduced from the sum of these six frequency components.

The best result at the 100 kHz sampling rate is obtained by removing the frequencies exceeding 50 kHz before sampling. Then, although the frequencies 50 kHz – 300 kHz are lost, at least the frequencies 0 kHz – 50 kHz are stored in their original format.

2.3 Pros and Cons of DSP

Analog filters have been studied in electronics for a long time. They are assembled from electronic components and typically pick up certain frequencies from the signal and remove the others. This raises the question: Why should the same be done digitally?

It is easy to make digital filters accurate, and their properties remain the same throughout the entire operating time. The properties of digital filters are determined by their coefficients, and computer program coefficients do not change, for example, over time or when the temperature fluctuates. Accuracy can also be easily improved by adding more computing power and computational accuracy. Of course, analog systems can be made to be equally accurate and to retain their properties, but that requires the use of more expensive and higher quality components. It is often said that the digital CD player brought HIFI quality available to the ordinary consumer when with analog equipment it was only available to the wealthy.

Digital filters have several good theoretical properties. They can, for example, be used to implement a completely *linear phase* filter, which is impossible to realize using analog filters. Linear phase means that all the frequencies the signal contains are delayed equal amount and the topic will discussed in more detail later during the course. Additionally, digital filters work just like computer programs, so complicated structures can be added to them that are impossible to implement by analog systems.

The main reason for the use of digital filters instead of analog components is money: the same signal processor can be used in a variety of applications thus it can be produced in larger batches and this brings processor prices down. On the other hand, the companies using processors implement their own product as a software instead of a physical device. This makes product duplication easy and the same product can be sold several times—just like a computer software.

However, very simple systems that do not need high precision are easiest to implement using analog components. The digital system always needs A/D and D/A converters as well as a processor. If the goal is only to split the car stereo loudspeaker signal into two different frequency bands, it is not worth building a digital system for this purpose.

2.4 Applications

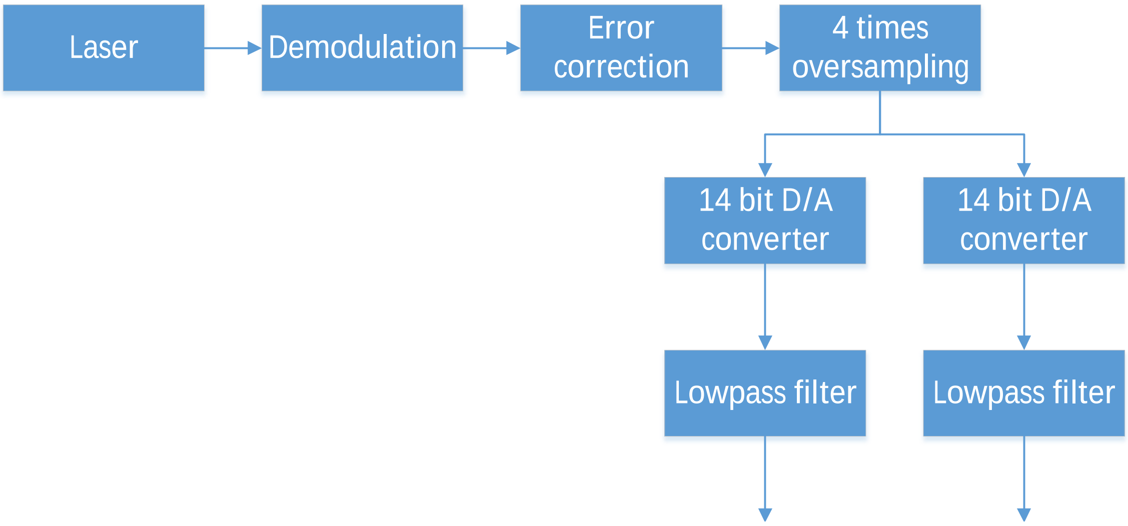
Next, we look at some of the typical applications rather superficially.

Compression: The compression algorithms aim at removing the *redundancy* between the signal samples and to reduce the number of required bits. There are many different compression methods. The method may be designed to efficiently compress, for example, an audio signal, digital image or video, or even 3D tomographic medical images. The methods utilize the characteristics of each application.

Compression Highlight: Spring 2017 international effort to develop a new standard for plenoptic image compression was launched by the JPEG Committee (which oversees the development of related standards). The proposal submitted by a research group in the Laboratory of Signal Processing, TUT, was selected the best.

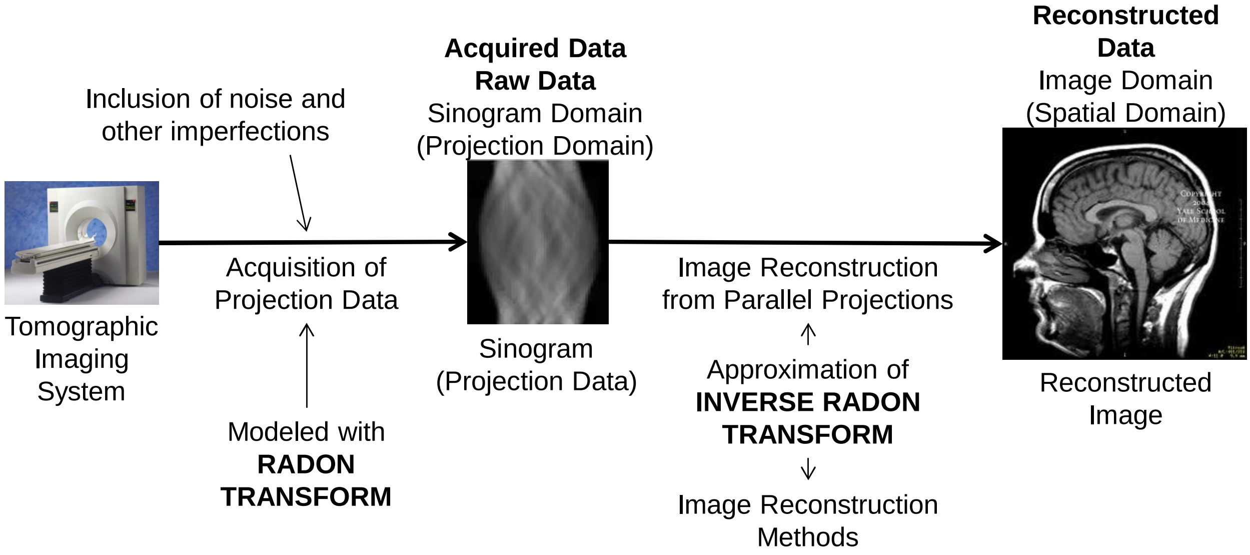
Echo Cancellation: As the telephone signals travel bidirectionally annoying echo may become a problem for the user. Each spoken word returns with a small time delay and produces an echo effect. In particular this is problematic when using a speakerphone, but also on standard phones. This problem can be effectively eliminated by adaptive signal processing methods.

CD player: Below is a simplified diagram of a typical CD player containing a number of items that use DSP.



Speech recognition: In the early 2000s, speech recognition was done by traditional approaches such as Hidden Markov Models combined with feedforward artificial neural networks. The recent advances in deep learning and big data have been utilized also in speech recognition especially in the form of a deep learning method called Long short-term memory (LSTM), a recurrent neural network (RNN). In 2015, Google speech recognition reportedly experienced a dramatic performance jump of 49% through Connectionist Temporal Classification (CTC) -trained LSTM (available through Google Voice).

Tomographic medical imaging: Medical imaging is a discipline within the medical field which involves the use of technology to take functional or anatomical images of the inside of the living body. The goal of medical imaging is to provide an image of the inside of the body in a way which is as non-invasive as possible. X-Ray CT, MRI, PET, SPECT and Electron Tomography are some of the tomographic imaging modalities. The big picture of tomographic image reconstruction is illustrated below.

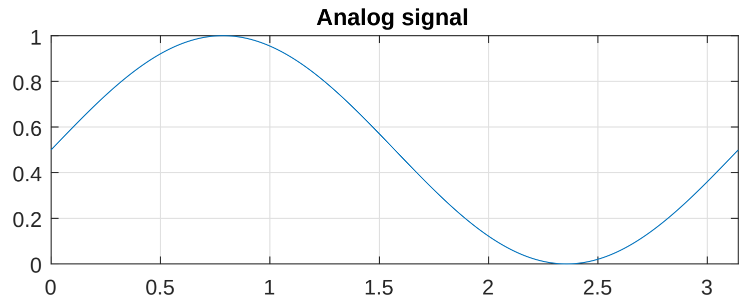

Inclusion of noise and other imperfections
Acquired Data
Raw Data
Sinogram Domain
(Projection Domain)
Reconstructed Data
Image Domain
(Spatial Domain)
Tomographic Imaging System
Acquisition of Projection Data
Modeled with **RADON TRANSFORM**
Sinogram
(Projection Data)
Image Reconstruction from Parallel Projections
Approximation of **INVERSE RADON TRANSFORM**
Image Reconstruction Methods
Reconstructed Image

Chapter 3

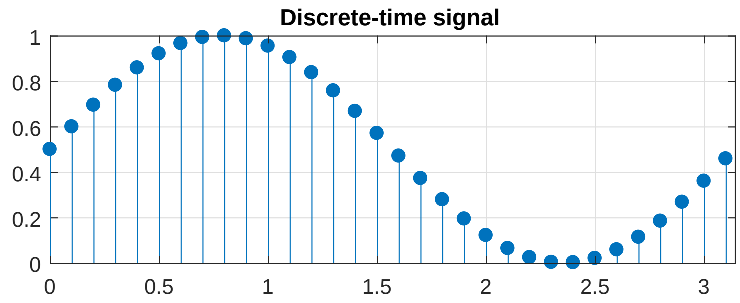
Signals and Systems

We have already seen examples of digital signals. Now we will define the concepts more precisely.

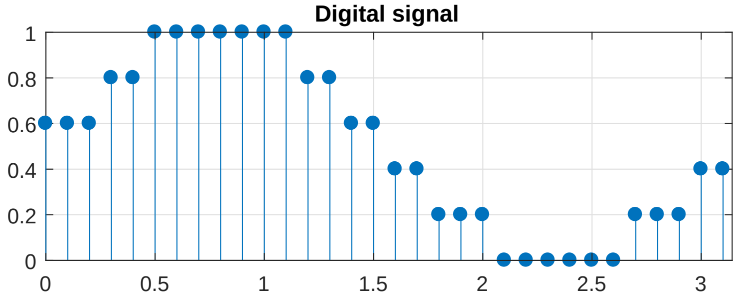
Analog signal is defined at each time point and can get an infinite number of different values (e.g., between $[0, 1]$, as shown below).



Discrete-Time Signal only gets values at certain times.



Digital Signal only gets a finite number of different values and only at certain times.

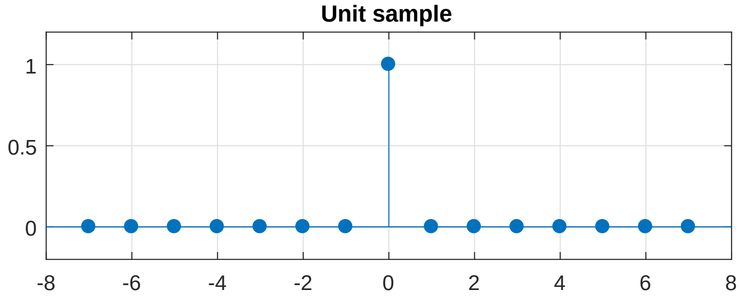


To simplify the mathematics, sequences of numbers are often used as a model of discrete-time signal. Unlike real-world signal, the sequence is infinitely long, but this is not a problem in terms of modeling.

3.1 Some Signals

Unit sample or *impulse* $\delta(n)$ is defined as follows:

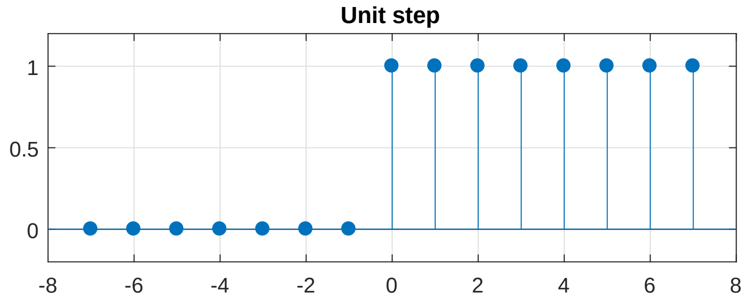
$$\delta(n) = \begin{cases} 1, & \text{when } n = 0, \\ 0, & \text{when } n \neq 0. \end{cases}$$



Matlab: `delta = [zeros(1,7),1,zeros(1,7)];`

Unit step $u(n)$ is defined as follows:

$$u(n) = \begin{cases} 1, & \text{when } n \geq 0, \\ 0, & \text{when } n < 0. \end{cases}$$



Matlab: `u = [zeros(1,7),ones(1,8)];`

These sequences can be represented using each other as follows:

$$\delta(n) = u(n) - u(n-1)$$

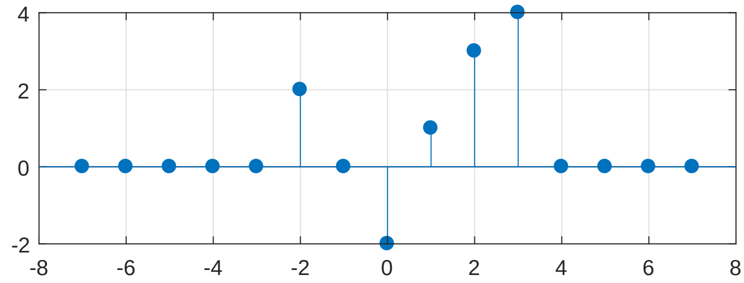
and

$$u(n) = \sum_{m=0}^{\infty} \delta(n-m) = \sum_{k=-\infty}^{n} \delta(k).$$

Any sequence of numbers can be represented by the shifted and scaled unit samples as follows:

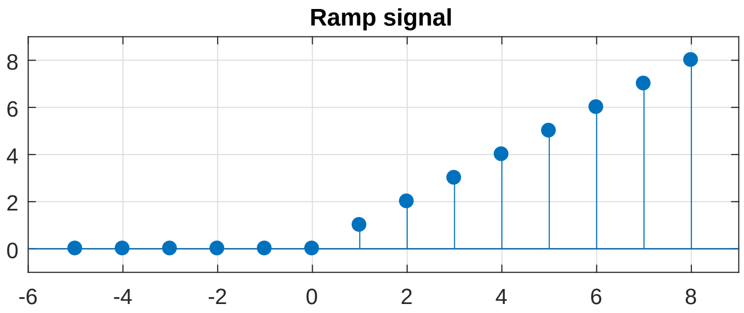
$$x(n) = \sum_{k=-\infty}^{\infty} x(k)\delta(n-k).$$

This representation is a special case of *convolution*, which is discussed soon in more detail. For example, the signal in the figure below may be represented in the form: $x(n) = 2\delta(n+2) - 2\delta(n) + \delta(n-1) + 3\delta(n-2) + 4\delta(n-3)$.



Ramp signal is defined as follows:

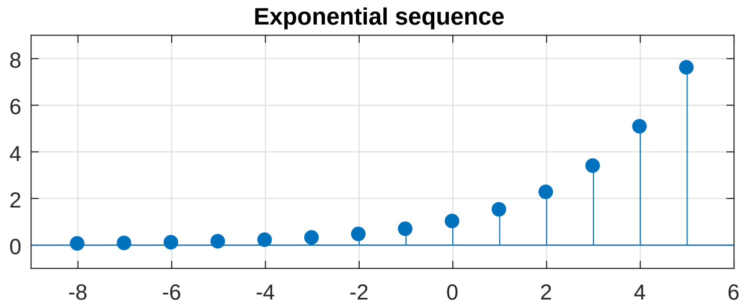
$$r(n) = nu(n) = \begin{cases} n, & \text{when } n \geq 0, \\ 0, & \text{when } n < 0. \end{cases}$$



Matlab: `r = [zeros(1,7),0:7];`

Exponential sequence is defined as follows:

$$x(n) = A\alpha^n.$$



Matlab: `x = -7:7; e = 1.5.^x;`

3.1.1 Properties of the Sequences

Discrete-time signal is *periodic* if there exists $N \in \mathbb{N}$ such that

$$x(n) = x(n+N)$$

for every value of the index n. The number N is called the *period length*.

We will often use *angular frequency* ω (measured in radians per second) instead of the ordinary frequency f (measured in Hz). The connection between them is $\omega = 2\pi f$.

Sinusoidal sequence $x(n) = \sin(\omega n)$ is periodic. It has period length

$$N = \frac{2\pi}{\omega} = \frac{1}{f}.$$

However, a sequence is in the strict sense of the definition periodic only if the resulting N is an integer.

If $\mathcal{S}$ is a set of all signals, then the mapping $\mathbf{F} : \mathcal{S} \mapsto \mathcal{S}$ is called a discrete system or a filter. The argument of $\mathbf{F}$ is called *input*, and the resulting sequence is called *output*.

For example, the equation $y(n) = x(n-10)$ defines a filter that delays the signal 10 steps. Another example calculates the average:

$$y(n) = \frac{1}{5}(x(n) + x(n-1) + x(n-2) + x(n-3) + x(n-4))$$

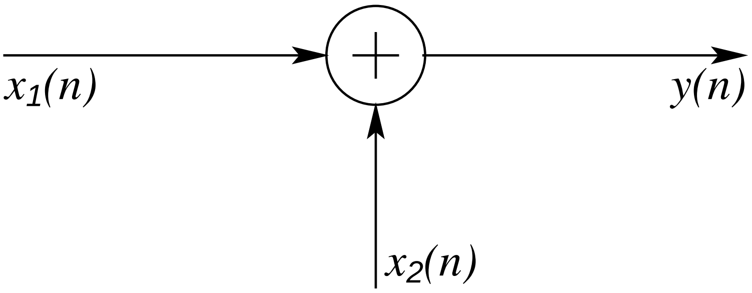
or more generally:

$$y(n) = \frac{1}{2K+1}\sum_{k=0}^{2K} x(n-k).$$

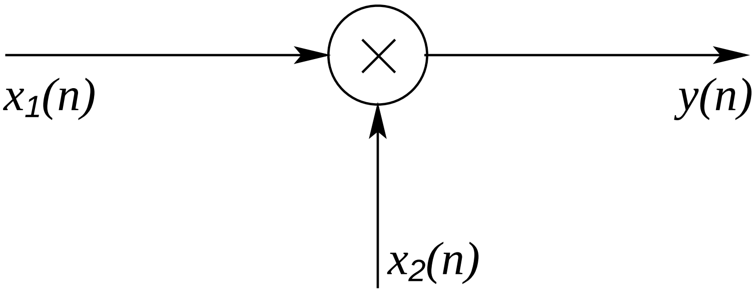
3.1.2 Basic operations for sequences

The basic operations for the sequences (signals) are shown next. Also, the symbols used for operations in block diagrams are included.

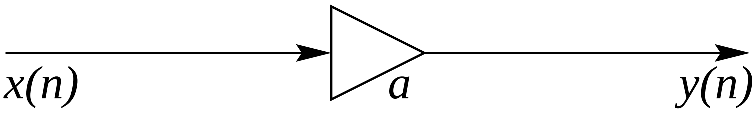
Addition: $y(n) = x_1(n) + x_2(n)$



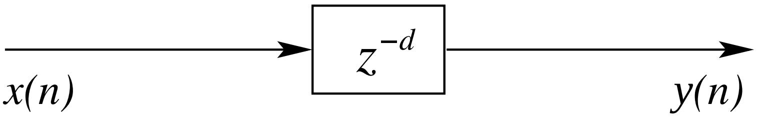
Multiplication: $y(n) = x_1(n) \cdot x_2(n)$



Scalar multiplication: $y(n) = ax(n)$



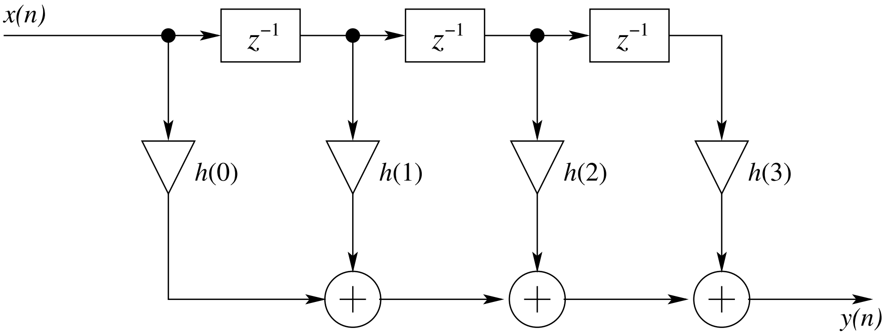
Delay: $y(n) = x(n-d)$



These operations can be combined and thus more complex systems can be created. One example is the system

$$y(n) = \sum_{k=0}^{3} h(k)x(n-k),$$

whose block diagram is shown below.



3.2 Properties of Discrete Systems

According to the above definition, all operations within the set of signals are filters. So there is a multitude of filters, and most of them have no practical value. We will next go through some of the properties that the filters may have. This allows filters to be classified according to the intended application.

3.2.1 Memorylessness

System is *memoryless* if its output $y(n)$ depends only on the input at that same time $x(n)$.

For example $y(n) = e^{x(n)}$ and $y(n) = x(n)^2$ are memoryless, but $y(n) = x(n-10)$ is not.

3.2.2 Linearity

The system $\mathbf{F}(\cdot)$ is *linear* if

$$\mathbf{F}[ax_1(n) + bx_2(n)] = a\mathbf{F}[x_1(n)] + b\mathbf{F}[x_2(n)], \tag{3.1}$$

for all signals $x_1(n), x_2(n)$ and all scalars a, b.

Verbally the same thing can be expressed, for example, as follows:

- The sum of the outputs is output of the sum.
- The output of input $ax(n)$ is obtained by multiplying the output $\mathbf{F}[x(n)]$ of input $x(n)$ by the scalar a.

On the other hand, Eq. (3.1) also means that:

- Scalar multiplication can be performed before or after the filtering.

- The addition may be performed before or after the filtering.

The linearity of a system is shown by showing that the left and right sides of the Eq. (3.1) are equal regardless of the coefficients a and b and the signals $x_1(n)$ and $x_2(n)$.

For example, the system

$$\mathbf{F}[x(n)] = \frac{1}{2K+1}\sum_{k=0}^{2K} x(n-k)$$

is linear. It can be shown as follows:

Let a and b be any real numbers and $x_1(n)$ and $x_2(n)$ in any two sequences. Then

$$\begin{aligned}\mathbf{F}[ax_1(n)+bx_2(n)] &= \frac{1}{2K+1}\sum_{k=0}^{2K}(ax_1(n-k)+bx_2(n-k))\\ &= a\frac{1}{2K+1}\sum_{k=0}^{2K}x_1(n-k)+b\frac{1}{2K+1}\sum_{k=0}^{2K}x_2(n-k)\\ &= a\mathbf{F}[x_1(n)]+b\mathbf{F}[x_2(n)].\end{aligned}$$

Thus, the system is linear.

Generally, it is easier to show non-linearity: it is enough to find such coefficients and signals that the left and right sides are different for at least one index n.

The system

$$\mathbf{F}[x(n)] = x(n)^2$$

is not linear, since Eq. (3.1) is not valid for example with the following values: $x_1(n) = u(n)$, $x_2(n)$ is zero signal, $a = 2$, $b = 0$ and $n = 0$. Then

$$\begin{aligned}\text{left-hand side} &= \mathbf{F}[ax_1(n)+bx_2(n)] = \mathbf{F}[2\cdot 1] = 2^2 = 4,\\ \text{right-hand side} &= a\mathbf{F}[x_1(n)]+b\mathbf{F}[x_2(n)] = 2\cdot 1^2 = 2.\end{aligned}$$

3.2.3 Shift-invariance

Denote $y(n) = \mathbf{F}[x(n)]$. System $\mathbf{F}(\cdot)$ is *shift-invariant* or *time-invariant* if

$$y(n-k) = \mathbf{F}[x(n-k)]$$

whenever $k \in \mathbb{Z}$. This means that the system is not dependent on time, and the shift can be made before or after filtering.

The following diagram illustrates the shift-invariance.

$$\begin{tikzcd}
x(n) \arrow[r, "\text{shift}"] \arrow[d, "\text{filtering}"'] & x(n\text{-}k) \arrow[d, "\text{filtering}"] \\
F[x(n)] \arrow[r, "\text{shift}"] & \textcircled{?}
\end{tikzcd}$$

If the result is the same for both of the above paths, the system is shift-invariant. If instead different signals are obtained, the system is not shift-invariant.

In practice, all reasonable systems are shift-invariant, since otherwise the system would work differently at different times of the day.

3.2.4 Causality

System $\mathbf{F}(\cdot)$ is said to be *causal*, if the output $y(n)$ depends only on the input values $x(n), x(n-1), x(n-2), \ldots$ and not on the values $x(n+1), x(n+2), x(n+3), \ldots$ Causality thus means that the system does not need to know in advance any future values of the input.

Consider the system

$$\mathbf{F}[x(n)] = \frac{1}{5}(x(n) + x(n-1) + x(n-2) + x(n-3) + x(n-4)).$$

To calculate the output in n, only the input values $x(n), x(n-1), x(n-2), x(n-3)$ and $x(n-4)$ are needed and thus the system is causal.

If the system under consideration is

$$\mathbf{F}[x(n)] = \frac{1}{5}(x(n+2) + x(n+1) + x(n) + x(n-1) + x(n-2)),$$

the values $x(n+2)$ and $x(n+1)$ are required to calculate the output and so the system is not causal.

3.2.5 Stability

The sequence of numbers $x(n)$ is said to be *bounded* if there exists an upper bound $M \in \mathbb{R}$ such that

$$|x(n)| \leq M,$$

given any index n. The discrete system $\mathbf{F}(\cdot)$ is *stable* if every bounded input causes bounded output (BIBO). In other words, from the condition

$$\exists M_1 \in \mathbb{R} : \forall n \in \mathbb{Z} : |x(n)| \leq M_1$$

follows the condition

$$\exists M_2 \in \mathbb{R} : \forall n \in \mathbb{Z} : |y(n)| \leq M_2,$$

where $y(n) = \mathbf{F}[x(n)]$.

Next we consider the system

$$y(n) = \mathbf{F}[x(n)] = \frac{1}{5}(x(n) + x(n-1) + x(n-2) + x(n-3) + x(n-4)),$$

and prove that it is stable. Input $x(n)$ is assumed to be bounded. Then there exists a real number M_1 such that $|x(n)| \leq M_1$ given any index n. To prove stability, we must show that there exists $M_2 \in \mathbb{R}$ such that $|y(n)| \leq M_2$ for all $n \in \mathbb{Z}$. It is easy to prove that the average of five inputs is always between the largest and the smallest input:

$$-M_1 \leq \min_{n-4 \leq k \leq n} x(k) \leq y(n) \leq \max_{n-4 \leq k \leq n} x(k) \leq M_1.$$

Consequently, the response $y(n)$ is bounded for all $n \in \mathbb{Z}$ and so the system $\mathbf{F}$ is stable.

The system $y(n) = \mathbf{F}[x(n)] = nx(n)$ is unstable since, e.g., $x(n) = u(n)$ (which is bounded) outputs

$$y(n) = \begin{cases} 0, & \text{when } n < 0, \\ n, & \text{when } n \geq 0, \end{cases}$$

which clearly is not bounded.

Also the system $y(n) = 1.1y(n-1) + x(n)$ is not stable because for the input $u(n)$ the output grows without bound. Later we will see that the stability analysis of these types of systems can be done simply by examining the so-called pole-zero plot.

3.3 Linear Time-Invariant (LTI) Systems

The design and implementation of linear and time-invariant (LTI) systems is straightforward and they can be used to implement frequency-modifying filters. The basic properties of these systems are linearity:

1. *Scalar multiplication can be done before or after the filtering.*

2. *Addition can be done before or after the filtering.*

and shift-invariance:

3. *Shift can be done before or after the filtering.*

Thus, in the case of LTI systems it makes no difference whether scalar multiplication, addition or shift is done before or after the filtering.

3.3.1 LTI: Impulse Response and Convolution

When the above three LTI properties are valid, a convenient representation for the system is *convolution*, where all system properties are determined via *impulse response* (*unit sample response*). Impulse response is defined as follows:

Let $\mathbf{F}(\cdot)$ *be an LTI system. Then its impulse response (i.e. the response of the system to an impulse) is* $\mathbf{F}[\delta(n)]$ *and it is denoted as* $h(n) = \mathbf{F}[\delta(n)]$.

We saw earlier that each signal can be represented by shifted and scaled impulses:

$$x(n) = \sum_{k=-\infty}^{\infty} x(k)\delta(n-k).$$

This can be used to derive convolution for all LTI systems. If $\mathbf{F}(\cdot)$ is LTI, then its response to input $x(n)$ can be represented in the form

$$\begin{aligned} y(n) = \mathbf{F}[x(n)] = & \quad \mathbf{F}[\textstyle\sum_{k=-\infty}^{\infty} x(k)\delta(n-k)] & \\ = & \quad \textstyle\sum_{k=-\infty}^{\infty} \mathbf{F}[x(k)\delta(n-k)] & \text{(Linearity)} \\ = & \quad \textstyle\sum_{k=-\infty}^{\infty} x(k)\mathbf{F}[\delta(n-k)] & \text{(Linearity)} \\ = & \quad \textstyle\sum_{k=-\infty}^{\infty} x(k)h(n-k). & \text{(Shift-invariance)} \end{aligned}$$

This is referred to as *convolution* or *convolution sum* of $x(n)$ and $h(n)$ and it is denoted by $x(n) * h(n)$, i.e.

$$y(n) = x(n) * h(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k).$$

The convolution is commutative. Therefore, the following is an equivalent formulation:

$$y(n) = h(n) * x(n) = \sum_{k=-\infty}^{\infty} h(k)x(n-k).$$

In most cases the latter form is more illustrative.

3.3.2 LTI: Properties of Convolution

Below is a brief summary of the most important properties of convolution and hence LTI systems:

Commutativity: $x(n) * h(n) = h(n) * x(n)$.

Distributivity: $x(n) * (h_1(n) + h_2(n)) = x(n) * h_1(n) + x(n) * h_2(n)$.

Cascade: If the systems $\mathbf{F}_1(\cdot)$ and $\mathbf{F}_2(\cdot)$ having impulse responses $h_1(n)$ and $h_2(n)$ are cascaded, then the impulse response of the entire system $\mathbf{F}_1(\mathbf{F}_2(\cdot))$ is $h_1(n) * h_2(n)$. The system $\mathbf{F}_1(\mathbf{F}_2(\cdot))$ is called *cascade* of the systems $\mathbf{F}_1(\cdot)$ ja $\mathbf{F}_2(\cdot)$ and it is denoted by $\mathbf{F}_1 \circ \mathbf{F}_2(\cdot) = \mathbf{F}_1(\mathbf{F}_2(\cdot))$.

Causality: The system is causal if and only if its impulse response $h(n)$ satisfies the condition:

$$n < 0 \Rightarrow h(n) = 0.$$

Stability: The system is stable if and only if its impulse response $h(n)$ satisfies the condition:

$$\sum_{k=-\infty}^{\infty} |h(k)| < \infty.$$

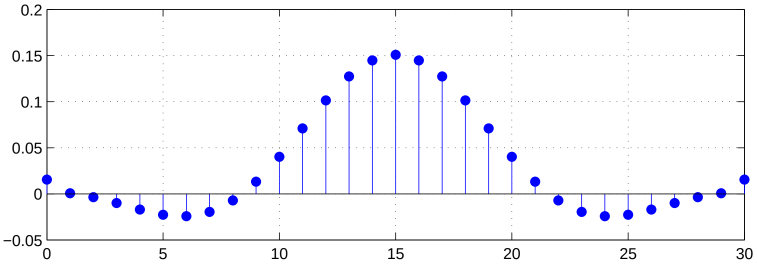
3.3.3 LTI: FIR and IIR Filters

LTI systems are divided into two classes according to the length of the impulse response: finite impulse response (FIR) and infinite impulse response (IIR).

FIR filter: Finite impulse response of the FIR filters means that the filter output has the form:

$$y(n) = \sum_{k=-M_1}^{M_2} h(k)x(n-k),$$

where both M_1 and M_2 are finite. The figure below shows the impulse response of an FIR filter designed with Matlab. Convolution with this impulse response removes from the signal the frequencies higher than a certain limit frequency. The length of the impulse response is 31 and all other impulse response values are equal to zero.



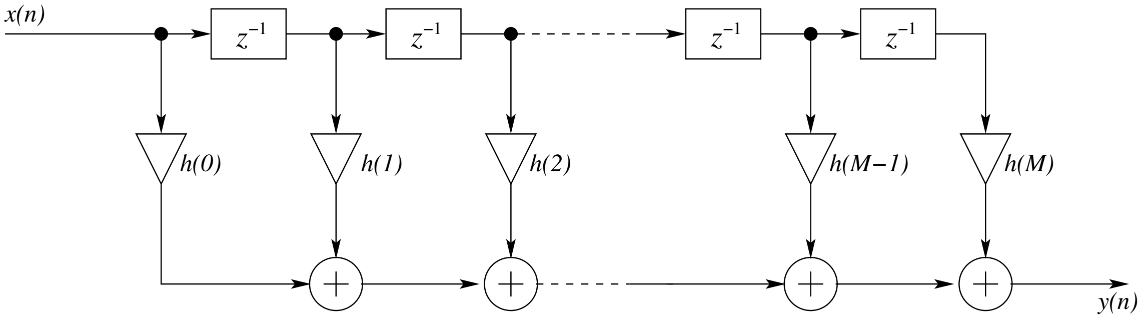
Filtering by this filter can be expressed as

$$y(n) = \sum_{k=0}^{30} h(k)x(n-k).$$

More general formulation for a causal FIR filter is

$$y(n) = \sum_{k=0}^{M} h(k)x(n-k),$$

where M is called the *order* of the filter and its length is $M+1$. Below is the block diagram of the FIR filter.



IIR filter: The impulse response of the IIR filter is infinitely long, i.e. there is no limit, after which the impulse response would always be zero. IIR filters can be implemented by means of difference equations. An infinite impulse response is obtained by feedback (recursion), i.e., by using previous outputs to calculate the current output.

Example: Consider the system determined by the difference equation

$$y(n) = -0.9y(n-1) + x(n)$$

and find the impulse response $h(n)$ of the system.

By definition, the impulse response is the system output when the input is the unit sample $x(n) = \delta(n)$. In these kind of tasks we assume that the system is initially at rest, that is, $y(n) = 0$, when $n < 0$. Now the impulse response function satisfies the following recursive relationship:

$$h(n) = -0.9h(n-1) + \delta(n).$$

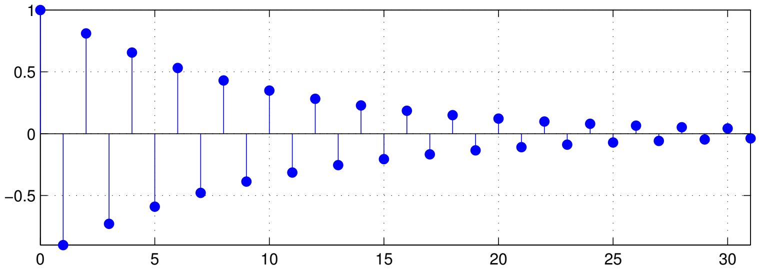
Since this form is recursive, it is not yet the "final answer" for $h(n)$, but we can study this recursion to find $h(n)$. The initial rest condition gives that $h(n) = 0$ for $n < 0$. The following table shows what happens in the recursion:

n	$\delta(n)$	$h(n-1)$	$h(n) = -0.9h(n-1) + \delta(n)$
< 0	0	0	0
0	1	0	1
1	0	1	-0.9
2	0	-0.9	$(-0.9)^2$
3	0	$(-0.9)^2$	$(-0.9)^3$
4	0	$(-0.9)^3$	$(-0.9)^4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

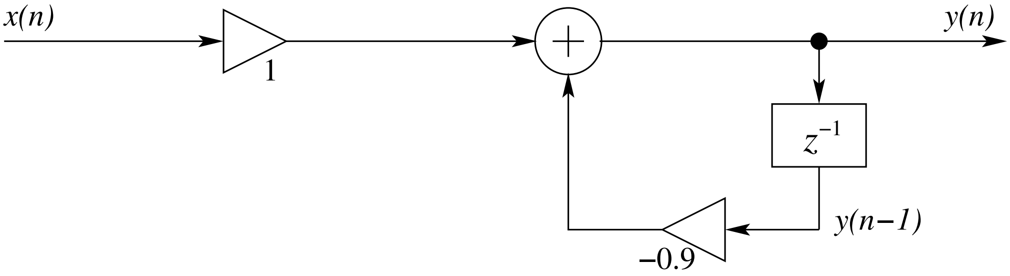
By inspection we see that the impulse response is:

$$h(n) = \begin{cases} (-0.9)^n, & n \geq 0 \\ 0, & n < 0 \end{cases} = (-0.9)^n u(n).$$

This figure shows the first 32 terms of the impulse response. However, the impulse response has infinitely many non-zero values.



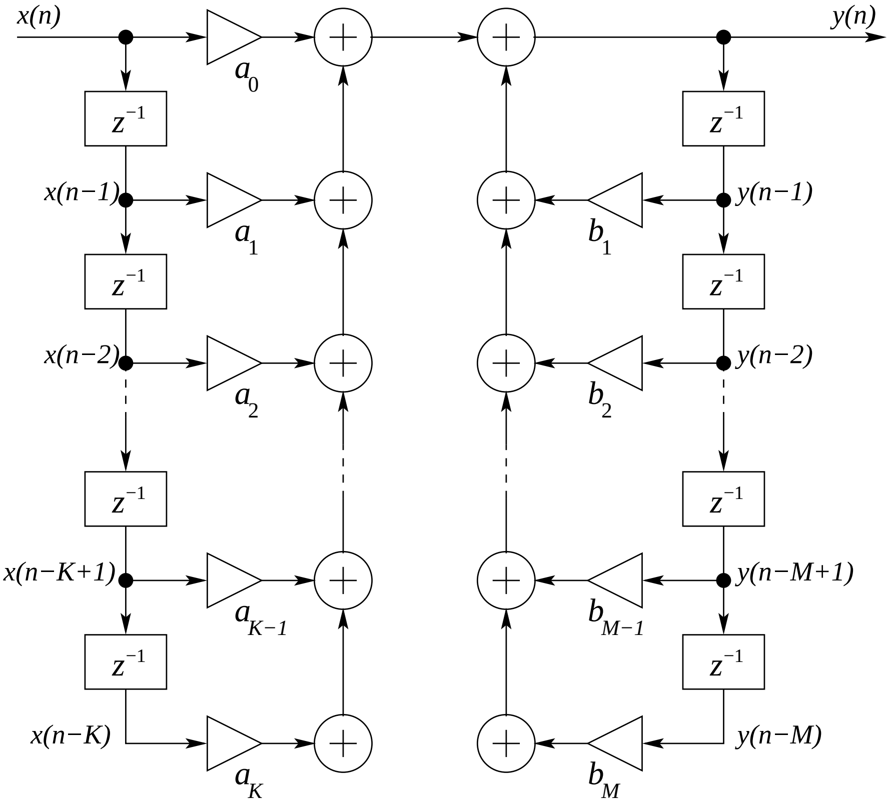
Below is the block diagram of this IIR filter.



The general form of the IIR filter is:

$$y(n) = \sum_{k=0}^{K} a_k x(n-k) + \sum_{m=1}^{M} b_m y(n-m), \tag{3.2}$$

where the coefficients $a_k, k = 0, 1, \ldots, K$ and $b_m, m = 1, 2, \ldots, M$ are the feedforward and feedback filter coefficients, respectively. Here is this difference equation in the form of a diagram.



In part of the literature (and Matlab) another formulation is used for the IIR filter:

$$\sum_{k=0}^{K} b_k x(n-k) = \sum_{m=0}^{M} a_m y(n-m),$$

which can cause confusion. If $a_0 = 1$ (as in all Matlab-designed filters), we can write the expression in the more familiar format:

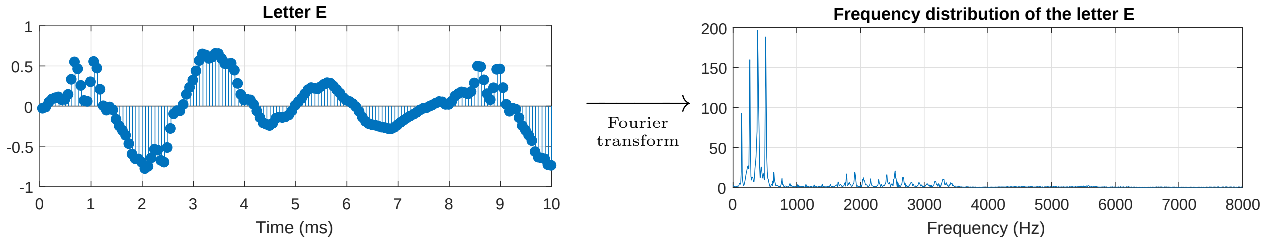
$$y(n) = \sum_{k=0}^{K} b_k x(n-k) - \sum_{m=1}^{M} a_m y(n-m).$$

Notice that in comparison to our previous formulation in Eq. (3.2) the roles of a and b have interchanged, there is an additional feedback filter coefficients $a_0 = 1$, and the rest of the feedback filter coefficients have opposite sign.

Chapter 4

Fourier Transform

Fourier transform connects the signal values to the frequencies the signal contains and answers the question: *How much the signal contains each frequency?* As a result of the transform, the frequency distribution of the signal is obtained. Below is an example of a test signal on the left and its Fourier transform on the right.



Both continuous and discrete signals can be Fourier transformed. Depending on the signal periodicity, the result is either a discrete signal or a continuous-time signal representing the frequencies contained in the signal. The table below shows four different transform types.

Original signal	*Continuous-time*	*Discrete-time*
Non-periodic	Result is non-periodic continuous-time signal. Referred to as **Fourier transform**. (continuous → continuous)	Result is periodic continuous-time signal. Referred to as **discrete-time Fourier transform (DTFT)**. (discrete → continuous)
Periodic	Result is non-periodic discrete-time signal. Referred to as **Fourier series**. (continuous → discrete)	Result is periodic discrete-time signal. Referred to as **discrete Fourier transform (DFT)**. (discrete → discrete)

The Fourier transform represents the signal to be transformed using complex exponential functions $(\ldots, e^{-3it}, e^{-2it}, e^{-it}, e^{0}, e^{it}, e^{2it}, e^{3it}, \ldots)$, where $i = \sqrt{-1}$ is the *imaginary unit*. According to Euler's formula, each of these functions is the sum of the sine and cosine:

$$e^{ikt} = \cos(kt) + i\sin(kt).$$

The most important transform type for applications is the discrete Fourier transform (DFT) and its fast algorithm fast Fourier transform (FFT). In DFT the calculations can be done as matrix multiplications that are finite and easy to perform on a computer. The other transform types are used as theoretical tools for example in filter design. Their calculations include infinite sums as well as integrals that are more difficult to handle with the computer.

We follow the common practice and denote signals with lower-case letters and their Fourier transforms with the corresponding capital letters: the Fourier transform of the signal x is X, of y it is Y, etc.

4.1 Fourier Transform (Non-Periodic Continuous-Time Signal)

When we transform non-periodic continuous-time signal $x(t)$ ($t \in \mathbf{R}$), its Fourier transform is defined as the integral

$$X(e^{i\omega}) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t}\,dt.$$

The result is function of the angular frequency $\omega \in \mathbf{R}$. *Inverse Fourier transform* gives back the original signal:

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(e^{i\omega})e^{i\omega t}\,d\omega.$$

The function $X(e^{i\omega})$ is now function of the real variable ω. In principle, we could use the notation $X(\omega)$ instead of $X(e^{i\omega})$, but it could cause confusion in our future notations.

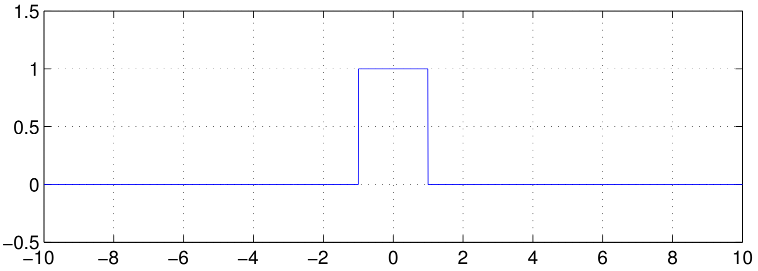
For example, the Fourier transform of the signal

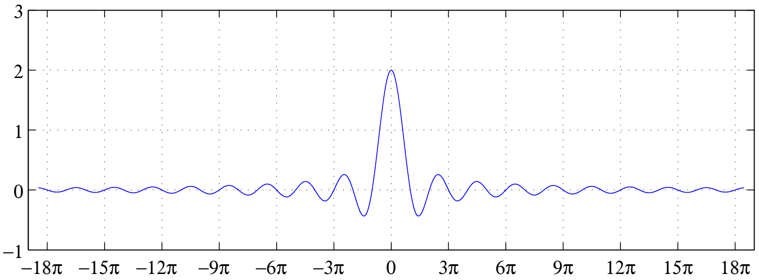
$$x(t) = \begin{cases} 1, & \text{when } -1 \le t \le 1 \\ 0, & \text{otherwise} \end{cases}$$

is

$$X(e^{i\omega}) = \begin{cases} \frac{2\sin(\omega)}{\omega}, & \text{when } \omega \neq 0 \\ 2, & \text{when } \omega = 0. \end{cases}$$

The upper plot shows $x(t)$ and the lower $X(e^{i\omega})$.





Generally, the Fourier transform is a complex valued function, but in this example, the resulting function happened to be real. This is the case whenever the signal to be transformed is symmetrical with respect to the y-axis. However, typically the result has also complex part. Then the transform is presented as two plots: the first is the absolute value of the function and the second the phase angle of the function.

4.2 Fourier Series (Periodic Continuous-Time Signal)

In the case of a periodic signal, the amount of information is smaller in that sense that one period of the signal determines its behavior completely. Thus a discrete number of frequencies is sufficient to represent a periodic signal.

Let $x(t)$ be a periodic continuous-time signal with period 2π (i.e. $x(t) = x(t + 2\pi)$). Its Fourier series is a discrete-time signal

$$X(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} x(t)e^{-int}\,dt,$$

where $n \in \mathbb{Z}$.

If we know the Fourier series $X(n)$, we get back to the original signal by the formula

$$x(t) = \sum_{n=-\infty}^{\infty} X(n)e^{int}.$$

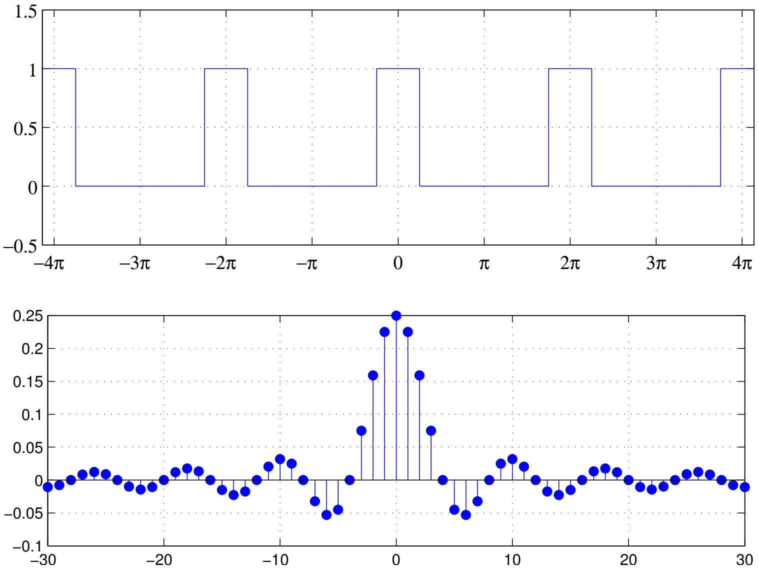
An example of Fourier series presentation of a continuous periodic signal is the function $x(t)$ defined as follows:

$$x(t) = \begin{cases} 1, & \text{when } -\frac{\pi}{4} \le t \le \frac{\pi}{4}, \\ 0, & \text{in other points of the interval } [-\pi, \pi]. \end{cases}$$

The function $x(t)$ is called *rectangular wave* or *pulse wave*. Outside the interval $[-\pi, \pi]$ the function $x(t)$ is periodic: $x(t) = x(t + 2\pi)$. The Fourier series representation of this function is

$$X(n) = \begin{cases} \frac{1}{4}\frac{\sin(n\pi/4)}{n\pi/4}, & \text{when } n \neq 0, \\ \frac{1}{4}, & \text{when } n = 0. \end{cases}$$

Again, the result is real, because the signal is symmetric with respect to the y-axis. The signal (top) and its Fourier series coefficients (bottom):

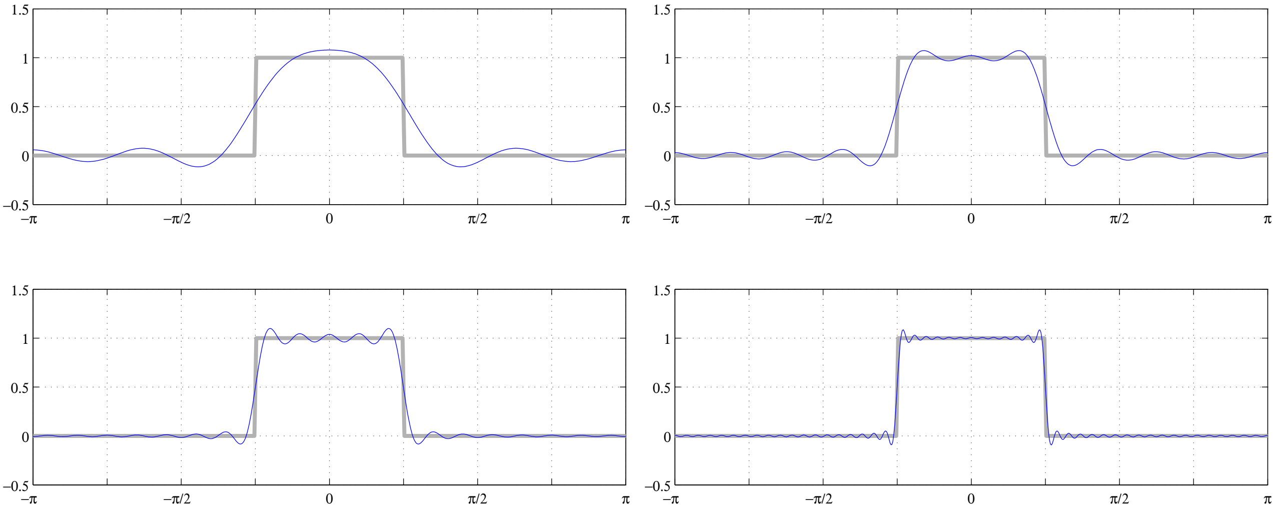


The transition between the minimum and the maximum is instantaneous for an ideal rectangular wave; this is not realizable in physical systems. However, sine and cosine waveforms are easy to generate in such systems and they can be used to generate rectangular waveforms. This is done by taking a small fraction of the Fourier series coefficients. If we use the coefficients $X(-5), X(-4), \ldots, X(5)$ (11 in total) the approximation is:

$$\begin{aligned} \hat{x}(t) &= X(-5)e^{-5it} + X(-4)e^{-4it} + X(-3)e^{-3it} + \cdots + X(4)e^{4it} + X(5)e^{5it} \\ &= -0.0450e^{-5it} + 0.0750e^{-3it} + 0.1592e^{-2it} + 0.2251e^{-it} + 0.2500 \\ &+ \ 0.2251e^{it} + 0.1592e^{2it} + 0.0750e^{3it} - 0.0450e^{5it}, \end{aligned}$$

which is real.

This approximation using 11 coefficients at the top left in the figure below is not yet very accurate, but by increasing the number of coefficients, the function becomes closer to the true rectangular wave. In the upper right 21 coefficients are used, bottom left 41, and finally at the bottom right 101 coefficients.



Even though the approximation becomes more accurate when the number of coefficients is increased, the overshoot at the jump discontinuity does not die out as more terms are added to the sum. It remains at about 9 % of the magnitude of the jump. This type of behavior, typical for the Fourier series, is referred to as the *Gibbs phenomenon*.

4.3 DTFT (Non-Periodic Discrete-Time Signal)

The Fourier transform of the discrete-time non-periodic signal $x(n)$ is a signal

$$X(e^{i\omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-i\omega n},$$

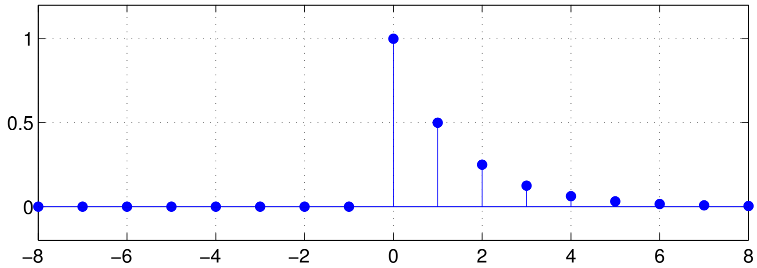
where X is now a function of real variable ω. If function X is known, the original signal can be obtained using the formula

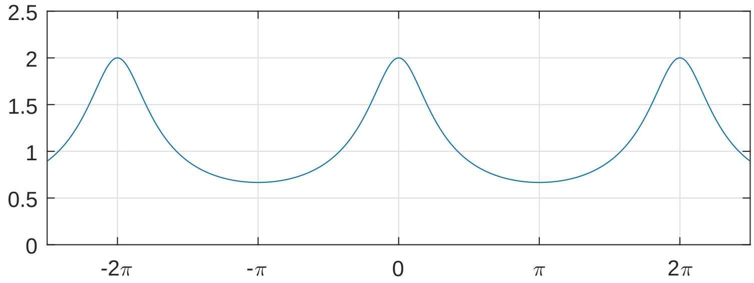
$$x(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{i\omega})e^{i\omega n}\,d\omega.$$

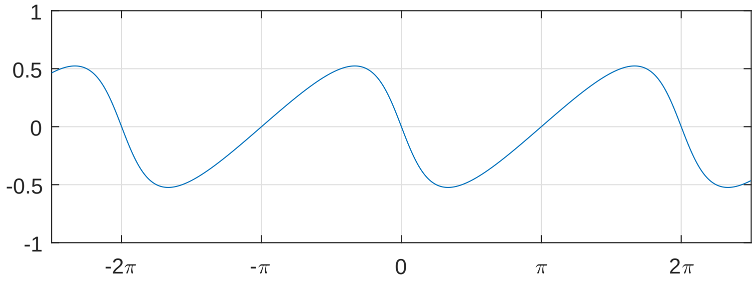
Example: Determine the DTFT of the signal $x(n) = 0.5^n u(n)$. Now

$$
\begin{aligned}
X(e^{i\omega}) &= \sum_{n=-\infty}^{\infty} x(n)e^{-i\omega n} \\
&= \sum_{n=-\infty}^{\infty} 0.5^n e^{-i\omega n} u(n) \\
&= \sum_{n=0}^{\infty} 0.5^n e^{-i\omega n} \\
&= \sum_{n=0}^{\infty} (0.5e^{-i\omega})^n \quad \text{(geom. series, } |0.5e^{-i\omega}| = 0.5 < 1\text{)} \\
&= \frac{1}{1 - 0.5e^{-i\omega}}.
\end{aligned}
$$

Below are graphs of the original signal $x(n)$, absolute value of the DTFT $|X(e^{i\omega})|$ and the phase angle of the DTFT $\arg(X(e^{i\omega}))$ as a function of angular frequency ω.







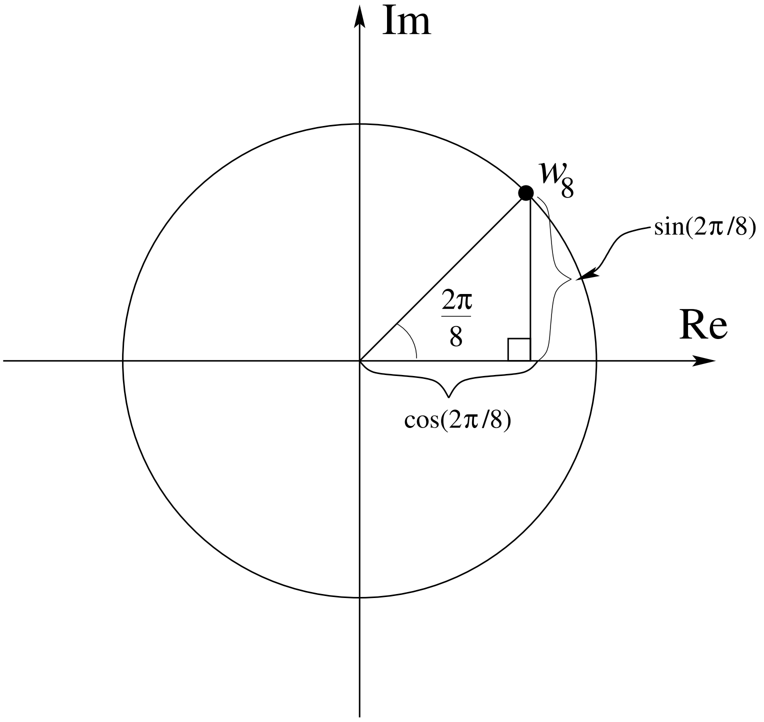
4.4 DFT (Periodic Discrete-Time Signal)

For the applications, the most important of the four types of transforms is the discrete Fourier transform (DFT) that transforms a discrete-time periodic signal into a discrete-time periodic signal. There is now finite number of frequencies in the original signal and the result of the transform contains information of all these frequencies.

Very compact way to define the DFT is to use $w_N = e^{2\pi i/N}$, which is called *N'th root of unity*. As the name suggests, the N'th power of w_N is equal to one:

$$w_N^N = (e^{2\pi i/N})^N = e^{2\pi i} = \cos(2\pi) + i\sin(2\pi) = 1.$$

The number w_N is located on the complex plane unit circle at an angle $2\pi/N$ which is $\frac{1}{N}$ of the whole circle (see figure).



The DFT of a periodic discrete-time signal $x(n)$ (period N) is defined by the formula

$$X(n) = \sum_{k=0}^{N-1} x(k) w_N^{-kn}.$$

The inverse DFT (IDFT) of the signal $X(n)$ is

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) w_N^{kn}.$$

DFT and IDFT can also be represented in a matrix form, which is usually the most convenient form of representation. Let $x(n)$ be the signal to be transformed, and the vector

$$\mathbf{x} = \begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ \vdots \\ x(N-1) \end{pmatrix}$$

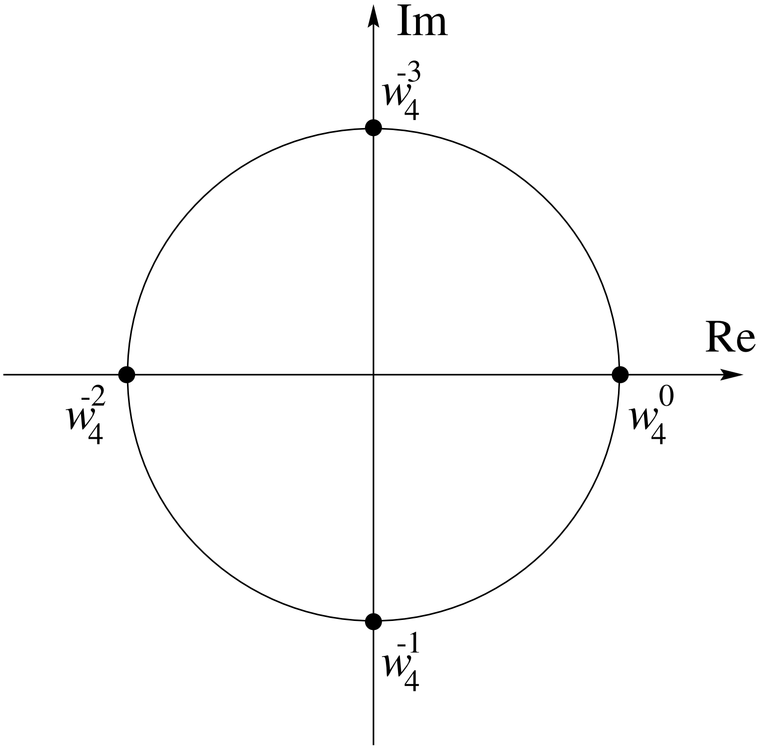
one period of the signal. The DFT is obtained by multiplying the vector $\mathbf{x}$ by the matrix

$$\begin{pmatrix} w_N^0 & w_N^0 & w_N^0 & \dots & w_N^0 \\ w_N^0 & w_N^{-1} & w_N^{-2} & \dots & w_N^{-(N-1)} \\ w_N^0 & w_N^{-2} & w_N^{-4} & \dots & w_N^{-2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_N^0 & w_N^{-(N-1)} & w_N^{-2(N-1)} & \dots & w_N^{-(N-1)^2} \end{pmatrix}.$$

For example, if the period $N = 4$, DFT can be calculated by multiplying the vector by the matrix

$$\begin{pmatrix} w_4^0 & w_4^0 & w_4^0 & w_4^0 \\ w_4^0 & w_4^{-1} & w_4^{-2} & w_4^{-3} \\ w_4^0 & w_4^{-2} & w_4^{-4} & w_4^{-6} \\ w_4^0 & w_4^{-3} & w_4^{-6} & w_4^{-9} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}.$$

The matrix can be generated using the unit circle. In the case $N = 4$, all elements are non-positive powers of w_4. Number w_4 is located on the unit circle at an angle $\frac{2\pi}{4}$, which is quarter of the whole circle. Negative powers are obtained like the positive ones, except that the direction of rotation changes to clockwise.



For example, in case $N = 6$, the transform matrix is as follows:

$$\begin{array}{cl}\begin{pmatrix} w_6^0 & w_6^0 & w_6^0 & w_6^0 & w_6^0 & w_6^0 \\ w_6^0 & w_6^{-1} & w_6^{-2} & w_6^{-3} & w_6^{-4} & w_6^{-5} \\ w_6^0 & w_6^{-2} & w_6^{-4} & w_6^{-6} & w_6^{-8} & w_6^{-10} \\ w_6^0 & w_6^{-3} & w_6^{-6} & w_6^{-9} & w_6^{-12} & w_6^{-15} \\ w_6^0 & w_6^{-4} & w_6^{-8} & w_6^{-12} & w_6^{-16} & w_6^{-20} \\ w_6^0 & w_6^{-5} & w_6^{-10} & w_6^{-15} & w_6^{-20} & w_6^{-25} \end{pmatrix} & \begin{array}{l} \leftarrow \text{exponent change} \quad 0 \\ \leftarrow \text{exponent change} \ -1 \\ \leftarrow \text{exponent change} \ -2 \\ \leftarrow \text{exponent change} \ -3 \\ \leftarrow \text{exponent change} \ -4 \\ \leftarrow \text{exponent change} \ -5 \end{array}\end{array}$$

Example: Calculate DFT of input $x(n)$: $(0, 1, 2, 3)^T$. The transform is matrix multiplication:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 6 \\ -2+2i \\ -2 \\ -2-2i \end{pmatrix}.$$

When the period $N = 4$, the transform matrix is simple, but let's have another example with different period length.

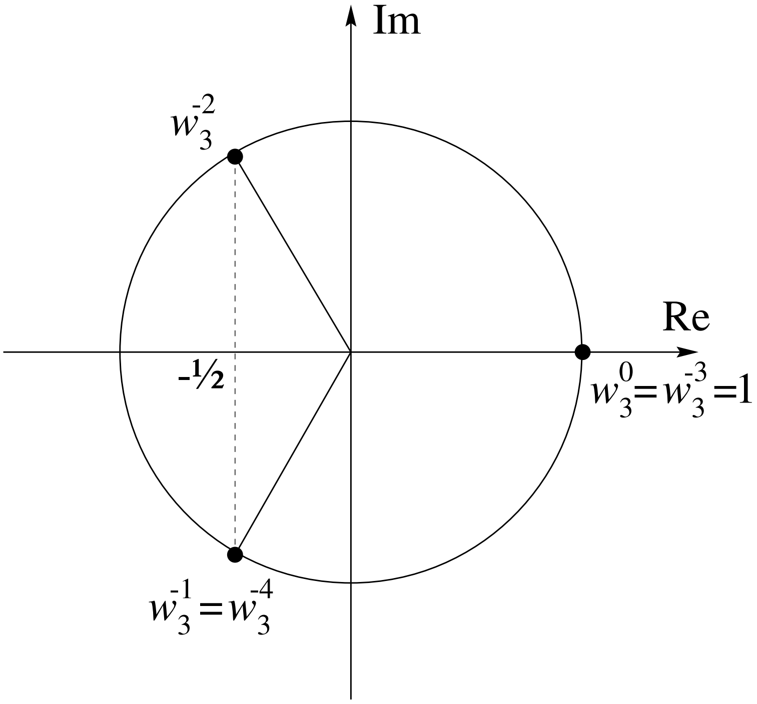
Example: Calculate DFT of input $x(n)$: $(0, 1, 2)^T$. Now, the period is $N = 3$ and the transform matrix is

$$\begin{pmatrix} w_3^0 & w_3^0 & w_3^0 \\ w_3^0 & w_3^{-1} & w_3^{-2} \\ w_3^0 & w_3^{-2} & w_3^{-4} \end{pmatrix}.$$

The numbers $w_3^{-k}, k = 0, 1, 2, 4$, can be calculated using the Euler's formula and the transform matrix is

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -\frac{1}{2} - i\frac{\sqrt{3}}{2} & -\frac{1}{2} + i\frac{\sqrt{3}}{2} \\ 1 & -\frac{1}{2} + i\frac{\sqrt{3}}{2} & -\frac{1}{2} - i\frac{\sqrt{3}}{2} \end{pmatrix}.$$

Calculation of the transform matrix can be illustrated using the unit circle:



The desired DFT is now

$$\mathbf{X} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -\frac{1}{2} - i\frac{\sqrt{3}}{2} & -\frac{1}{2} + i\frac{\sqrt{3}}{2} \\ 1 & -\frac{1}{2} + i\frac{\sqrt{3}}{2} & -\frac{1}{2} - i\frac{\sqrt{3}}{2} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 3 \\ -\frac{3}{2} + i\frac{\sqrt{3}}{2} \\ -\frac{3}{2} - i\frac{\sqrt{3}}{2} \end{pmatrix}.$$

IDFT of the vector $\mathbf{X}$ is obtained by multiplying it with the matrix

$$\frac{1}{N} \begin{pmatrix} w_N^0 & w_N^0 & w_N^0 & \dots & w_N^0 \\ w_N^0 & w_N & w_N^2 & \dots & w_N^{N-1} \\ w_N^0 & w_N^2 & w_N^4 & \dots & w_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_N^0 & w_N^{N-1} & w_N^{2(N-1)} & \dots & w_N^{(N-1)^2} \end{pmatrix},$$

i.e. the inverse matrix of the DFT transform matrix. The matrix differs from the DFT matrix so that the exponents are having opposite sign and the matrix has the scalar multiplier $\frac{1}{N}$. When $N = 4$ the IDFT matrix is

$$\frac{1}{4} \begin{pmatrix} w_4^0 & w_4^0 & w_4^0 & w_4^0 \\ w_4^0 & w_4 & w_4^2 & w_4^3 \\ w_4^0 & w_4^2 & w_4^4 & w_4^6 \\ w_4^0 & w_4^3 & w_4^6 & w_4^9 \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}.$$

Example: Let the DFT of a signal be $X(n) : (-3, \frac{3i+9\sqrt{3}}{3i+\sqrt{3}}, \frac{6i+3\sqrt{3}}{\sqrt{3}})^T$. We find the original signal $x(n)$ by taking the IDFT of the signal $X(n)$ as follows:

$$\frac{1}{3}\begin{pmatrix} 1 & 1 & 1 \\ 1 & -\frac{1}{2}+i\frac{\sqrt{3}}{2} & -\frac{1}{2}-i\frac{\sqrt{3}}{2} \\ 1 & -\frac{1}{2}-i\frac{\sqrt{3}}{2} & -\frac{1}{2}+i\frac{\sqrt{3}}{2} \end{pmatrix}\begin{pmatrix} -3 \\ \frac{3i+9\sqrt{3}}{3i+\sqrt{3}} \\ \frac{6i+3\sqrt{3}}{\sqrt{3}} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -4 \end{pmatrix}.$$

Thus $x(n)$: $(1, 0, -4)^T$.

4.4.1 DFT Properties

The most important properties of the DFT are:

Sequence of numbers	*DFT*
$x(n), x_1(n), x_2(n)$	$X(n), X_1(n), X_2(n)$
$ax_1(n) + bx_2(n)$	$aX_1(n) + bX_2(n)$
$x(n+m)$	$w_N^{nm}X(n)$
$w_N^{-nm}x(n)$	$X(n+m)$
$x_1(n) * x_2(n)$ (convolution)	$X_1(n)X_2(n)$
$x_1(n)x_2(n)$	$\frac{1}{N}X_1(n) * X_2(n)$
$\overline{x(n)}$ (complex conjugate)	$\overline{X(-n)}$

In addition, the following properties hold true if the sequence $x(n)$ is real.

- $X(n) = \overline{X(-n)}$,
- Re $[X(n)]$ =Re $[X(-n)]$,
- Im $[X(n)] = -$Im $[X(-n)]$,
- $|X(n)| = |X(-n)|$ (symmetric with respect to origin), and
- $\arg[X(n)] = -\arg[X(-n)]$.

The Fourier transform converts the convolution into a multiplication. The simplicity and clarity of the multiplication compared to the convolution make this property an important tool for us. Additionally this gives frequency interpretation for the convolution: the convolution means multiplication of the frequency distributions. This enables the design of such sequences that in convolution eliminate certain frequencies completely—it is sufficient that the frequency response in the area to be removed is zero.

4.4.2 FFT

Calculation of the DFT by using the definition is quite laborious. Multiplication of N-dimensional vector by an $N \times N$ matrix requires N^2 multiplications and $N(N-1)$ additions. The time required for the N-dimensional DFT is therefore proportional to the square of the dimension $O(N^2)$. However, using the so-called *fast Fourier Transform* (FFT), the DFT can be implemented with complexity $O(N \log N)$, i.e. significantly faster, especially when N is large.

In an article published in 1965, Cooley and Tukey presented faster way to perform DFT, and this result gave the Fourier analysis new practical applications and through the computing revolution of the next decade, made FFT one of the indispensable algorithms in digital signal processing. Although the principle of FFT was already invented by Gauss, Cooley and Tukey were the ones to formulate it as a general version useful in computer-based computing.

This Cooley's and Tukey's FFT is based on the so-called *divide and conquer* algorithm design method, where the given task is first converted to minor subproblems, which can then be solved considerably more easily than the original problem. In the case of the FFT algorithm, the vector to be transformed is divided into two parts and the vectors thus obtained are recursively transformed by the same algorithm. The recursion ends when the dimension of the vector to be transformed is one because the Fourier transform of a one-dimensional vector is by definition the vector itself. Since in every step the vector is divided in two, its original dimension must be of the form 2^k, $k \in \mathbb{N}$.

When deriving the transform we need the following connection between the numbers w_k and w_{2k}:

$$w_{2k}^2 = e^{\frac{2 \cdot 2\pi i}{2k}} = e^{\frac{2\pi i}{k}} = w_k. \tag{4.1}$$

Let $x(n)$ be a periodic sequence of numbers with the period $N = 2^k$ ($k \in \mathbb{N}$). Its DFT is by the definition

$$X(n) = \sum_{k=0}^{N-1} x(k) w_N^{-nk}.$$

Let us consider separately the components having even and odd indices:

$$X(n) = \sum_{k=0}^{\frac{N}{2}-1} x(2k) w_N^{-n(2k)} + \sum_{k=0}^{\frac{N}{2}-1} x(2k+1) w_N^{-n(2k+1)}.$$

Since by equation (4.1) $w_N^2 = w_{N/2}$, then

$$X(n) = \sum_{k=0}^{\frac{N}{2}-1} x(2k) w_{N/2}^{-nk} + w_N^{-n} \sum_{k=0}^{\frac{N}{2}-1} x(2k+1) w_{N/2}^{-nk}.$$

If we denote $x_0(k) = x(2k)$ and $x_1(k) = x(2k+1)$ (and Fourier transforms correspondingly $X_0(n)$ and $X_1(n)$), then we obtain

$$X(n) = X_0(n) + w_N^{-n} X_1(n), \tag{4.2}$$

when $n = 0, 1, 2, \ldots, N/2 - 1$ and

$$X(n) = X_0(n - N/2) + w_N^{-n} X_1(n - N/2), \tag{4.3}$$

when $n = N/2, N/2 + 1, \ldots, N - 1$.

By means of modulo operation, these can also be presented as a single formula:

$$X(n) = X_0 \left(\bmod\left(n, N/2\right)\right) + w_N^{-n} X_1 \left(\bmod\left(n, N/2\right)\right), \tag{4.4}$$

where $n = 0, 1, \ldots, N-1$ and $\mathrm{mod}(a, b)$ is the remainder after division of a by b.

Thus, the DFT for a periodic sequence with period N is obtained by calculating two DFTs having lengths $N/2$. These two are further recursively calculated by calculating a total of four DFTs with lengths $N/4$. We have now obtained the FFT algorithm.

FFT algorithm

1. If the dimension of the vector to be transformed is $N = 1$, return it as such.
2. Divide the sequence $x(n)$ to be transformed into two sequences $x_0(n)$ and $x_1(n)$ of which $x_0(n)$ consists of the components having even indices and $x_1(n)$ of those having odd indices.
3. Calculate the DFTs of the sequences $x_0(n)$ and $x_1(n)$ recursively using this algorithm.
4. The result is obtained by combining the sequences $X_0(n)$ and $X_1(n)$ by formulas (4.2) and (4.3) or (4.4).

Example: We consider a periodic sequence $(0, 1, 2, 3)^T$ having period 4 and find its DFT using the FFT algorithm. First, we form the sequences

$$x_0(n) : (0, 2)^T \text{ and } x_1(n) : (1, 3)^T$$

and recursively divide them to sequences

$$x_{00}(n) : 0, x_{01}(n) : 2, x_{10}(n) : 1 \text{ and } x_{11}(n) : 3.$$

Now the dimensions of the vectors to be transformed are $N = 1$, so we obtain

$$X_{00}(0) = 0, X_{01}(0) = 2, X_{10}(0) = 1 \text{ and } X_{11}(0) = 3.$$

We combine the sequences $X_{00}(n)$ and $X_{01}(n)$ by formulas (4.2) and (4.3) to obtain $X_0(n)$:

$$\begin{aligned} X_0(0) &= \underbrace{X_{00}(0)}_{=0} + \underbrace{w_2^0}_{=1} \underbrace{X_{01}(0)}_{=2} = 2, \\ X_0(1) &= \underbrace{X_{00}(1-1)}_{=0} + \underbrace{w_2^{-1}}_{=-1} \underbrace{X_{01}(1-1)}_{=2} = -2. \end{aligned}$$

这里的（0）（1）从0开始的指顺序
位置，即
第一个，第二个

The same is done for the sequences $X_{10}(n)$ and $X_{11}(n)$ to obtain $X_1(n)$:

$$\begin{aligned} X_1(0) &= \underbrace{X_{10}(0)}_{=1} + \underbrace{w_2^0}_{=1} \underbrace{X_{11}(0)}_{=3} = 4, \\ X_1(1) &= \underbrace{X_{10}(1-1)}_{=1} + \underbrace{w_2^{-1}}_{=-1} \underbrace{X_{11}(1-1)}_{=3} = -2. \end{aligned}$$

Now we have obtained the DFTs $X_0(n)$: $(2, -2)^T$ and $X_1(n)$: $(4, -2)^T$. Finally we combine these by formulas (4.2) and (4.3) to obtain $X(n)$:

$$\begin{aligned}
X(0) &= \underbrace{X_0(0)}_{=2} + \underbrace{w_4^0}_{=1}\underbrace{X_1(0)}_{=4} = 6,\\
X(1) &= \underbrace{X_0(1)}_{=-2} + \underbrace{w_4^{-1}}_{=-i}\underbrace{X_1(1)}_{=-2} = -2+2i,\\
X(2) &= \underbrace{X_0(2-2)}_{=2} + \underbrace{w_4^{-2}}_{=-1}\underbrace{X_1(2-2)}_{=4} = -2,\\
X(3) &= \underbrace{X_0(3-2)}_{=-2} + \underbrace{w_4^{-3}}_{=i}\underbrace{X_1(3-2)}_{=-2} = -2-2i.
\end{aligned}$$

The above algorithm is called *radix-2 decimation-in-time* (DIT) FFT algorithm. Similarly, the original division can be into four different parts, giving the so-called *radix-4* FFT algorithm. Division into eight parts produces the *radix-8* algorithm, etc. In these the length of the sequence must be a power of four or eight, respectively.

Chapter 5

z-Transform

The z-transform can be considered as a generalization of the Fourier transform and it is an important tool for analysis and design of discrete-time systems. It can also be considered to be a discrete-time equivalent of the *Laplace transform.*

The z-transform converts a sequence of numbers to a rational function and by studying properties of this function we can find out certain properties of the sequence. For example, the causality or stability of a system can easily be seen from the z-transform of the impulse response.

5.1 Definition of z-Transform

Let $x(n)$ be a sequence of numbers. Its z-transform is a series

$$X(z) = \ldots + x(-3)z^3 + x(-2)z^2 + x(-1)z + x(0) + x(1)z^{-1} + x(2)z^{-2} + x(3)z^{-3} + \ldots$$

We will use a shorter notation for this

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n}. \tag{5.1}$$

The convergence of the z-transform depends on both the values $x(n)$ and the point $z \in \mathbf{C}$, where $X(z)$ is calculated. Therefore, we need to specify for the z-transform the part of the complex plane in which the z-transform series converges. This *region of convergence* (ROC) of the series is the complex plane region in which the series (5.1) converges absolutely, i.e.

$$\sum_{n=-\infty}^{\infty} \left|x(n)z^{-n}\right| < \infty.$$

It can be shown that the ROC of the series $X(z)$ is always a ring-shaped region

$$r_- < |z| < r_+.$$

Possibly the ROC also includes edge points of the ring, i.e. points that satisfy the condition $|z| = r_-$ or $|z| = r_+$. In addition, r_+ may also be infinite and r_- may be zero.

Since the z-transform is a generalization of the Fourier transform, we use the same notation as for the Fourier transform: the sequence is denoted by a lower case letter and its z-transform by the corresponding capital letter, e.g.

$$\begin{aligned} x(n) &\leftrightarrow X(z), \\ y(n) &\leftrightarrow Y(z), \\ w(n) &\leftrightarrow W(z). \end{aligned}$$

In signal processing problems, we often consider sequences having z-transforms that are rational functions, i.e., of the form

$$X(z) = \frac{N(z)}{D(z)} = \frac{\sum_{m=0}^{M} a_m z^{-m}}{\sum_{k=0}^{K} b_k z^{-k}}.$$

The roots of the denominator polynomial $D(z)$ are called *poles* and the roots of the numerator $N(z)$ *zeros*. Note that all the exponents are usually negative, so in order to calculate the roots they must be expanded to be positive.

Example: Calculate the poles and zeros when

$$X(z) = \frac{1 - 4z^{-1} + z^{-2}}{1 - z^{-1} + \frac{1}{2}z^{-2}}.$$

Expand the negative powers away by multiplying the numerator and denominator by z^2:

$$X(z) = \frac{z^2 - 4z + 1}{z^2 - z + \frac{1}{2}}.$$

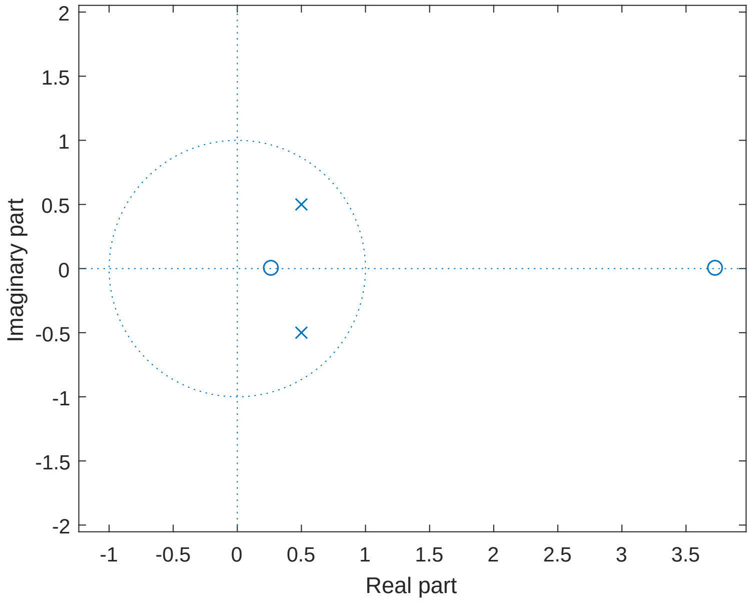
The roots of the numerator (i.e., *zeros*) are solved by the quadratic formula:

$$z_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{4 \pm \sqrt{16 - 4}}{2} = 2 \pm \sqrt{3} = \begin{cases} 3.73, \\ 0.27. \end{cases}$$

The roots of the denominator (i.e., *poles*) are obtained correspondingly:

$$p_{1,2} = \frac{1 \pm \sqrt{1 - 2}}{2} = \frac{1}{2} \pm \frac{i}{2}.$$

Poles and zeros are often shown as a *pole-zero plot* in the complex plane, where the locations of the poles are indicated by a cross '×' and the locations of the zeros are indicated by a circle '○'. The pole-zero plot of the previous example is:



The function $X(z)$ and the ROC determine the signal $x(n)$ uniquely. If the ROC is unknown, $X(z)$ may be the z-transform of several functions. We will mostly consider causal signals $x(n)$, for which $x(n) = 0$, for $n < 0$. In this case, the z-transform obtains the form

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n}.$$

5.2 Useful z-Transforms

Here are examples of z-transforms of some useful simple signals. Since the z-transform is linear, more complex z-transforms can often be determined by utilizing these simple cases as building blocks.

Impulse: The z-transform of the signal $x(n) = \delta(n)$ is

$$X(z) = 1$$

and the ROC is the whole complex plane $\mathbf{C}$.

Delayed impulse: Let the sequence be $x(n) = \delta(n-d)$ $(d \in \{1, 2, 3, \ldots\})$. Then

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n} = \sum_{n=-\infty}^{\infty} \underbrace{\delta(n-d)}_{=0,\text{when } n\neq d} z^{-n} = z^{-d},$$

and since the value of $X(z)$ is infinite if $z = 0$, the ROC is $\mathbf{C} \setminus \{0\}$.

Exponential sequence: The z-transform of the signal $x(n) = a^n u(n)$ is

$$\begin{aligned} X(z) &= \sum_{n=-\infty}^{\infty} x(n) z^{-n} = \sum_{n=-\infty}^{\infty} a^n u(n) z^{-n} \\ &= \sum_{n=0}^{\infty} \left(az^{-1}\right)^n = \frac{1}{1-az^{-1}} = \frac{z}{z-a}, \end{aligned}$$

and the ROC is $|z| > |a|$ (see convergence of geometric series). Furthermore, we can observe that the z-transform has a pole at point $z = a$ and a zero at point $z = 0$.

Two different signals may have the same z-transform, but a different ROC. Namely, the z-transform of the signal

$$x(n) = \begin{cases} -a^n, & \text{when } n < 0 \\ 0, & \text{when } n \geq 0 \end{cases}$$

is

$$\begin{aligned} X(n) &= -\sum_{n=-\infty}^{-1} a^n z^{-n} = -\sum_{n=1}^{\infty} \left(a^{-1}z\right)^n = 1 - \sum_{n=0}^{\infty} \left(a^{-1}z\right)^n \\ &= 1 - \frac{1}{1-a^{-1}z} = 1 + \frac{a}{z-a} = \frac{z}{z-a}, \end{aligned}$$

and the ROC of the sequence is $\left|\frac{z}{a}\right| < 1$, i.e. $|z| < |a|$. This and the exponential sequence have the same z-transform, but different ROCs. Consequently, the ROC is an essential part of the z-transform.

The table below summarizes the most useful z-transforms:

Sequence	z-Transform	ROC
$\delta(n)$	1	$z \in \mathbf{C}$
$\delta(n-m)$, $m > 0$	z^{-m}	$z \in \mathbf{C} \setminus \{0\}$
$\alpha^n u(n)$	$\frac{1}{1-\alpha z^{-1}}$	$\lvert z\rvert > \lvert\alpha\rvert$

5.3 Inverse z-Transform

The inverse transform of the function $X(z)$ is

$$x(n) = \frac{1}{2\pi i} \int_C X(z) z^{n-1}\, dz,$$

where C is a counterclockwise closed path encircling the origin and entirely in the ROC. In practice, $x(n)$ is typically obtained easier by division or partial fraction decomposition.

Example: Let the z-transform of the signal $x(n)$ be

$$X(z) = \frac{z^{-2} + 2z^{-1} + 2}{z^{-1} + 1}.$$

Find out $x(n)$, when the ROC is $|z| > 1$.

We try to modify the expression into a form where it is composed of our 'familiar' z-transforms:

$$\begin{aligned} & \frac{z^{-2}+2z^{-1}+2}{z^{-1}+1} \\ = & \frac{z^{-1}(z^{-1}+1)+(z^{-1}+1)+1}{z^{-1}+1} \\ = & z^{-1}+1+\frac{1}{z^{-1}+1} \\ = & z^{-1}+1+\frac{1}{1-(-z^{-1})}. \end{aligned}$$

We observe that $X(z)$ consists of the sum of the z-transforms of the signals $\delta(n-1)$, $\delta(n)$ and $(-1)^n u(n)$. Therefore

$$x(n) = \delta(n-1)+\delta(n)+(-1)^n u(n) = \begin{cases} 0, & \text{when } n < 0, \\ 2, & \text{when } n = 0, \\ 0, & \text{when } n = 1, \\ (-1)^n, & \text{when } n > 1. \end{cases}$$

This might not always be an easy thing to do, but then we can use polynomial long division.

Example: Consider the signal $x(n)$ with the z-transform

$$X(z) = \frac{z^3-2z-1}{z^3-z^2-z}$$

and ROC $z \neq 0$. We will now do the polynomial long division:

$$\begin{array}{r|l} & 1+z^{-1} \\ \cline{2-2} z^3-z^2-z & z^3 \qquad -2z-1 \\ & \mp z^3 \pm z^2 \pm z \\ \cline{2-2} & \qquad z^2-z \\ & \qquad \mp z^2 \pm z \quad \pm 1 \\ \cline{2-2} & \qquad\qquad\quad 0 \end{array}$$

Thus $X(z)$ consists of the sum of the z-transforms of the signals $\delta(n-1)$ and $\delta(n)$, so

$$x(n) = \delta(n-1)+\delta(n) = \begin{cases} 1, & \text{when } n = 0, 1, \\ 0, & \text{otherwise.} \end{cases}$$

5.4 Properties of z-Transform

z-transform is linear: Let $X(z)$ and $Y(z)$ be the z-transforms of the sequences $x(n)$ and $y(n)$. Then the z-transform of the sequence $ax(n)+by(n)$ is $aX(z)+bY(z)$. If $X(z)$ and $Y(z)$ have the ROCs R_x and R_y then the ROC of $aX(z)+bY(z)$ is *at least* $R_x \cap R_y$.

Delayed sequence: If the z-transform of the sequence $x(n)$ is $X(z)$ and the sequence $w(n) = x(n-d)$ $(d \in \mathbb{N})$, then the z-transform of the sequence $w(n)$ is

$$W(z) = z^{-d}X(z).$$

The ROC of $W(z)$ is ROC of $X(z)$ except $z = 0$ (since the value of $W(z)$ is infinite if $z = 0$). The corresponding result is also valid for advancing the sequence, but it is less used. It is formulated as follows:

$$w(n) = x(n+d) \Rightarrow W(z) = z^{d}X(z),$$

whenever $d \in \mathbb{N}$. The ROC of $W(z)$ is ROC of $X(z)$ except $z = \infty$ (since the value of $W(z)$ is infinite if $z = \infty$).

z-transform of convolution: We denote $w(n) = x(n) * y(n)$. Then $W(z) = X(z)Y(z)$, and the ROC of $W(z)$ is *at least* the intersection of ROCs of $X(z)$ and $Y(z)$.

Elementwise multiplication: We denote $w(n) = x(n)y(n)$. Then the z-transform of the sequence $w(n)$ is

$$W(z) = \frac{1}{2\pi i}\int_C Y\left(\frac{z}{v}\right) X(v)v^{-1}\,dv,$$

where C is a counterclockwise closed path encircling the origin and being entirely in the ROCs of both $X(z)$ and $Y(z)$.

Complex conjugate: If $y(n) = \overline{x(n)}$, then $Y(z) = \overline{X(\overline{z})}$.

Parseval's theorem:

$$\sum_{n=-\infty}^{\infty} x_1(n)\overline{x_2(n)} = \frac{1}{2\pi i}\oint_C X_1(v)\overline{X_2\left(\frac{1}{\overline{v}}\right)}v^{-1}\,dv.$$

Differentiation of $X(z)$: The z-transform of the sequence $nx(n)$ is $-zX'(z)$ and the ROC is the ROC of $X(z)$ except possibly excluding the boundary if $X(z)$ is irrational.

Time reversal: The z-transform of the sequence $x(-n)$ is $X(z^{-1})$ and the ROC is $\frac{1}{r_+} < |z| < \frac{1}{r_-}$.

Initial value theorem: If the sequence $x(n)$ is causal (i.e. $x(n) = 0$, when $n < 0$), then

$$x(0) = \lim_{|z|\to\infty} X(z).$$

It is enough that $|z|$ approaches infinity along any path.

5.5 Transfer Function of FIR Filter

Previously we stated that an LTI system is fully defined when its impulse response $h(n)$ is known. However, the more common way is to use *transfer function* $H(z)$ which is the z-transform of the impulse response $h(n)$.

Using the impulse response, the relationship between the input $x(n)$ and the output $y(n)$ has the form

$$y(n) = h(n) * x(n).$$

By taking z-transforms of both sides of the equation, we obtain

$$Y(z) = H(z)X(z). \tag{5.2}$$

Example: Consider a system, which is defined by the following difference equation

$$y(n) = 0.0349x(n) + 0.4302x(n-1) - 0.5698x(n-2) + 0.4302x(n-3) + 0.0349x(n-4).$$

The impulse response of the system is

$$h(n) = \begin{cases} 0.0349, & \text{when } n = 0 \text{ tai } n = 4, \\ 0.4302, & \text{when } n = 1 \text{ tai } n = 3, \\ -0.5698, & \text{when } n = 2, \\ 0, & \text{otherwise.} \end{cases}$$

The transfer function is the z-transform of the impulse response:

$$H(z) = \sum_{k=-\infty}^{\infty} h(k)z^{-k} = 0.0349 + 0.4302z^{-1} - 0.5698z^{-2} + 0.4302z^{-3} + 0.0349z^{-4}.$$

5.6 Frequency Response

The transfer function is also used to define other important concepts. When we substitute $z = e^{i\omega}$ into equation (5.2), we obtain its special case

$$Y(e^{i\omega}) = H(e^{i\omega})X(e^{i\omega}). \tag{5.3}$$

This means that, when filtering the signal, its frequencies (i.e., the signal's DTFT $X(e^{i\omega})$) are multiplied by the *frequency response* $H(e^{i\omega})$ of the system. Thus, the frequency behavior of the system depends entirely on the the frequency response function $H(e^{i\omega})$. The frequency response is a complex valued function of the real variable ω that is obtained by evaluating the transfer function when the complex plane unit circle is circulated counterclockwise. This is due to the fact that the expression $e^{i\omega}$ goes through all the points on the unit circle when the variable $\omega \in [0, 2\pi]$.

When we substitute $z = e^{i\omega}$ into the definition of the z-transform we obtain formula for calculating the frequency response:

$$H(e^{i\omega}) = \sum_{n=-\infty}^{\infty} h(n)e^{-i\omega n}.$$

In fact, this is the DTFT of the impulse response $h(n)$.

Example: The frequency response of the system in the previous example is

$$H(e^{i\omega}) = 0.0349 + 0.4302e^{-i\omega} - 0.5698e^{-2i\omega} + 0.4302e^{-3i\omega} + 0.0349e^{-4i\omega}.$$

5.7 Amplitude and Phase Responses

The absolute value of the frequency response $|H(e^{i\omega})|$ is called the *amplitude response* (magnitude response or gain) and its phase angle the *phase response* (phase shift). The meaning of the amplitude response is revealed by taking absolute values from both sides of the equation $Y(e^{i\omega}) = H(e^{i\omega})X(e^{i\omega})$:

$$|Y(e^{i\omega})| = |H(e^{i\omega})| \cdot |X(e^{i\omega})|.$$

The amplitude response therefore indicates how much different frequencies of the signal are attenuated or amplified in the filtering. For example, if the amplitude response of the filter has value 0.6 for some frequency ω (i.e. $|H(e^{i\omega})| = 0.6$), the amplitude of the signal at that frequency is 0.6 times the original amplitude. If the amplitude response is one, the corresponding frequency will not change in the filtering at all (except for possible delay) and if the amplitude response is zero, the frequency disappears completely in the filtering.

The meaning of the phase response becomes apparent when viewing the phase angles of both sides of the equation $Y(e^{i\omega}) = H(e^{i\omega})X(e^{i\omega})$. When complex numbers are multiplied, the phase angles are summed and the following equation is obtained

$$\arg(Y(e^{i\omega})) = \arg(H(e^{i\omega})) + \arg(X(e^{i\omega})).$$

Thus, it is possible to deduce from the phase response how much each frequency is delayed in the filtering. If for example, the phase response of a filter is $\arg(H(e^{i\omega})) = -\pi/2$ at some frequency ω, the filter transforms the signal $\sin(\omega n)$ into $\sin(\omega n - \pi/2)$ (and possibly attenuates or amplifies it).

Example: The frequency response at the frequency $\omega = 0.05 \cdot 2\pi$ of a system defined by the difference equation

$$y(n) = 0.0349x(n) + 0.4302x(n-1) - 0.5698x(n-2) + 0.4302x(n-3) + 0.0349x(n-4)$$

is

$$H(e^{0.05\cdot 2\pi i}) = 0.2467 - 0.1793i.$$

The amplitude response at this frequency is

$$|0.2467 - 0.1793i| = \sqrt{0.2467^2 + (-0.1793)^2} = 0.3050$$

and the phase response

$$\arg(0.2467 - 0.1793i) = \arctan(-0.1793/0.2467) = -0.6283.$$

If the input of the system is now a signal oscillating at that particular frequency

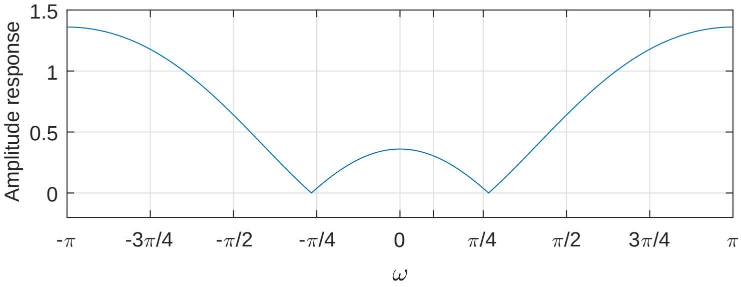
$$x(n) = u(n)\sin(0.05 \cdot 2\pi n),$$

the response is a signal with the same frequency, amplitude 0.3050 times the original and phase delayed by 0.6283 radians. As an equation, this signal is $y(n) = 0.3050u(n)\sin(0.05 \cdot 2\pi n - 0.6283)$. The actual filtering result differs slightly from this estimate in the beginning of the filtering. However, in an exercise task we will notice that elsewhere the estimate is exactly the same as the actual filtering result.

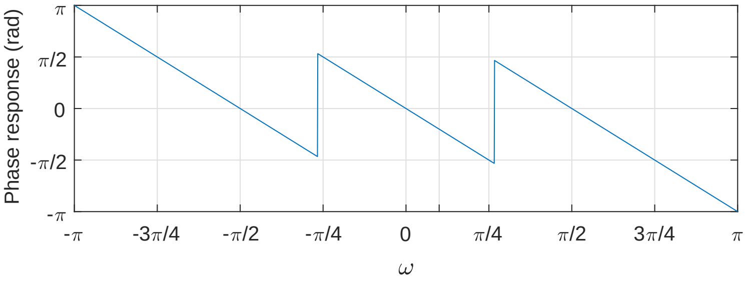
It should be remembered that the amplitude and phase responses are functions whose values depend on the frequency variable ω. For the system we have been studying in the examples, the amplitude response formula is

$$|H(e^{i\omega})| = |0.0349 + 0.4302e^{-i\omega} - 0.5698e^{-2i\omega} + 0.4302e^{-3i\omega} + 0.0349e^{-4i\omega}|.$$

The amplitude response of this function is shown below.



The phase response looks like the following. It has a jump of size π at the points where the frequency response goes to zero. (The reason for this will be considered later.)



Example: We will consider the transfer function

$$H(z) = \frac{z^2 + 0.5z + 1}{-2z^2 + z + 0.5}.$$

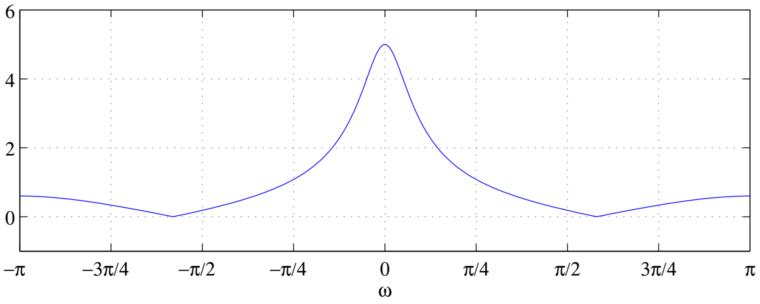
The frequency response of the system is obtained by substituting the expression $e^{i\omega}$ for z

$$H(e^{i\omega}) = \frac{(e^{i\omega})^2 + 0.5e^{i\omega} + 1}{-2(e^{i\omega})^2 + e^{i\omega} + 0.5} = \frac{e^{2i\omega} + 0.5e^{i\omega} + 1}{-2e^{2i\omega} + e^{i\omega} + 0.5}.$$

The amplitude response is the absolute value of this expression

$$|H(e^{i\omega})| = \frac{|e^{2i\omega} + 0.5e^{i\omega} + 1|}{|-2e^{2i\omega} + e^{i\omega} + 0.5|},$$

which cannot be further simplified. However, it can be plotted with Matlab.



Example: Let $h(n) = 0.9^n u(n)$. Then

$$H(z) = \frac{1}{1 - 0.9z^{-1}}, \qquad |z| > 0.9.$$

The frequency response is

$$H(e^{i\omega}) = \frac{1}{1 - 0.9e^{-i\omega}},$$

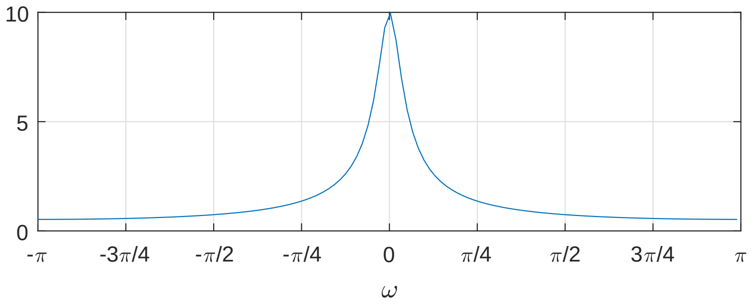
whereby the amplitude response is

$$\left|H(e^{i\omega})\right| = \frac{1}{|1 - 0.9e^{-i\omega}|},$$

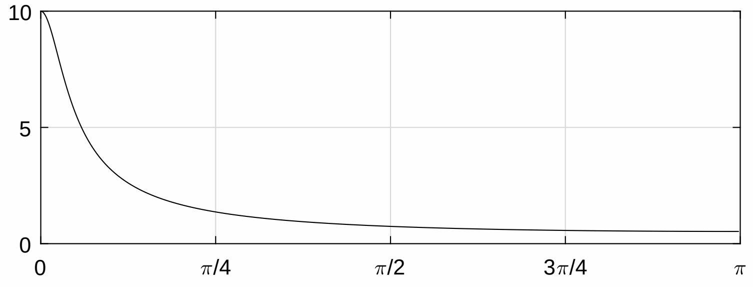
and the phase response

$$\arg(H(e^{i\omega})) = \arg\left(\frac{1}{1 - 0.9e^{-i\omega}}\right).$$

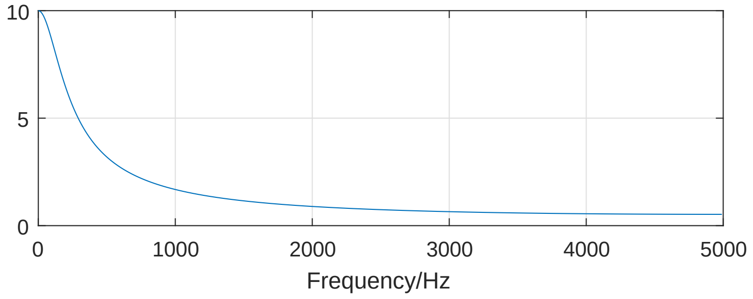
Here is the amplitude response:



If the impulse response consists of real numbers (as is usually the case), the amplitude response is symmetrical with respect to the vertical axis. Consequently, only the right-hand side is usually shown.



As mentioned earlier, the frequency response shows how the system works when inputted with sinusoidal signals having different frequencies. Let us assume that the input of the system of the previous example is a signal with the sampling rate 10000 Hz. Then the zero frequency of the previous figure corresponds to a signal having the frequency of 0 Hz and the frequency $\omega = \pi$ (extreme right) corresponds to the Nyquist frequency 5000 Hz.



It is now directly seen that, for example, at 2500 Hz, the signal decreases to an amplitude of about 0.8 times. Correspondingly, the small frequencies are amplified.

5.8 Decibel Scale

In signal processing, it is often customary to represent amplitude response plots in *decibels* (dB). Since the decibel scale is logarithmic, it is especially good at representing values that range from very small to very large numbers. The logarithmic scale approximately matches the human perception of both sound and light.

If input signal power of the system is P_0 and the output power is P, amplification in *bels* is

$$\log_{10} \frac{P}{P_0} \text{ B.}$$

Generally, the values to be obtained are in the range of 1-10 B. The bel is too large for typical uses, so the decibel (dB), equal to one tenth of the bel, is more commonly used. In decibels the amplification is thus

$$10 \log_{10} \frac{P}{P_0} \text{ dB.}$$

Often in signal processing we study the change of amplitude. The power is proportional to the square of the amplitude. If the amplitude of the input is A_0 and of the output A, then the amplification in dB is

$$10\log_{10}\frac{A^2}{A_0^2}\ \text{dB} = 20\log_{10}\frac{A}{A_0}\ \text{dB}.$$

If the ratio $\frac{A}{A_0}$ is less than one, we talk about attenuation.

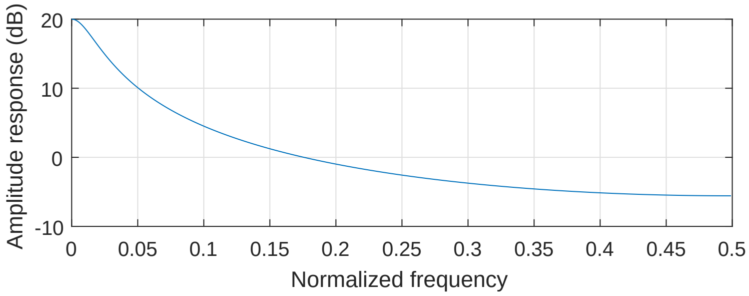
If the system attenuates the amplitude of the signal to half (0.5 times), the attenuation in decibels is

$$20\log_{10} 0.5\ \text{dB} = -6.02\ \text{dB}.$$

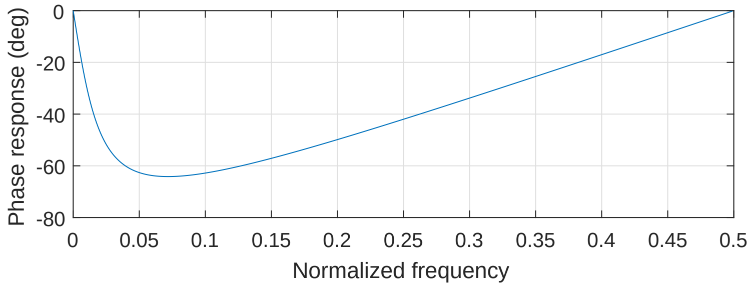
If the power is attenuated to half, the change in decibels is

$$10\log_{10} 0.5\ \text{dB} = -3.01\ \text{dB}.$$

In the previous example we had the system $h(n) = 0.9^n u(n)$. The amplitude response in decibel scale of that system is shown below. Now the frequency axis is normalized so that 0.5 corresponds to the Nyquist frequency.



The phase response is also typically shown only for the non-negative frequencies and the values are shown in degrees.



In Matlab similar figures can be easily plotted using the command `freqz`.

5.9 Transfer Function of IIR Filter

The z-transform is also a powerful tool for analyzing the IIR filters. Since the impulse response of the IIR filters has an infinite number of terms, the calculation of the transfer function directly using the definition is difficult. It can be more easily calculated by taking the z-transform directly from the difference equation of the IIR filter.

Consider the general form of the IIR filter:

$$y(n) = \sum_{k=0}^{K} a_k x(n-k) + \sum_{m=1}^{M} b_m y(n-m).$$

Using the linearity of the z-transform and the formula of the z-transform of the delayed signal, we obtain

$$Y(z) = \sum_{k=0}^{K} a_k X(z) z^{-k} + \sum_{m=1}^{M} b_m Y(z) z^{-m}.$$

Moving the latter sum to the left we can get all the $Y(z)$ terms on the left

$$Y(z)\left(1 - \sum_{m=1}^{M} b_m z^{-m}\right) = X(z) \sum_{k=0}^{K} a_k z^{-k},$$

which can be solved for the $Y(z)$:

$$Y(z) = \frac{\sum_{k=0}^{K} a_k z^{-k}}{1 - \sum_{m=1}^{M} b_m z^{-m}} X(z).$$

When we compare this with the z-transform of convolution $Y(z) = H(z)X(z)$, we notice that the transfer function is

$$H(z) = \frac{\sum_{k=0}^{K} a_k z^{-k}}{1 - \sum_{m=1}^{M} b_m z^{-m}}.$$

Example: Consider the filter defined by the differential equation

$$y(n) = 0.5x(n) - x(n-1) + 2x(n-2) - y(n-1) + y(n-2).$$

When we take the z-transforms of both sides of the equation we obtain

$$Y(z) = 0.5X(z) - X(z)z^{-1} + 2X(z)z^{-2} - Y(z)z^{-1} + Y(z)z^{-2}.$$

Further, by grouping, we obtain:

$$Y(z)(1 + z^{-1} - z^{-2}) = X(z)(0.5 - z^{-1} + 2z^{-2}).$$

Thus, the transfer function $H(z)$ is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.5 - z^{-1} + 2z^{-2}}{1 + z^{-1} - z^{-2}} = \frac{0.5z^2 - z + 2}{z^2 + z - 1}.$$

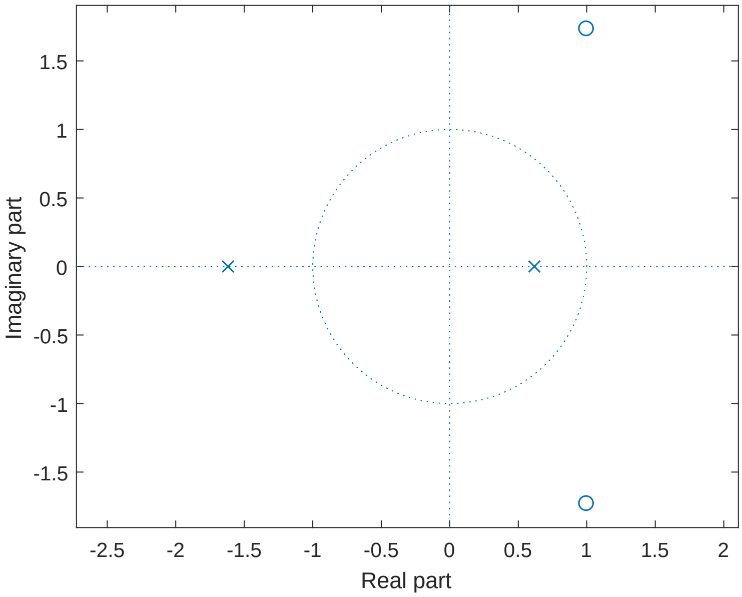
Let's find the poles and zeros of this function. The poles are the roots of the denominator, i.e.

$$\begin{aligned} p_1 &= \frac{-1-\sqrt{5}}{2}, \\ p_2 &= \frac{-1+\sqrt{5}}{2}. \end{aligned}$$

The zeros are the roots of the numerator, i.e.

$$\begin{aligned} z_1 &= 1+\sqrt{3}i, \\ z_2 &= 1-\sqrt{3}i. \end{aligned}$$

The pole-zero plot is plotted by Matlab command `zplane([0.5,-1,2],[1,1,-1]);`



From the amplitude response shown below, it can be deduced, that the system amplifies the Nyquist frequency by about 10 decibels. This means that at that frequency, the amplitude change a can be obtained from the equation

$$10 \text{ dB} = 20 \log_{10} a \text{ dB},$$

i.e.

$$\frac{1}{2} = \log_{10} a.$$

By raising 10 to these powers, we obtain

$$10^{\frac{1}{2}} = a,$$

i.e.

$$a = \sqrt{10} \approx 3.16.$$

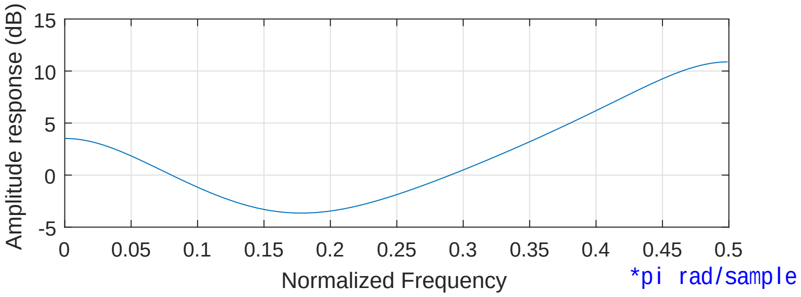
The amplitude of the signal thus increases to 3.16 times. If we want to know the change of the power P, it is obtained from the formula

$$10 \text{ dB} = 10 \log_{10} P \text{ dB},$$

which has the solution

$$P = 10.$$

The power thus increases tenfold.



Another common situation is the one where we know the transfer function and want to know the corresponding difference equation. Let the the transfer function be

$$H(z) = \frac{0.5z^2 + z + 0.25}{z^3 - z + 0.5}.$$

First, we change the powers of z from positive to negative, i.e. we multiply the numerator and denominator by z^{-3}

$$H(z) = \frac{0.5z^{-1} + z^{-2} + 0.25z^{-3}}{1 - z^{-2} + 0.5z^{-3}}.$$

Then we substitute $\frac{Y(z)}{X(z)} = H(z)$:

$$\frac{Y(z)}{X(z)} = \frac{0.5z^{-1} + z^{-2} + 0.25z^{-3}}{1 - z^{-2} + 0.5z^{-3}}.$$

We multiply this expression by the denominators:

$$Y(z)(1 - z^{-2} + 0.5z^{-3}) = X(z)(0.5z^{-1} + z^{-2} + 0.25z^{-3}),$$

i.e.

$$Y(z) - Y(z)z^{-2} + 0.5Y(z)z^{-3} = 0.5X(z)z^{-1} + X(z)z^{-2} + 0.25X(z)z^{-3}.$$

We know the signals corresponding to these terms:

$$y(n) - y(n-2) + 0.5y(n-3) = 0.5x(n-1) + x(n-2) + 0.25x(n-3).$$

Finally we move everything else except $y(n)$ to the right-hand side:

$$y(n) = 0.5x(n-1) + x(n-2) + 0.25x(n-3) + y(n-2) - 0.5y(n-3).$$

5.10 Stability

The stability of the IIR filter can easily be studied using the pole-zero plot. It can be shown that the (causal) IIR filter is stable if and only if all of its poles are within the unit circle.

For example, previously considered filter

$$y(n) = 0.5x(n) - x(n-1) + 2x(n-2) - y(n-1) + y(n-2)$$

is not stable because the poles are

$$p_{1,2} = \frac{-1 \pm \sqrt{5}}{2},$$

one of which is outside the unit circle ($|\frac{-1-\sqrt{5}}{2}| > 1$).

Let's take another example. For the filter

$$y(n) = x(n) - 2x(n-1) + 2x(n-2) + \frac{1}{2}y(n-1) - \frac{1}{8}y(n-2),$$

the transfer function is

$$\begin{aligned} & y(n) - \frac{1}{2}y(n-1) + \frac{1}{8}y(n-2) = x(n) - 2x(n-1) + 2x(n-2) \\ \Leftrightarrow\ & Y(z) - \frac{1}{2}Y(z)z^{-1} + \frac{1}{8}Y(z)z^{-2} = X(z) - 2X(z)z^{-1} + 2X(z)z^{-2} \\ \Leftrightarrow\ & Y(z)(1 - \frac{1}{2}z^{-1} + \frac{1}{8}z^{-2}) = X(z)(1 - 2z^{-1} + 2z^{-2}) \\ \Leftrightarrow\ & \frac{Y(z)}{X(z)} = \frac{1 - 2z^{-1} + 2z^{-2}}{1 - \frac{1}{2}z^{-1} + \frac{1}{8}z^{-2}} = \frac{z^2 - 2z + 2}{z^2 - \frac{1}{2}z + \frac{1}{8}} \qquad (= H(z)). \end{aligned}$$

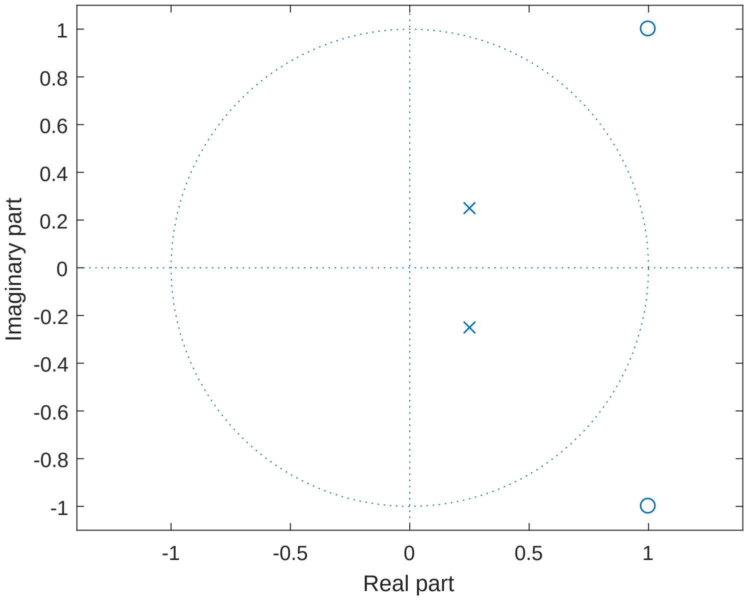
The poles are the roots of the polynomial

$$z^2 - \frac{1}{2}z + \frac{1}{8},$$

i.e.

$$p_{1,2} = \frac{\frac{1}{2} \pm \sqrt{\frac{1}{4} - 4 \cdot \frac{1}{8}}}{2} = \frac{\frac{1}{2} \pm \sqrt{-\frac{1}{4}}}{2} = \frac{\frac{1}{2} \pm i\sqrt{\frac{1}{4}}}{2} = \frac{\frac{1}{2} \pm \frac{i}{2}}{2} = \frac{1}{4} \pm \frac{i}{4}.$$

Both are within the unit circle ($|p_{1,2}| < 1$), so the filter is stable. The pole-zero plot of the system is shown below.



Chapter 6

FIR Filter Design

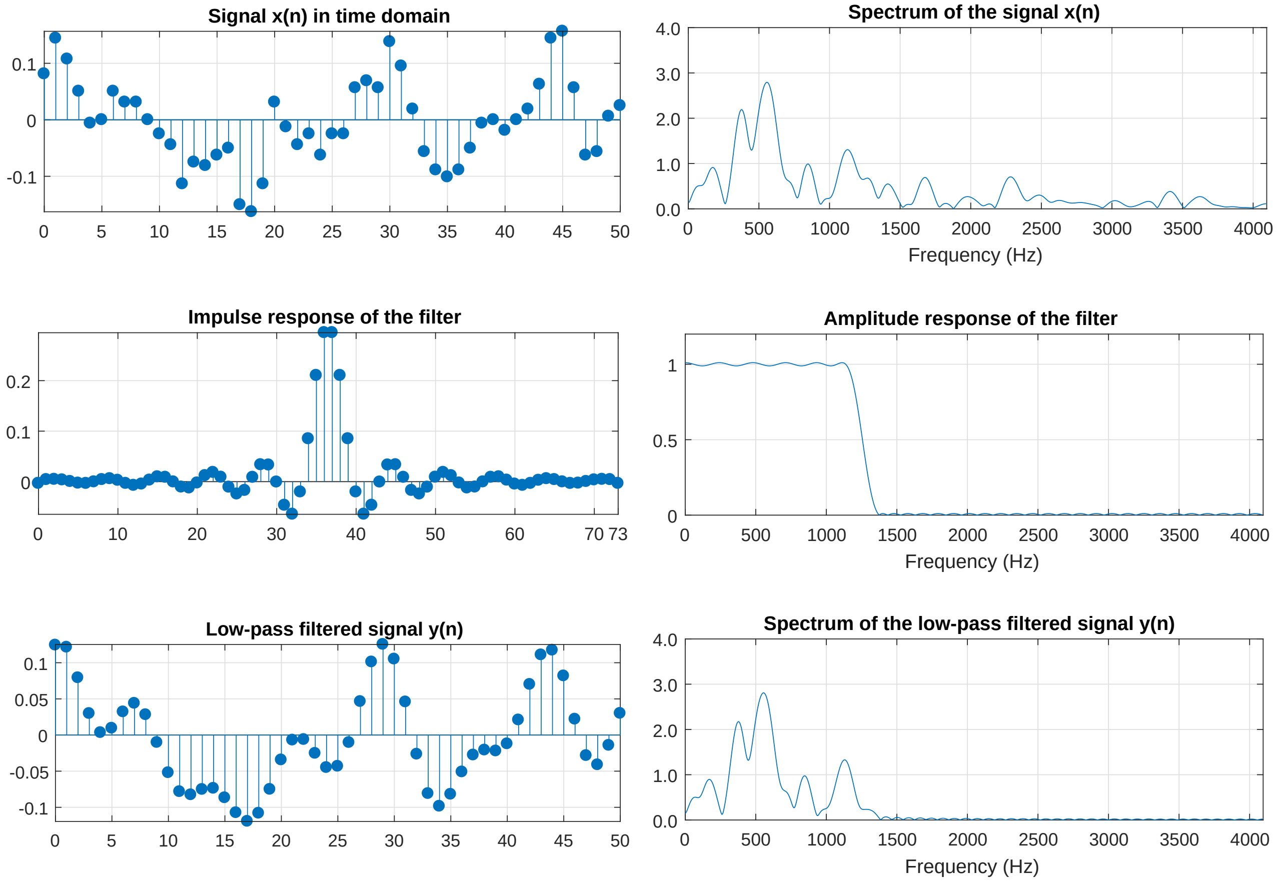
We have now seen that the FIR filter has a clear frequency domain interpretation. The filter changes the amplitude and phase of each frequency component, i.e. amplifies/attenuates and delays it. This determines the application of the FIR filter: the filter can attenuate certain frequencies and amplify others. A common case is a filter that completely eliminates a given frequency band and preserves the other band as it was. Frequency domain filtering applications are, for example, to remove some narrow bandwidth interference from the signal, to divide the signal into low and high frequencies for more efficient compression or to emphasize low and high frequencies, and thus compensate for distortion caused by cheap loudspeakers.

As we noticed when exploring the z-transform and the Fourier transform properties, the convolution in time domain corresponds to multiplication in z- and frequency domains. Conversely, it can be stated that the frequency domain multiplication can be implemented as time domain convolution. Convolution coefficients, i.e. the impulse response of the filter, must therefore be designed such that the frequency response is as desired. The design task could be, e.g., the following: find out the impulse response of a filter whose amplitude response is one for the frequencies 0 Hz – 1100 Hz and zero for the frequencies 1400 Hz – 4096 Hz, where 4096 Hz is the Nyquist frequency. Such a filter would preserve the frequencies up to 1100 Hertz as such and would remove the frequencies higher than 1400 Hertz. We will here explore one method to find suitable impulse response coefficients.

The frequencies in the signal $x(n)$ are revealed by its DTFT $X(e^{i\omega})$. When the signal is filtered by a filter having the frequency response $H(e^{i\omega})$, the frequency content of the output $y(n)$ is given by the equation

$$Y(e^{i\omega}) = H(e^{i\omega})X(e^{i\omega}).$$

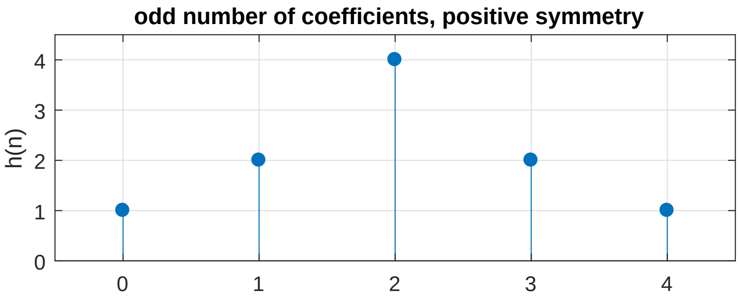
The illustration below shows the effect of filtering. The test signal is on the top left and its frequencies on the top right. In the middle row are the impulse response of a filter on the left and its amplitude response on the right. The filter is designed to meet the above requirements, i.e. to preserve the low frequencies and to remove the large ones. When the test signal is filtered with this filter, the convolution of the signal and the impulse response is calculated in the time domain which means multiplication in the frequency domain. In the convolution result (bottom row) the high frequencies are removed and the low frequencies remain unchanged. This is due to the amplitude response being close to zero at the high frequencies and close to one at low ones. Multiplication by zero eliminates and by one preserves the frequencies.

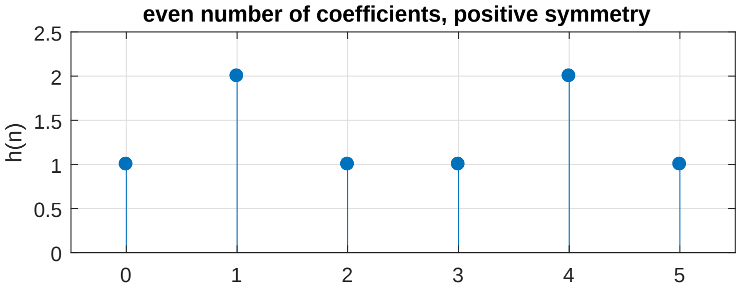


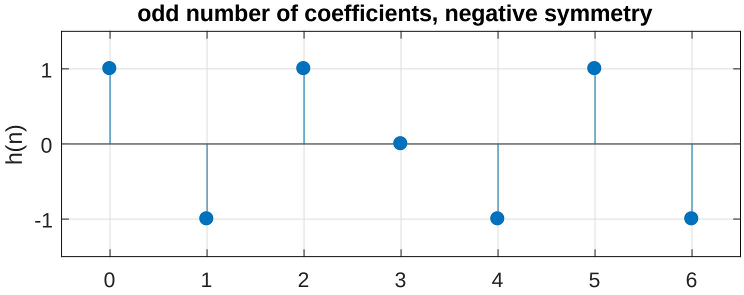
The specifications of the FIR filter requirements are usually presented in the frequency domain. The requirements can be divided into the amplitude response and phase response requirements.

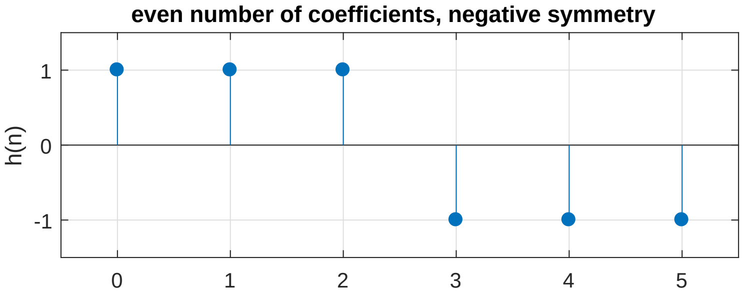
6.1 FIR Phase Response Requirements

The requirements for the phase response are usually simpler than for the amplitude response and a typical requirement is a linear phase response. If the phase response is linear, all frequencies are delayed by the same amount. This requirement can be easily realized by examining only the subclass of FIR filters with negatively or positively symmetric impulse responses (i.e. $h(n) = \pm h(N - n - 1)$). This will give us four different cases. The figures corresponding to these cases are below.


odd number of coefficients, positive symmetry


even number of coefficients, positive symmetry


odd number of coefficients, negative symmetry


even number of coefficients, negative symmetry

For example, the phase response of the filter in the first figure is calculated as follows. The impulse response of the filter is

$$h(n) = \delta(n) + 2\delta(n-1) + 4\delta(n-2) + 2\delta(n-3) + \delta(n-4).$$

The frequency response of this filter is

$$H(e^{i\omega}) = 1 + 2e^{-i\omega} + 4e^{-2i\omega} + 2e^{-3i\omega} + e^{-4i\omega},$$

and we simplify this by taking $e^{-2i\omega}$ as the common factor

$$\begin{aligned} H(e^{i\omega}) &= e^{-2i\omega}\left[e^{2i\omega} + 2e^{i\omega} + 4 + 2e^{-i\omega} + e^{-2i\omega}\right] \\ &= e^{-2i\omega}\left[(e^{2i\omega} + e^{-2i\omega}) + 2(e^{i\omega} + e^{-i\omega}) + 4\right] \\ &= e^{-2i\omega}\left[2\cos(2\omega) + 4\cos(\omega) + 4\right]. \end{aligned}$$

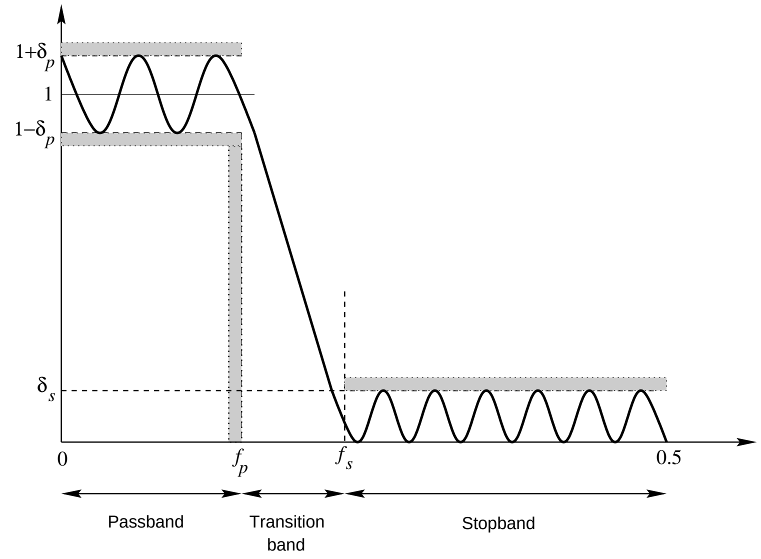
It is relatively easy to demonstrate that for the frequencies $\omega \in [0, \pi]$ the expression inside the square brackets is real and positive, so the phase response is the same as the phase angle of the complex term $e^{-2i\omega}$, i.e. -2ω.

Notice that the phase response indicates how many degrees or radians each frequency component is delayed. Thus the phase response should be of the form $-c\omega$ for each frequency component to be delayed the same amount in samples (or in time). *Group delay* is defined as $\tau(\omega) = -\frac{d\theta(\omega)}{d\omega}$, where $\theta(\omega)$ is the phase response at frequency ω and it tells how many samples the delay is.

The group delay can be used to define the concept of linear phase response in a general form: *phase response is linear if the group delay is constant at all points, where it is defined.* This definition also covers the filters having a phase response with a discontinuity point. The phase response is not defined in the case $H\left(e^{i\omega}\right) = 0$ since the phase angle of zero is undefined. These discontinuity points typically result in a jump of height π in the phase response.

6.2 FIR Amplitude Response Requirements

The requirements for the amplitude response indicate which frequencies are removed and which are preserved. These are presented using *passband, stopband* and *transition band.* Consider the following figure.



In the figure, the horizontal axis has the frequencies normalized with respect to the sampling rate. The vertical axis is the amplitude response of the filter. In this case, the filter is defined by the following parameters:

δ_p	Passband ripple,
δ_s	Stopband ripple,
f_p	Passband cut-off frequency,
f_s	Stopband cut-off frequency.

The passband consists of the frequencies $f \in [0, f_p]$, where we want the amplitude response to be close to one, stopband of the frequencies $f \in [f_s, 0.5]$ where we want the amplitude response to be close to zero, and the transition band of the frequencies $f \in (f_p, f_s)$. In the passband, the values of the amplitude response can deviate at most δ_p from the ideal value 1 and in the stopband at most δ_s from the ideal value 0.

Typical specifications could be e.g. the following:

δ_p	0.026 dB,
δ_s	-30 dB,
f_p	2500 Hz,
f_s	3000 Hz,
Sampling frequency	16000 Hz.

Conversion to normalized frequencies is done by dividing the frequencies f_p and f_s by the sampling frequency. The normalized requirements are

f_p	$2500/16000 = \frac{5}{32}$,
f_s	$3000/16000 = \frac{3}{16}$.

From this we obtain angular frequencies by scaling the values to interval $[0, \pi]$, i.e. by multiplying by value 2π:

ω_p	$2\pi \cdot 2500/16000 = \frac{5\pi}{16}$ rad,
ω_s	$2\pi \cdot 3000/16000 = \frac{3\pi}{8}$ rad.

6.3 Window Design Method

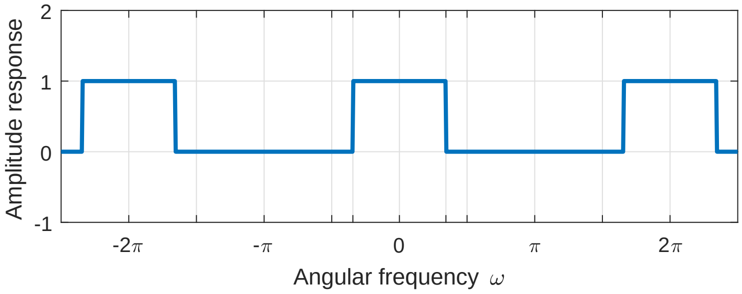
One way to satisfy the amplitude response requirements is simply to come up with an amplitude response graph that does this. However, this will easily create problems, as we will soon see. Consider the above conditions, which were

δ_p	0.026 dB,
δ_s	-30 dB,
ω_p	$2\pi \cdot 2500/16000 = \frac{5\pi}{16}$ rad,
ω_s	$2\pi \cdot 3000/16000 = \frac{3\pi}{8}$ rad.

These conditions are satisfied, for example, by an amplitude response that goes from one to zero in the midpoint of ω_p and ω_s, i.e., at angular frequency $\frac{11\pi}{32}$:

$$H(e^{i\omega}) = \begin{cases} 1, & \omega < \frac{11\pi}{32} \\ 0, & \omega \geq \frac{11\pi}{32}. \end{cases}$$

The figure below shows this amplitude response and also takes into account the response periodicity and the conjugate symmetry on the interval $[0, 2\pi]$.



It can be shown that in the general case the frequency response

$$H(e^{i\omega}) = \begin{cases} 1, & \text{when } \omega < \omega_c, \\ 0, & \text{when } \omega \geq \omega_c, \end{cases}$$

defining an *ideal low-pass filter* corresponds to the impulse response

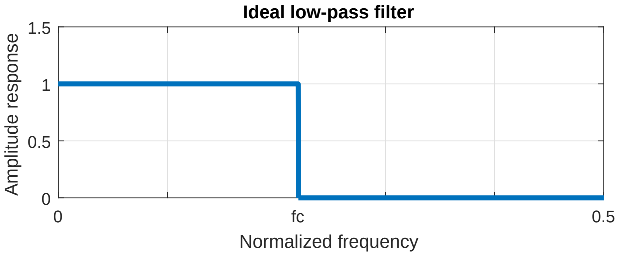
$$h(n) = \begin{cases} 2f_c \operatorname{sinc}(\omega_c n), & \text{when } n \neq 0, \\ 2f_c, & \text{when } n = 0, \end{cases}$$

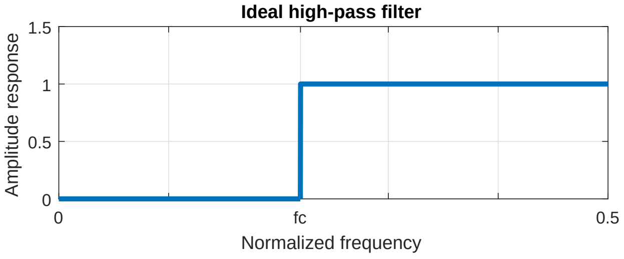
where the function $\operatorname{sinc}(x) = \sin(x)/x$. The problem in implementing this filter is the infinite length of the impulse response. In practice, the impulse response must always be truncated, which in turn causes problems in the frequency response. (We will study this soon in more detail.)

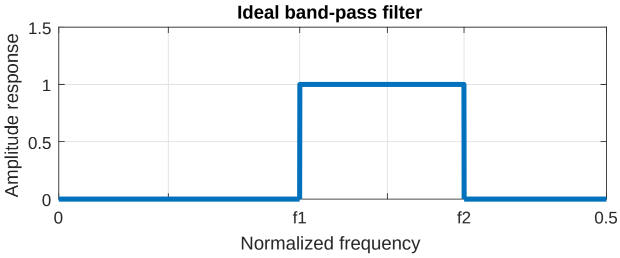
The following table shows ideal impulse responses for different filter types: *low-pass filters* (frequencies between $[0, f_c]$ are passed), *high-pass filters* (frequencies between $[f_c, 0.5]$ are passed), *band-pass filters* (frequencies between $[f_1, f_2]$ are passed) and *band-stop filters* (frequencies between $[f_1, f_2]$ are stopped).

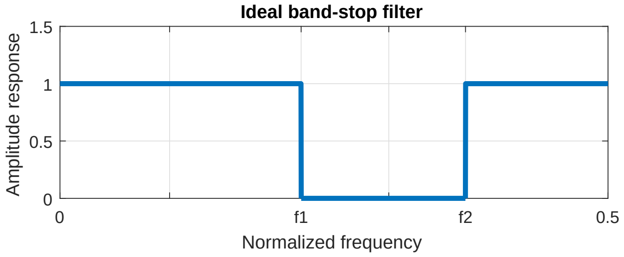
Ideal filter	Impulse response when	
type	$n \neq 0$	$n = 0$
Low-pass	$2f_c\text{sinc}(n \cdot 2\pi f_c)$	$2f_c$
High-pass	$-2f_c\text{sinc}(n \cdot 2\pi f_c)$	$1 - 2f_c$
Band-pass	$2f_2\text{sinc}(n \cdot 2\pi f_2) - 2f_1\text{sinc}(n \cdot 2\pi f_1)$	$2(f_2 - f_1)$
Band-stop	$2f_1\text{sinc}(n \cdot 2\pi f_1) - 2f_2\text{sinc}(n \cdot 2\pi f_2)$	$1 - 2(f_2 - f_1)$

The ideal amplitude responses of these filters are shown in the following figures.









Truncation of the infinitely long impulse response of the ideal filter is the most natural way to approximate it for real-world applications. The impulse response thus obtained corresponds to the ideal impulse response multiplied by the 'window signal'

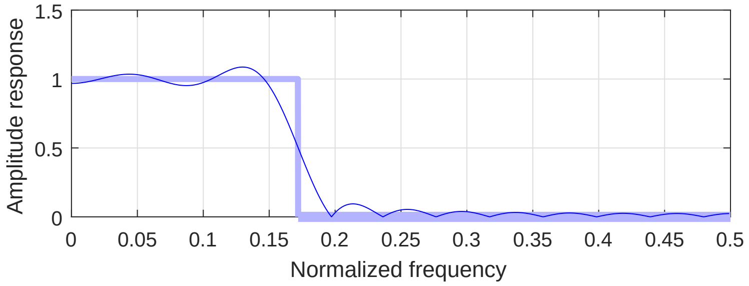
$$w(n) = \begin{cases} 1, & \text{when } -M < n < M, \\ 0, & \text{otherwise.} \end{cases}$$

This window function is called the *rectangular window*. The total number of window coefficients is denoted by N and it depends on the limit M by the formula $N = 2M + 1$. On the other hand, the limit M is obtained by dividing N by two and rounding down, i.e. $M = \lfloor \frac{N}{2} \rfloor$.

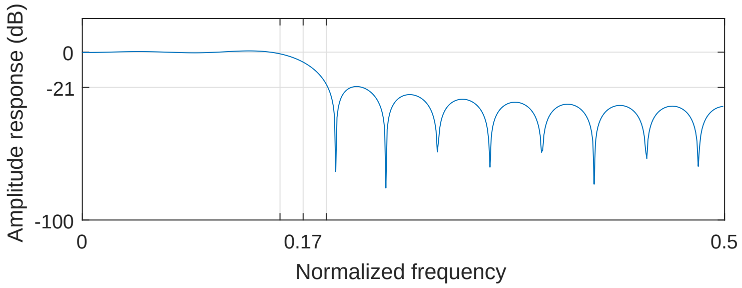
The behavior of the truncated impulse response $h_t(n) = w(n)h(n)$ in the frequency domain can be approximated by its DFT. The DFT of the product can be expressed by convolution:

$$H_t(n) = \frac{1}{N} W(n) * H(n).$$

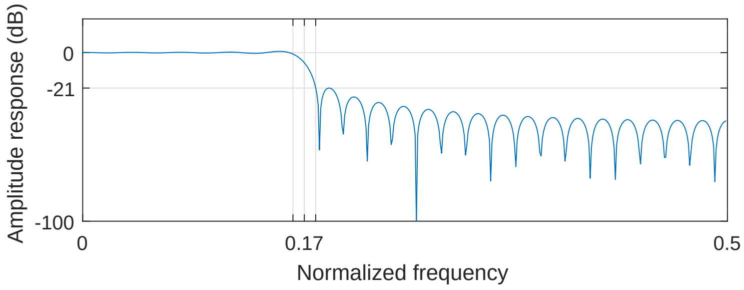
The effect of the truncation in the frequency domain is therefore convolution with the DFT of the window function $w(n)$. On the other hand, $W(n)$ is very similar to $h(n)$ and convolution with $W(n)$ thus causes oscillation in the resulting amplitude response and the Gibbs phenomenon, where oscillation is particularly strong near the edges of the frequency bands. We saw the Gibbs phenomenon already when studying the Fourier series. The figure below shows the amplitude response of the ideal low-pass filter after truncation of the impulse response. This is the above example where the cut-off frequency was $\omega_c = \frac{11\pi}{32}$ and as normalized $f_c = \frac{11}{64} \approx 0.17$. The impulse response consists of $N = 25$ terms, which are the ideal impulse response values at points $-12, -11, -10, \ldots, 10, 11, 12$.



The amplitude response shows oscillation around the cut-off frequency. This oscillation is particularly clear in the decibel scale, where the highest peak is -21 decibels.



By experimenting with impulse responses of different lengths, we notice that the increase in the length N does not improve the attenuation characteristics at all: the highest peak in the amplitude response in the stopband remains at -21 decibels. The amplitude response below is obtained by doubling the length of the impulse response to 51. The only effect is that the stopband peaks and the transition band get narrower.



Also, this does not influence the amount of passband oscillation, which is always constant and about 0.74 dB. Instead, the larger N narrows the transition band and for the rectangular window the connection between the width of the transition band Δf and the number of coefficients N is given by the formula

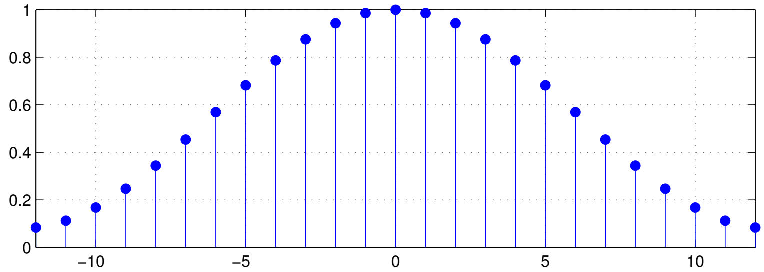
$$\Delta f = \frac{0.9}{N}.$$

We saw that the filter attenuation properties cannot be influenced by adding coefficients, but they can be improved by using a softer window function instead of the rectangular one.

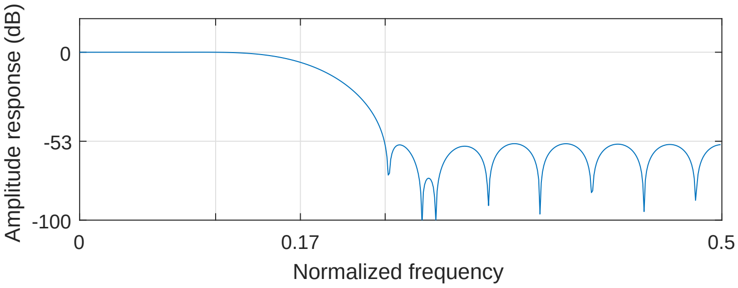
One of the frequently used window functions is the *Hamming window*, which is defined as follows:

$$w(n) = \begin{cases} 0.54 + 0.46\cos(\frac{2\pi n}{N}), & \text{when } -M \le n \le M, \\ 0, & \text{otherwise.} \end{cases}$$

The figure below shows the Hamming window coefficients in the case $N = 25$.



The Hamming window is used in such a way that the ideal impulse response is multiplied element-wise by the expression $w(n)$. The following figure shows the amplitude response when we are designing a low-pass filter that corresponds to the one we designed above using rectangular window.



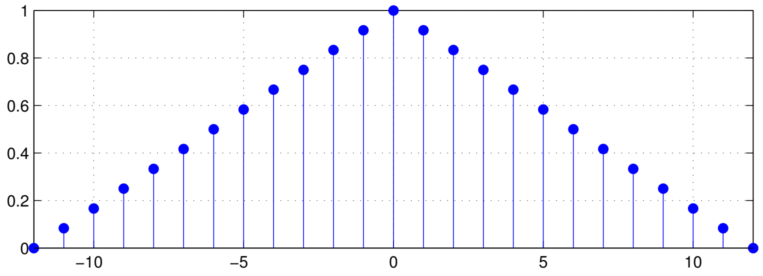
We can observe from the amplitude response that the Hamming window improves attenuation properties. Now the filter attenuates all frequencies in the stopband at least 53 dB. Moreover, the oscillation in the passband is no longer visible to the naked eye. By experimenting with different number of coefficients, the highest peak in the stopband is found to remain at 53 dB and the oscillation of the passband will remain in about 0.019 dB. Moreover, the width of the transition band is also in this case inversely proportional to the number of coefficients and it follows the formula

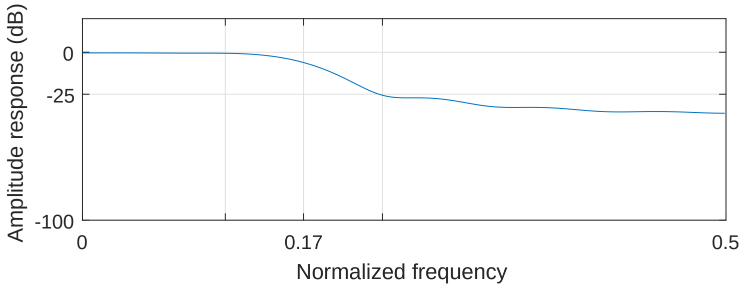
$$\Delta f = \frac{3.3}{N}.$$

Thus the transition band is clearly wider than the transition band obtained by using the corresponding rectangular window (which was $\Delta f = 0.9/N$). The choice of the window is therefore always a compromise between the attenuation characteristics and the number of coefficients. The Hamming window always needs larger amount of coefficients (approximately 3.7 times) in comparison with the rectangular window, for the transition bands to be equally wide.

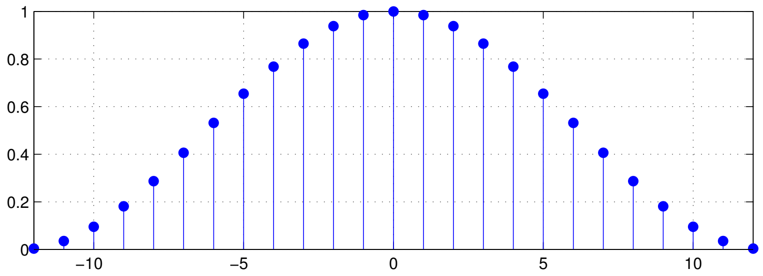
The Bartlett window, a triangular window function, is less commonly used in the filter design than the other windows considered here. Below are the Bartlett window coefficients

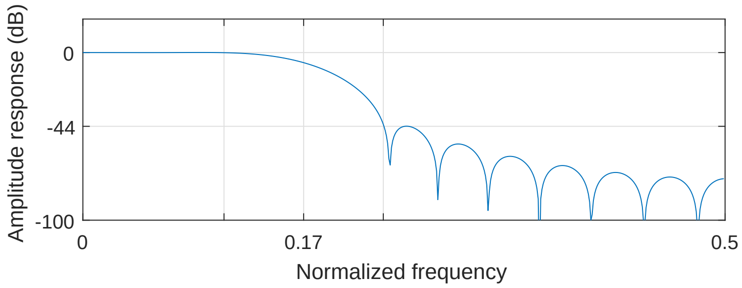
($N = 25$) and the amplitude response of a low-pass filter designed by using the Bartlett window.



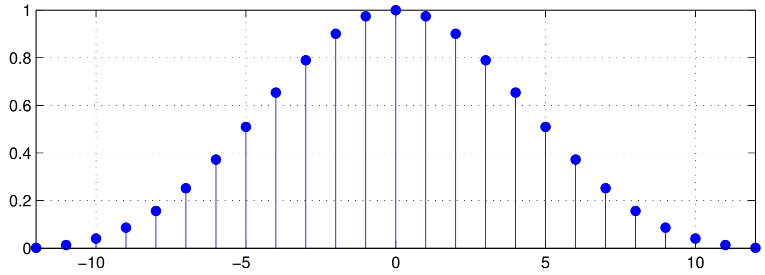


The Hanning (also called Hann) window is quite close to the Hamming window. However, its attenuation and ripple properties are slightly inferior but the number of coefficients on the other hand is smaller. Below are the Hanning window coefficients ($N = 25$) and the amplitude response of a low-pass filter designed by using the Hanning window.





The Blackman window consists of two cosine terms and has very good minimum stopband attenuation and passband ripple, but the required number of coefficients is higher. Below are the Blackman window coefficients ($N = 25$) and the amplitude response of a low-pass filter designed by using the Blackman window.



6.4 Summary of Window Design Method

This table can be used to calculate the number of required coefficients when a certain normalized transition bandwidth is desired. The better the attenuation and ripple properties of the window are, the higher the number in the formula for the transition band is, which means that more coefficients are needed to obtain the same transition bandwidth.

Name of the window function	Transition bandwidth (normalized)	Passband ripple (dB)	Minimum stopband attenuation (dB)	Window expression $w(n)$, when $\lvert n\rvert \leq (N-1)/2$
Rectangular	$0.9/N$	0.7416	21	1
Bartlett	$3.05/N$	0.4752	25	$1-\frac{2\lvert n\rvert}{N-1}$
Hanning	$3.1/N$	0.0546	44	$0.5+0.5\cos\left(\frac{2\pi n}{N}\right)$
Hamming	$3.3/N$	0.0194	53	$0.54+0.46\cos\left(\frac{2\pi n}{N}\right)$
Blackman	$5.5/N$	0.0017	74	$0.42+0.5\cos\left(\frac{2\pi n}{N}\right)+0.08\cos\left(\frac{4\pi n}{N}\right)$

The steps of the window design method are:

- Determine the ideal frequency response of the filter.
- Calculate the ideal impulse response corresponding to this frequency response using the inverse Fourier transform. The most common ideal impulse responses are listed in the table on page 86.
- Choose a window function that meets the desired requirements on the passband and the stopband. Determine also the number of required coefficients N from the normalized transition bandwidth Δf.
- Impulse response of the designed filter is

$$h_t(n) = w(n)h(n),$$

where $h(n)$ is the ideal impulse response and $w(n)$ is the selected window function.

Example: Let's use the same requirements for a low-pass filter that we had in the previous example:

δ_p	0.026 dB,
δ_s	-30 dB,
f_p	5000 Hz,
f_s	6000 Hz,
Sampling frequency	16000 Hz.

Since $\delta_p = 0.026$ dB, the only possible windows are the Hamming window ($\delta_p = 0.0194$ dB) and the Blackman window ($\delta_p = 0.0017$ dB). Of these, we choose the Hamming window, since it requires smaller number of coefficients than the Blackman window:

$$N = \begin{cases} \frac{3.3}{\Delta f}, & \text{Hamming window,} \\ \frac{5.5}{\Delta f}, & \text{Blackman window.} \end{cases}$$

The table also shows that by using the Hamming window the given stopband requirements are met. The column *Minimum stopband attenuation* shows that the minimum attenuation is 53 dB when only $\delta_s = -30$ dB (-53 dB < -30 dB) was required. Next, we need to find out the number of filter coefficients. It is obtained from the transition bandwidth. In our case, the transition band should be between 5000-6000 Hz. When normalized by the sampling rate, this is the interval $[5000/16000, 6000/16000]$ so the normalized width of the transition band is $\Delta f = 1000/16000 = 1/16$.

When using the Hamming window

$$\Delta f = \frac{3.3}{N},$$

then the number of required coefficients N is (by using exact value in calculations)

$$N = \frac{3.3}{\Delta f} = \frac{3.3}{1/16} = 3.3 \cdot 16 = 52.8.$$

This is rounded up to $N = 53$. In this course, the rounding of the filter length is always up to the nearest *odd* integer.

The ideal impulse response is in the case of a low-pass filter

$$h(n) = \begin{cases} 2f_c \operatorname{sinc}(n \cdot 2\pi f_c), & n \neq 0, \\ 2f_c, & n = 0, \end{cases}$$

that is, in our case

$$h(n) = \begin{cases} 2 \cdot 5500/16000 \operatorname{sinc}(2\pi \cdot 5500/16000 \cdot n), & n \neq 0, \\ 2 \cdot 5500/16000, & n = 0. \end{cases}$$

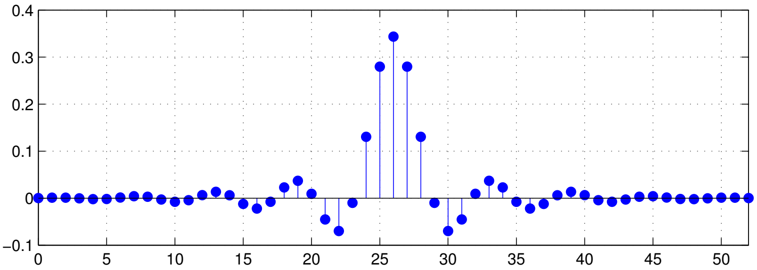
Here the number, $f_c = \frac{5500}{16000} = \frac{11}{32}$ is selected to be the midpoint of the transition band. The actual impulse response is now obtained by multiplying this $h(n)$ by the Hamming window

$$h_t(n) = \begin{cases} (0.54 + 0.46\cos(2\pi n/53)) \cdot 2 \cdot 11/32 \cdot \operatorname{sinc}(2\pi \cdot 11/32 \cdot n), & 0 < |n| \leq 26, \\ 2 \cdot 11/32, & n = 0, \\ 0, & \text{otherwise,} \end{cases}$$

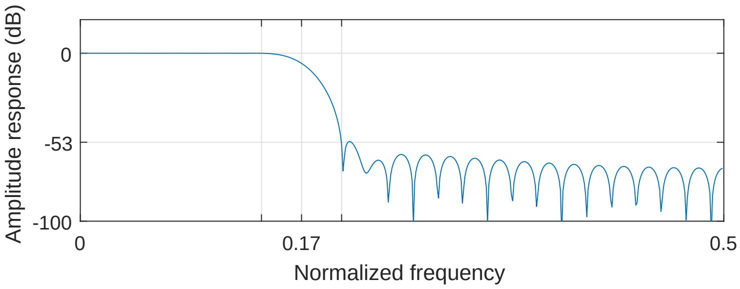
which simplifies to the form

$$h_t(n) = \begin{cases} \frac{11}{16} \cdot (0.54 + 0.46\cos(\frac{2\pi n}{53})) \cdot \operatorname{sinc}(\frac{11\pi n}{16}), & 0 < |n| \leq 26, \\ 11/16, & n = 0, \\ 0, & \text{otherwise.} \end{cases}$$

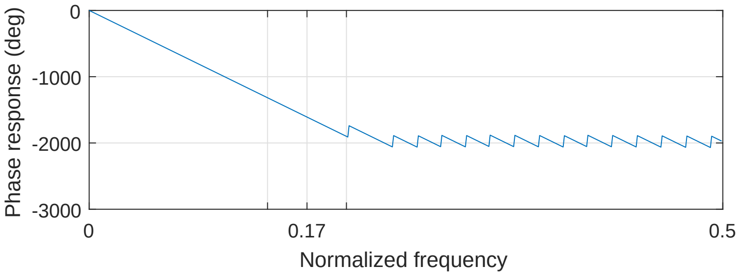
In order to have a causal filter, we have to move this impulse response 26 steps to the right (delay it). This does not affect the result other than delays it by 26 steps, the frequency domain behavior remains unchanged. The impulse response thus obtained is below.



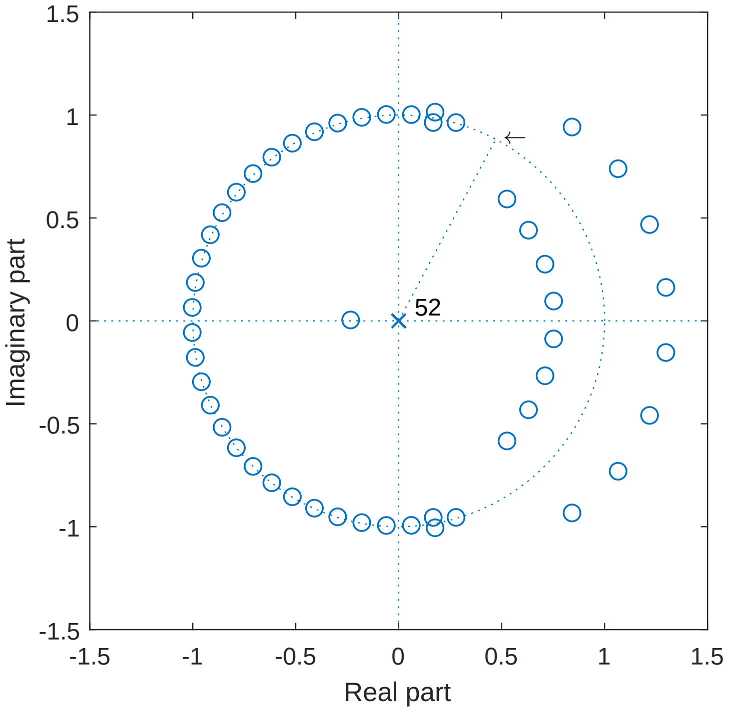
The amplitude response of this filter is as follows.



The phase response is below. Note the linearity in the pass band. The group delay is $\tau(\omega) = 26$, so the filter will delay all frequencies by 26 steps.



The pole-zero plot is below. The farthest zero at $z = 4.37$ is not shown in this figure. The figure clearly shows how the frequency axis of the Fourier transform corresponds to the upper half of the unit circle, i.e. the points $\{e^{i\omega} \mid \omega \in [0, \pi]\}$. The frequency response at frequency ω is obtained by evaluating the transfer function $H(z)$ for $z = e^{i\omega}$. The figure shows by the dashed line the phase angle $\omega_c = \frac{11\pi}{32}$ corresponding to the transition band midpoint, and the frequency response at this frequency is obtained by evaluating the transfer function at the point indicated by the arrow.



The frequencies lower than ω_c can be obtained by rotating the unit circle clockwise from ω_c and higher by rotating counterclockwise. The zero frequency is at the complex plane point $1+0i$ and the Nyquist frequency at $-1+0i$. Thus, all the zeros of the transfer function that are located on the unit circle have larger phase angle than $\omega_c = \frac{11\pi}{32}$ and each of them causes a downward spike in the amplitude response.

Example: As another example, we design a high-pass filter with the following requirements:

δ_p	0.06 dB,
δ_s	-28 dB,
f_s	9 kHz,
f_p	10 kHz,
Sampling frequency	48 kHz.

The table shows that the rectangular and Bartlett windows are not satisfying the requirements, and the Hanning window is the first one to satisfy the passband (0.0546 dB $<$ 0.06 dB) and the stopband (-44 dB $<$ -28 dB) requirements. Because the Hanning window requires less coefficients than the other suitable ones, we choose it. Then

$$\Delta f = \frac{3.1}{N}.$$

The transition bandwidth normalized by the sampling rate is

$$\Delta f = \frac{10 \text{ kHz}}{48 \text{ kHz}} - \frac{9 \text{ kHz}}{48 \text{ kHz}} = \frac{1}{48}.$$

The number of required coefficients is

$$N = \frac{3.1}{\Delta f} = \frac{3.1}{1/48} = 148.8,$$

which is rounded to $N = 149$. We choose f_c to be the midpoint of the normalized transition band $[9/48, 10/48]$ i.e. $f_c = 9.5/48$. Then we obtain

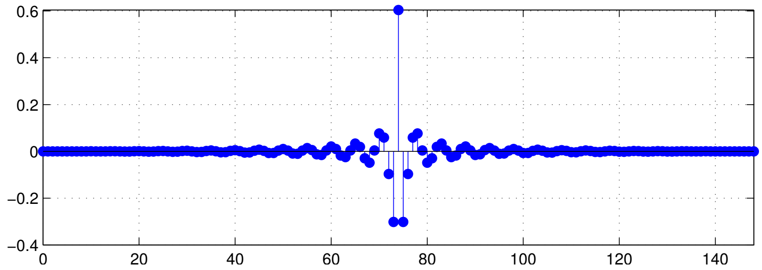
$$h(n) = \begin{cases} -2 \cdot \frac{9.5}{48}\mathrm{sinc}(2\pi \cdot \frac{9.5}{48}n), & n \neq 0, \\ 1 - 2 \cdot \frac{9.5}{48}, & n = 0. \end{cases}$$

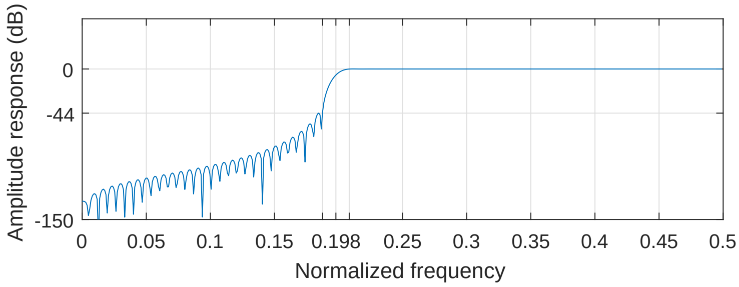
This must still be multiplied by the Hanning window, whereby the actual impulse response in simplified form is

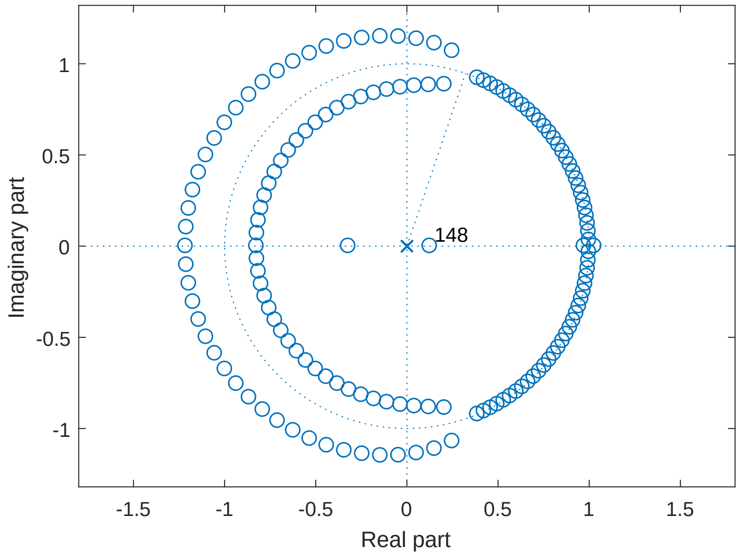
$$h_t(n) = \begin{cases} -(0.5 + 0.5\cos(\frac{2\pi n}{149}))(\frac{19}{48}\mathrm{sinc}(\frac{19}{48}\pi n)), & 0 < |n| \leq 74, \\ \frac{29}{48}, & n = 0, \\ 0, & \text{otherwise.} \end{cases}$$

Again, this impulse response must be shifted 74 steps to the right to produce a causal filter.

The impulse and amplitude responses and the pole-zero plot are below. The farthest zeros at $z = 7.9116$ and $z = -3.1176$ are not shown in the pole-zero plot.







Chapter 7

IIR Filter Design

The conventional method of designing the IIR filters is to first design the corresponding analog filter and then convert it to the digital format. The conversion can be done using certain mathematical techniques such as the bilinear transform, impulse invariance, or pole-zero matching method. We do not study these design methods in this course (with the exception of an exercise task providing a very brief overview of the bilinear transform design method). Instead, we examine how IIR filters can be designed using Matlab.

We compare four classical IIR filter types:

- Butterworth filter
- Type I Chebyshev filter
- Type II Chebyshev filter
- Elliptic filter

7.1 Butterworth IIR Filter

The Butterworth type IIR filters are characterized by monotonic amplitude response in both the passband and the stopband. Additionally, the beginning of the passband and the end of the stopband (for a low-pass filter) are both maximally flat. Two Matlab commands are needed in designing the Butterworth type IIR filter. The first one calculates the required filter order `N` and cutoff frequencies `Wn` of the analog prototype filter. When designing a low-pass filter, the command has the form

```
[N, Wn] = buttord(2*Wp, 2*Ws, Rp, Rs);
```

where the input parameters are

- `Wp` is the passband cut-off frequency,
- `Ws` is the stopband cut-off frequency,
- `Rp` is the passband ripple in decibels,
- `Rs` is the minimum stopband attenuation in decibels.

The first two parameters `Wp` and `Ws` have multiplier 2 since Matlab normalizes the frequencies to interval $[0, 1]$ and in this course they are normalized to $[0, 0.5]$.

The second command designs the digital filter

```
[a,b] = butter (N, Wn);
```

The same command is also used for designing high-pass filters. In that case, the string 'high' is added as the last argument of the command, i.e.

```
[a,b] = butter (N, Wn, 'high');
```

If we want band-pass or band-stop filters, the arguments Ws and Wp are replaced by the vectors [W1, W2] and [W3, W4] indicating the edge frequencies of the transition bands. By adding the string 'stop' as the last argument, we obtain a band-stop filter, otherwise a band-pass filter will be obtained.

In all cases, two resulting vectors contain the transfer function coefficients: `a`$= (a_0, a_1, \ldots, a_N)$ and `b`$= (1, b_1, \ldots, b_N)$. The first one has the numerator and the latter the denominator coefficients:

$$\frac{Y(z)}{X(z)} = H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}}{1 - (b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N})}.$$

This results in the actual filter coefficients when both sides of the equation are multiplied by $X(z) \cdot \left(1 - \sum_{k=1}^{N} b_k z^{-k}\right)$

$$Y(z) \cdot \left(1 - \sum_{k=1}^{N} b_k z^{-k}\right) = X(z) \sum_{k=0}^{N} a_k z^{-k}$$

and further by the inverse z transform

$$y(n) - \sum_{k=1}^{N} b_k y(n-k) = \sum_{k=0}^{N} a_k x(n-k).$$

This can be expressed in the form

$$y(n) = \sum_{k=0}^{N} a_k x(n-k) + \sum_{k=1}^{N} b_k y(n-k).$$

Example: We design a low-pass filter with the passband between $[0, 0.11]$ (in normalized frequencies), the stopband between $[0.145, 0.5]$, the passband ripple 0.01 dB and the minimum stopband attenuation 47 dB.

```
[N, Wn] = buttord (0.22, 0.29, 0.01, 47)
```

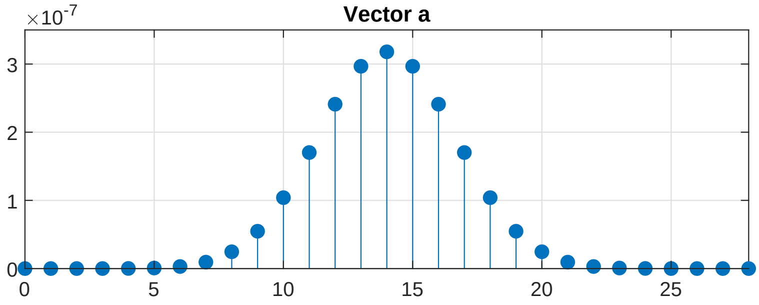
This command returns the values:

```
N = 28
Wn = 0.2443
```

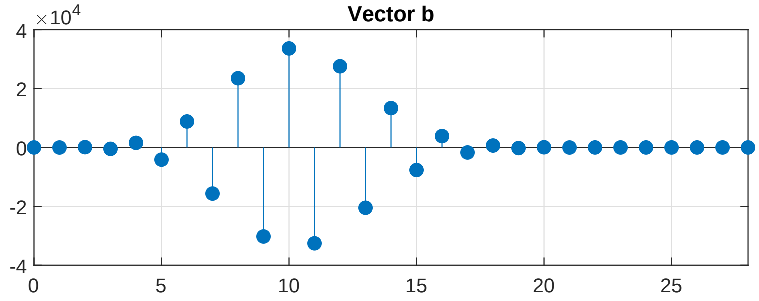
The actual filter design is done with the command

```
[a,b] = butter(N,Wn);
```

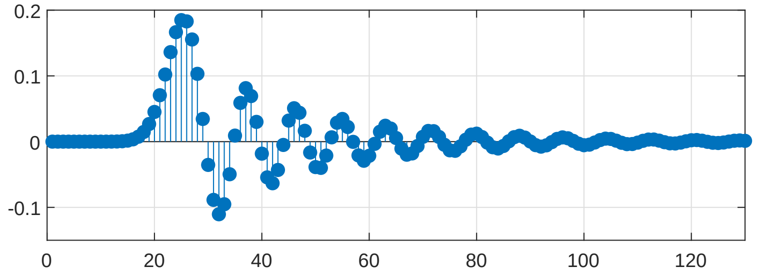
This results in two vectors: the vector `a`



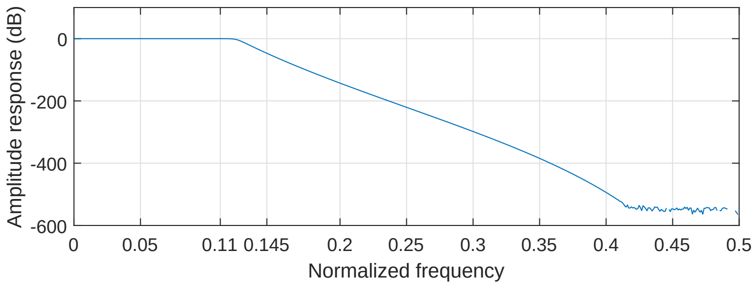
and the vector `b` (Note that `b` is given in Matlab notation. In our notation it corresponds to `-b`.):



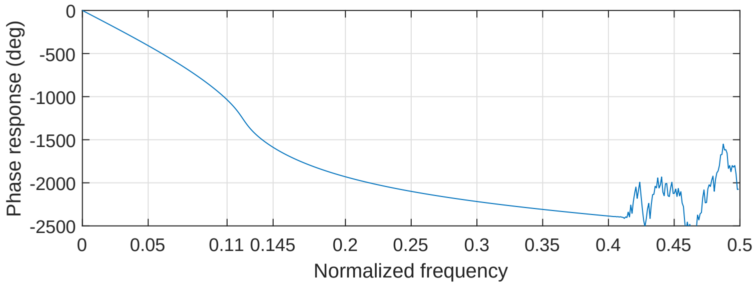
The beginning of the impulse response of the designed filter is obtained by the command `impz(a,b)`.



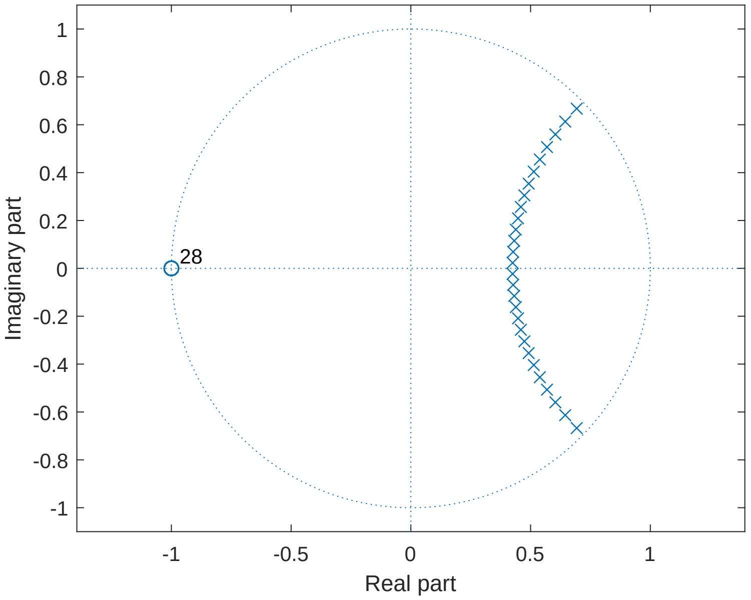
The amplitude response (`freqz(a,b)`) is shown in the following figure. Note the magnificent attenuation characteristics of the Butterworth filter at the end of the stopband. The attenuation approaches $-\infty$ dB when approaching the normalized frequency of 0.5. Clearly Matlab does not have high enough precision when $|H(e^{i\omega})|$ is less than -500 dB (on a linear scale the numbers are of the order 10^{-25}).



The phase response is fairly linear in the passband and therefore suitable for most applications. The plot is below.



Finally, the poles and zeros:



The figure shows that all the zeros are at $z = -1$, which corresponds to the Nyquist frequency. Thus, the amplitude response of the filter at this frequency is zero on the linear scale, i.e. in decibels the amplitude response approaches $-\infty$. From the graphs of the amplitude and phase response, we can see that precision of Matlab's calculations is not sufficient in the vicinity of the Nyquist frequency. For the same reason, the `zplane` command cannot calculate the pole-zero plot of this filter correctly. The above plot is obtained by using the command `butter` with three output values:

```
[z,p,K] = butter(N,Wn);
```

Then the zeros are in the column vector `z`, the poles in the vector `p` and the amplification at the zero frequency (so-called *gain*) in the variable `K`. The resulting poles and zeros are correct because the Butterworth filter design algorithm calculates them first and then converts the result into the filter coefficients (see Exercise 10, Task 3).

7.2 Type I Chebyshev IIR Filter

The Chebyshev filters of the first type are characterized by maximally flat stopband, but the passband exhibits equiripple behavior. The filter design commands are similar to those of the Butterworth filters. The order of the filter can be estimated by command

```
[N, Wn] = cheb1ord(2*Wp, 2*Ws, Rp, Rs);
```

The filter design command is of the form

```
[a,b] = cheby1(N,Rp,Wn);
```

Other than the low-pass filters are designed in the same way as in the case of the Butterworth filter. Consider the type I Chebyshev filter corresponding to the Butterworth filter in the previous example:

```
[N, Wn] = cheb1ord(0.22, 0.29, 0.01, 47)
```

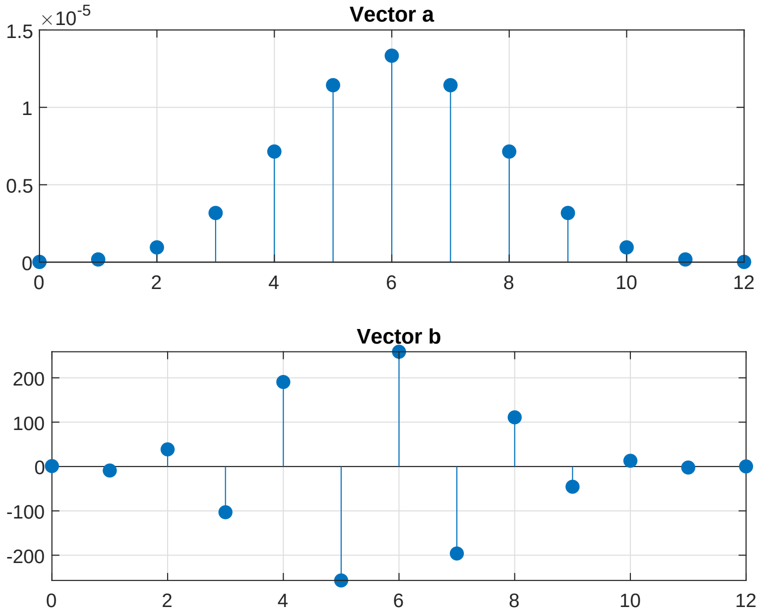
The command returns the values:

```
N = 12
Wn = 0.2200
```

The filter design is done with the command:

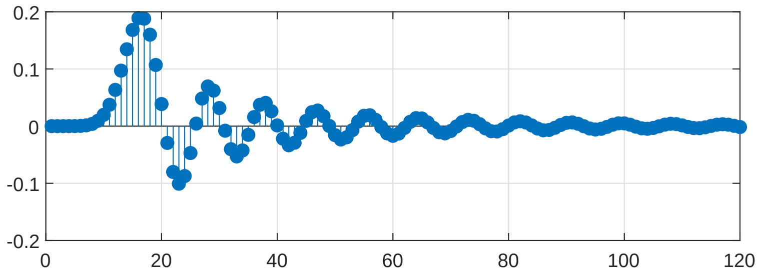
```
[a,b] = cheby1 (N, 0.01, Wn);
```

The coefficients of the obtained filter are in the vectors `a` and `b` shown below.

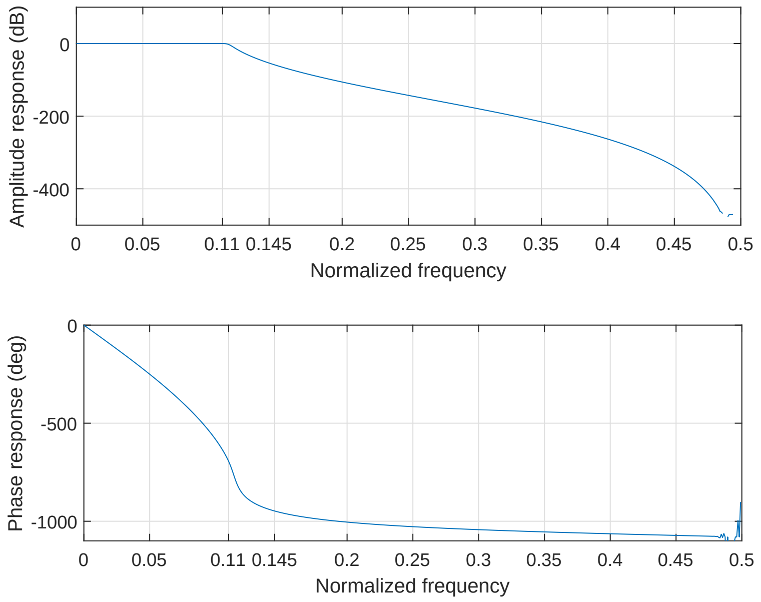

Vector a

Vector b

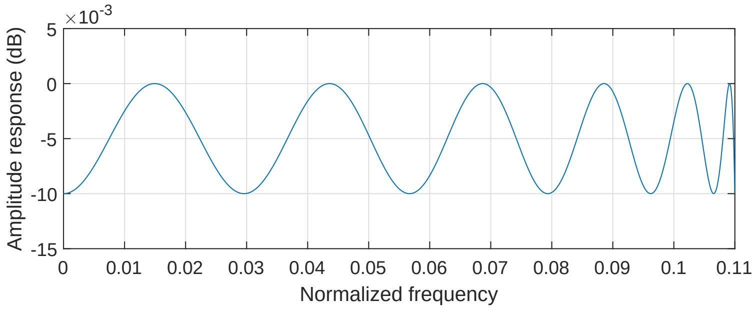
The beginning of the filter impulse response is shown in the figure below.



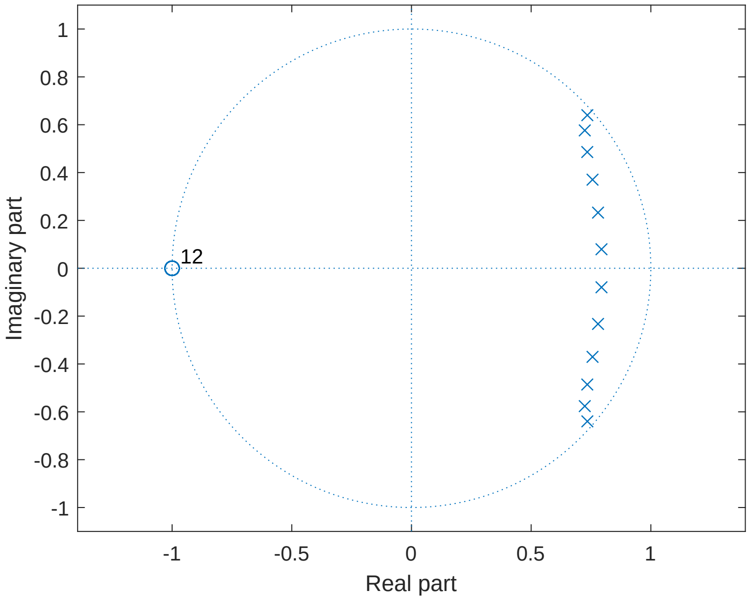
The amplitude and phase response of the filter are as follows:



From the amplitude response it is seen that the filter is similar to the Butterworth filter in the stopband: the amplitude response approaches infinite decibels attenuation when the frequency approaches the Nyquist frequency. However, the difference between these two filters is the behavior in the passband. The amplitude response of the Butterworth filter is a flat descending function but the amplitude response of type I Chebyshev filter oscillates below the ideal value (see figure below).



Again, `zplane` calculates the filter poles and zeros incorrectly and they are obtained from the command `cheb1` similarly as in the Butterworth filter case. The pole-zero plot of the filter is below.



7.3 Type II Chebyshev IIR Filter

Chebyshev filters of the second type are characterized by their equiripple stopband, but the passband is maximally flat. Compared to type I Chebyshev filters, the passband and stopband types are reversed. Similar commands as in the previous types are available to design this filter type. The order of the filter can be evaluated by the command

```
[N, Wn] = cheb2ord(2*Wp, 2*Ws, Rp, Rs);
```

The filter design command is

```
[a,b] = cheby2(N,Rs,Wn);
```

The type II Chebyshev filter satisfying the requirements of the previous example is designed as follows:

```
[N, Wn] = cheb2ord(0.22, 0.29, 0.01, 47)
```

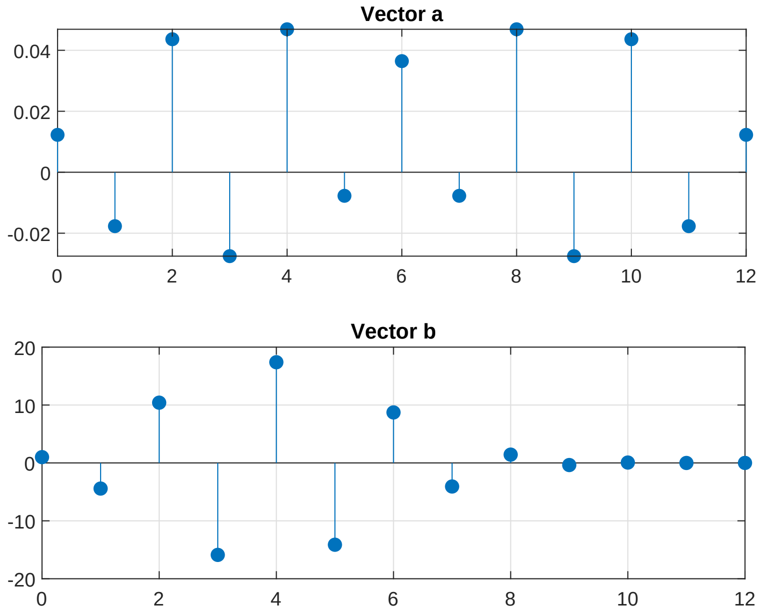
This gives the values:

```
N = 12
Wn = 0.2795
```

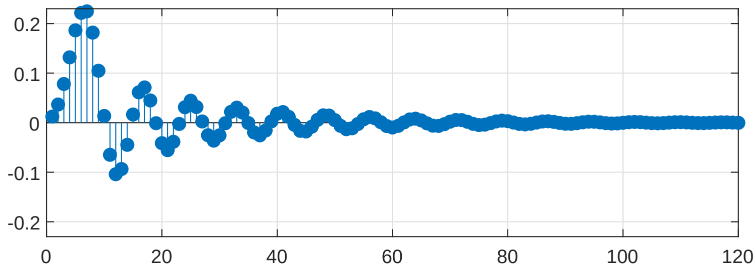
The actual filter design is done with the command

```
[a,b] = cheby2 (N, 47, Wn);
```

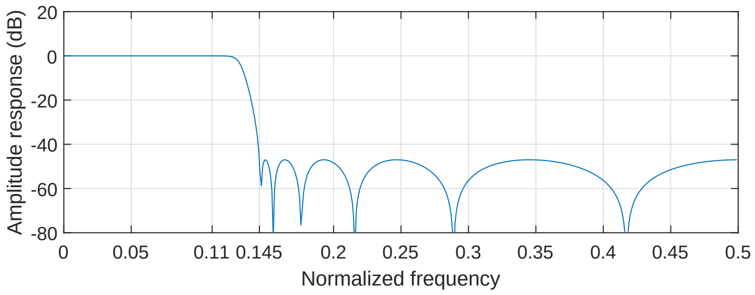
The resulting filter coefficients (vectors `a` and `b`) are shown below.

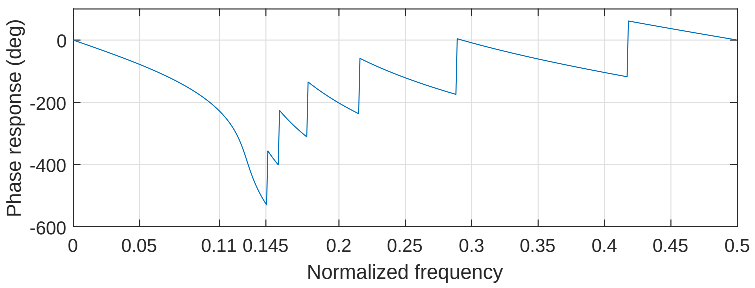


The beginning of the filter impulse response looks very similar to the other filter types.

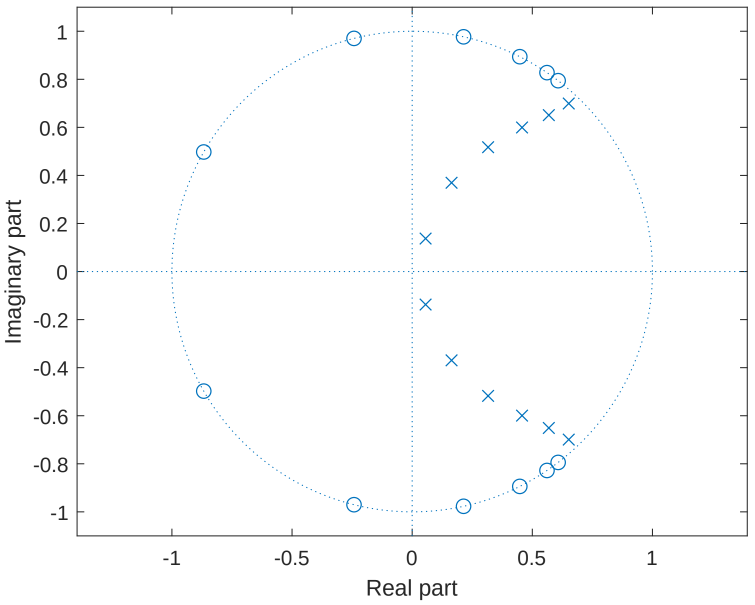


However, the amplitude and phase responses show a clear distinction compared to Butterworth and Type I Chebyshev filters. Now the stopband is equiripple, which means all the peaks in there are at the same height. On the other hand, the passband is similar to the Butterworth filter, i.e. it decreases monotonously as the frequency increases.





The pole-zero plot below partially explains the equiripple characteristic on the stopband. The zeros are located in the stopband with such intervals that their joint influence equilibrates the oscillations in the stopband.



7.4 Elliptic IIR Filters

Elliptic filters (also known as Cauer filters) are characterized by equiripple amplitude response *both* in the passband *and* the stopband. The filter design commands are

```
[N, Wn] = ellipord(2*Wp, 2*Ws, Rp, Rs);
```

and

```
[a,b] = ellip(N,Rp,Rs,Wn);
```

The design of our example filter is then made by commands

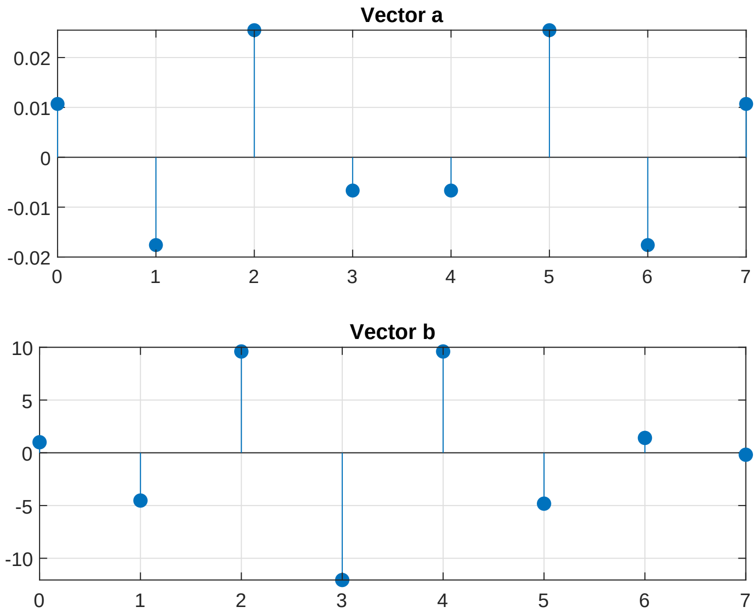
```
[N,Wn] = ellipord(0.22, 0.29, 0.01, 47)
```

The result is:

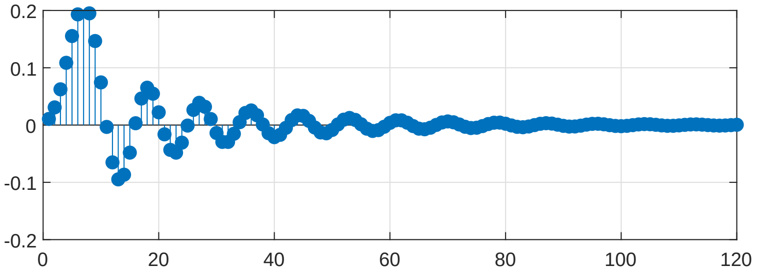
```
N = 7
Wn = 0.2200
```

Next, the filter design command:

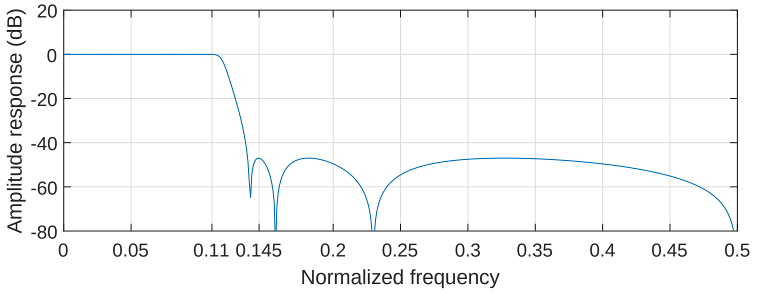
`[a,b] = ellip (N, 0.01, 47, Wn);` The resulting filter coefficients (vectors `a` and `b`) are below.

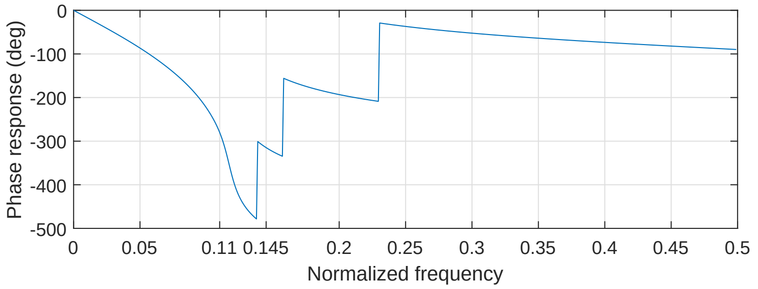


The beginning of the filter impulse response looks again very similar to the other filter types.

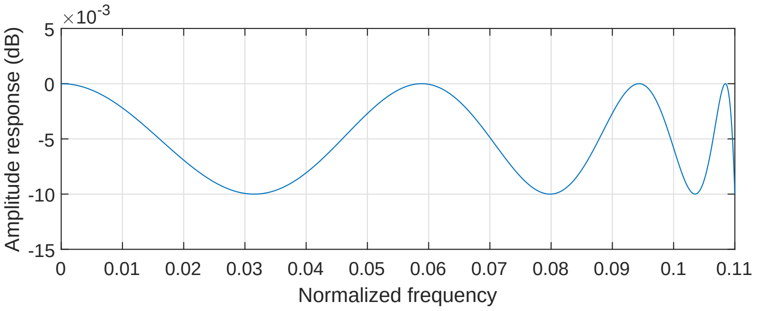


The amplitude and phase response of the filter are shown below.

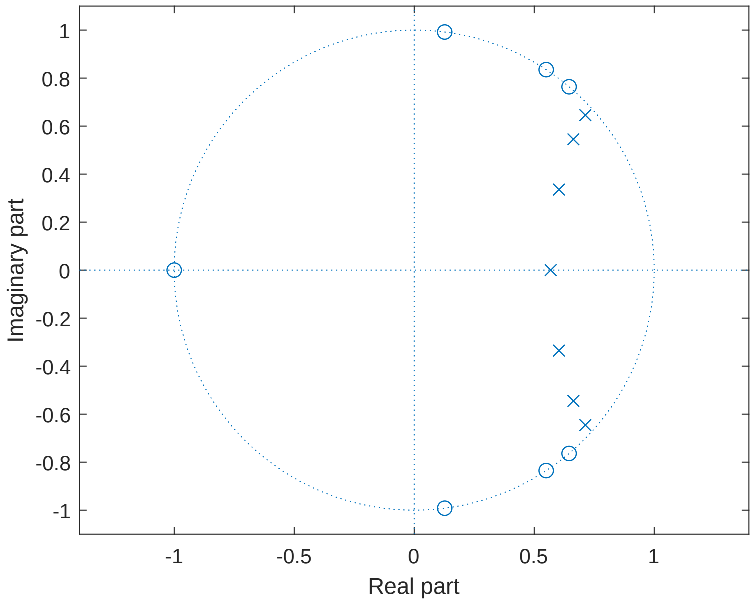




The amplitude response of the filter oscillates below the ideal value (see figure below).



The pole-zero plot of the filter is below.



7.5 Comparison of IIR Filter Types

The table below compares the four IIR filter types and the number of coefficients in our example case. The number of coefficients for the two Chebyshev filter types is always the same, Butterworth filter always has the highest number of coefficients and the elliptic filter the smallest. For comparison, the right-hand column has the number of coefficients of the corresponding FIR filter designed by the window design method: the number is clearly much higher than the IIR filter coefficient numbers.

	Butterworth	*Chebyshev I*	*Chebyshev II*	*Elliptic*	*Blackman*
Ripple on passband	no	yes	no	yes	yes
Ripple on stopband	no	no	yes	yes	yes
Number of coefficients	$29 + 28$	$13 + 12$	$13 + 12$	$8 + 7$	159

7.6 IIR Filter Implementation

In practice it is not possible to implement an IIR filter of an order higher than 2. Therefore, higher order filters should be divided into second order sections. There is a command for doing this in Matlab: `tf2sos`.

Consider the 4th order filter:

$$H(z) = \frac{0.5702 - 2.2627z^{-1} + 3.3851z^{-2} - 2.2627z^{-3} + 0.5702z^{-4}}{1 - 2.8767z^{-1} + 3.2538z^{-2} - 1.6844z^{-3} + 0.341z^{-4}}.$$

This is divided into 2nd order sections as follows.

```
   >> tf2sos(a,b)
ans =
0.5702 -1.1377 0.5702 1.0000 -1.1851 0.4031
1.0000 -1.9730 1.0000 1.0000 -1.6916 0.8460
```

The resulting matrix represents two 2nd order filters:

$$\begin{aligned} H_1(z) &= \frac{0.5702 - 1.1377z^{-1} + 0.5702z^{-2}}{1 - 1.1851z^{-1} + 0.4031z^{-2}} \\ H_2(z) &= \frac{1 - 1.9730z^{-1} + z^{-2}}{1 - 1.6916z^{-1} + 0.8460z^{-2}}. \end{aligned}$$

Chapter 8

Multirate DSP

In multirate DSP the sampling rate of a digital signal is changed in order to increase the efficiency of various signal processing operations.

Of course, the most simple method for converting the sampling rate is to convert the signal first to analog and then convert it back to digital at a new sampling rate. However, this procedure leads to additional quantization or rounding errors, which should be avoided. It is possible to solve the problem solely by means of DSP. This gives you a guaranteed optimum result that does not involve additional quantization noise.

The problem: There is a signal formed from an analog signal at the sampling rate f and we want to obtain a digital signal which is as close as possible to the signal that would have been obtained at another sampling rate $\hat{f}$.

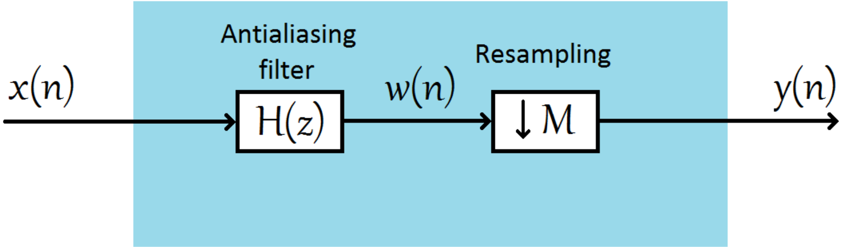
The two basic operations of multirate processing are

- *Decimation* (down-sampling) reduces the sampling rate of the signal by deleting part of the values of the original signal,
- *Interpolation* (up-sampling) increases the sampling rate of the signal by adding extra signal values.

Decimation and interpolation convert the sampling rate by some integer coefficient (for example, 1 kHz $\mapsto$ 2 kHz or 1 kHz $\mapsto$ 0.5 kHz). By combining these operations, resampling by any rational factor can be performed. If the two operations are combined, the interpolation must always be done before the decimation so that we do not lose unnecessarily information in the conversion.

8.1 Decimation

The diagram below shows the steps of decimating the signal $x(n)$. Before the actual sampling rate reduction, the signal must be filtered by a low-pass filter to prevent aliasing. The sampling rate of the filtered signal is reduced by taking only part of the signal values. The decimation operation is indicated by an arrow pointing downwards and the sampling rate reduction factor. For example, the symbol for reducing the sampling rate to one third is $\boxed{\downarrow 3}$. In the case shown in the diagram, the original sampling rate F_s will be reduced in the decimation to F_s/M. This is done by taking every Mth element into the decimated signal.



As the sampling rate decreases in the decimation, the risk of aliasing is obvious. The only way to prevent aliasing is to remove the frequencies greater than half of the new sampling frequency from the signal. The digital antialiasing filter thus is a low-pass filter that removes all the frequencies that are higher than $F_s/(2M)$. The low-pass filter design requirements given in normalized frequencies are as follows:

- The filter passband is $[0, \frac{1}{2M} - \Delta f]$.

- The filter stopband is $[\frac{1}{2M}, \frac{1}{2}]$.

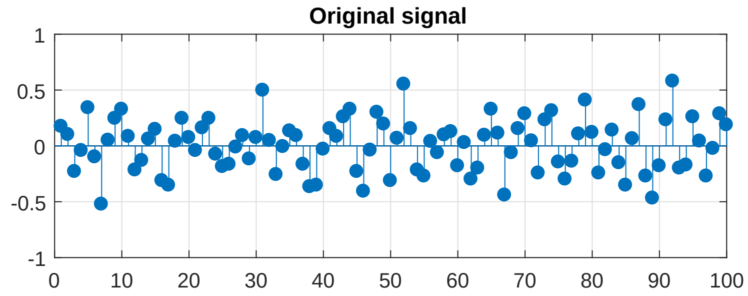
The transition bandwidth, Δf, is determined by the application. The narrower the transition band, the more coefficients the filter needs to have. Attenuation requirements also depend on the application.

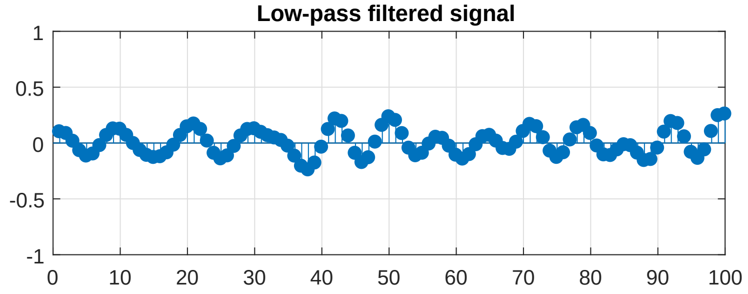
Mathematically the decimation process of the signal $x(n)$ by the factor M into the signal $y(n)$ can be expressed as follows:

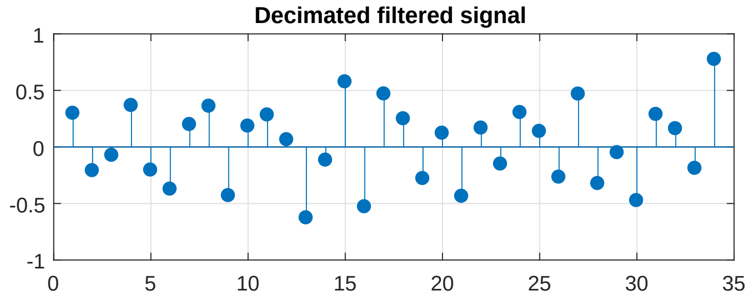
$$\begin{aligned} w(n) &= \sum_{k=-\infty}^{\infty} h(k)x(n-k), \\ y(n) &= w(nM) = \sum_{k=-\infty}^{\infty} h(k)x(nM-k). \end{aligned}$$

The figures below show the effect of decimation in the time domain. The original signal $x(n)$ is first filtered by a low-pass filter to obtain the signal $w(n)$ which can be safely decimated.

When decimating by factor 3, only every third value is taken to the new signal.

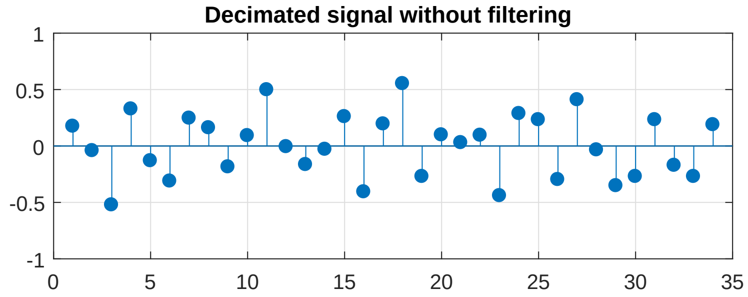




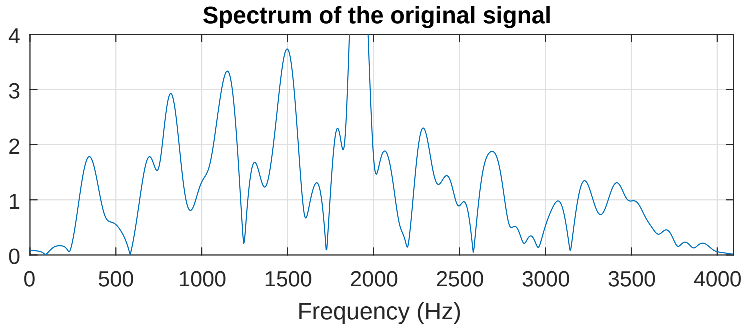


In the previous figure, the sample values are multiplied by the decimation factor $M = 3$ so that the magnitude of the samples is the same as in the original signal. Without this multiplication, sample values would be much lower than the original values, since in low-pass filtering the signal energy drops to about one third. Generally, this multiplication is omitted from the formulas because it is easiest to implement by designing the low-pass filter in such a way that its amplitude response is simply multiplied by M. Such a filter can be obtained from a normal low-pass filter by multiplying every filter coefficient by the factor M.

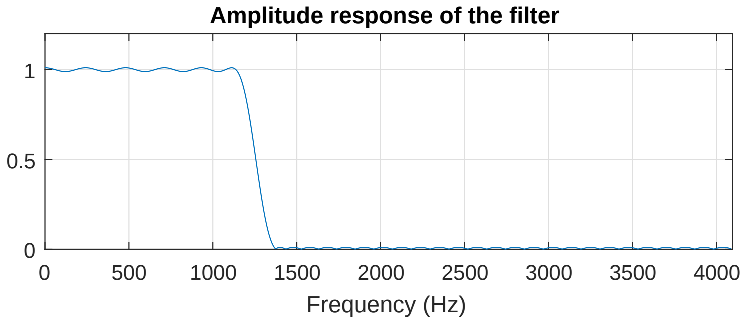
For comparison, this is the signal obtained by decimating without low-pass filtering.



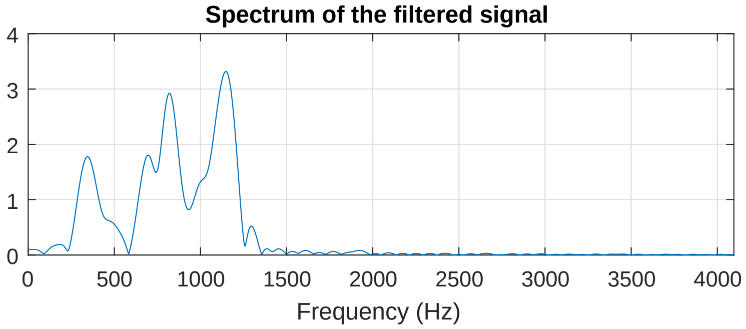
The decimation process is easier to grasp from the frequency domain representation. The first graph is the spectrum $|X(n)|$ (the absolute value of the DFT suitably windowed and interpolated) of the original signal $x(n)$. In this case the sampling rate is 8192 Hz and thus the maximum frequency in the signal is 4096 Hz.



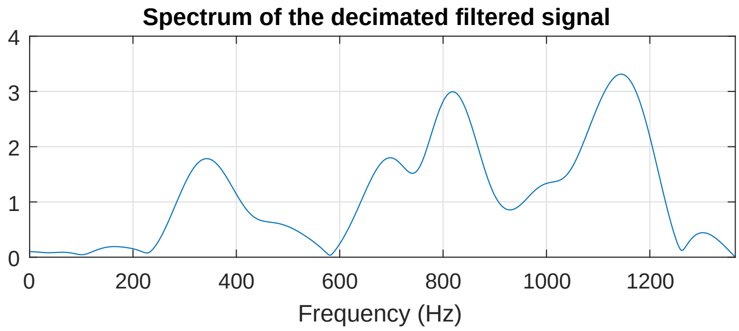
When we want to decimate this signal by factor 3, the frequencies above $1365\frac{1}{3}$ Hz must be removed first. For this purpose, we design a low-pass filter having stopband on the interval $[\frac{1}{6}, \frac{1}{2}]$ (frequencies normalized by the sampling frequency) and passband from zero to any number smaller than $1/6$ depending on the amount of coefficients we can afford to use. Here we selected the passband cut-off frequency to be 0.14, the passband ripple to be 0.09 dB and the minimum stopband attenuation to be 40 dB. The amplitude response of this filter is the following.



Filtering by this filter outputs the signal $w(n)$ whose spectrum is this.

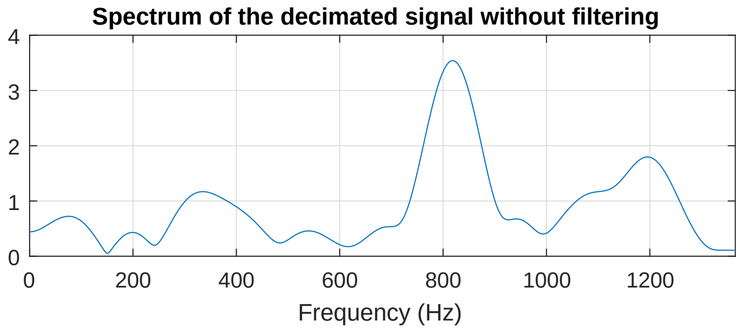


Taking every third value of $w(n)$ (and multiplying them by 3), the spectrum of the resulting signal is as follows:



Now, the new sampling rate is 8192/3 Hz.

Next graph shows the spectrum of the signal obtained by decimating without low-pass filtering where the aliasing is clearly detectable. The spectrum differs from the corresponding band of the original signal, which we would like the new spectrum to resemble as much as possible.



8.2 Multistage decimation

More efficient implementations of the decimators can be achieved by doing the decimation in several stages. For example, the decimation to one tenth of the original sampling rate can be done in one step by the factor 10 or in two steps: first by the factor 5 and then by the factor 2. The third possibility is to decimate first by the factor 2 and then by the factor 5. All of these methods require different number of filter coefficients, and in most cases, the single-stage decimation by the factor 10 requires more filter coefficients than the multistage implementations. Below are shown the different possibilities when the sampling frequency is originally 20 kHz and the target is 2 kHz. Additionally we assume that the frequencies up to 900 Hz of the signal must be preserved and that the filter type is an FIR filter with the Hamming window.

- Single-stage case: $x(n) \rightarrow \boxed{H(z)} \rightarrow \boxed{\downarrow 10} \rightarrow y(n)$
  Now the passband is $[0, 0.9]$ kHz and the stopband $[1, 10]$ kHz. The normalized transition band is therefore $[0.045, 0.05]$ and the required number of coefficients $N = 3.3/\Delta f = 3.3/0.005 \approx \mathbf{661}$.

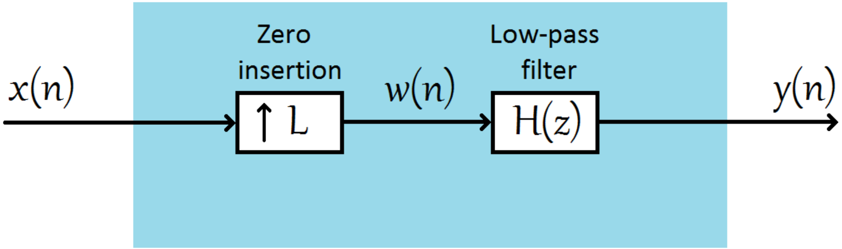
- 2-stage case 1: $x(n)\rightarrow\boxed{H_1(z)}\rightarrow\boxed{\downarrow 5}\rightarrow\boxed{H_2(z)}\rightarrow\boxed{\downarrow 2}\rightarrow y(n)$
  Now the passband for the filter $H_1(z)$ is $[0, 0.9]$ kHz and the stopband $[2, 10]$ kHz. The normalized transition band is therefore $[0.045, 0.1]$and the required number of coefficients $N_1 = 3.3/0.055 \approx 61$. The passband for the filter $H_2(z)$ is $[0, 0.9]$ kHzand the stopband $[1, 2]$ kHz. The normalized (by the sampling rate 4 kHz) transition band is $[0.225, 0.25]$ and the required number of coefficients $N_2 = 3.3/0.0250 \approx 133$. In total the required number of coefficients **194**.

- 2-stage case 2: $x(n)\rightarrow\boxed{H_1(z)}\rightarrow\boxed{\downarrow 2}\rightarrow\boxed{H_2(z)}\rightarrow\boxed{\downarrow 5}\rightarrow y(n)$
  $H_1(z)$: Now the passband for the filter $H_1(z)$ is $[0, 0.9]$ kHz and the stopband $[5, 10]$ kHz. The normalized transition band is therefore $[0.045, 0.25]$ and the required number of coefficients $N_1 = 3.3/0.205 \approx 17$. The passband for the filter $H_2(z)$ is $[0, 0.9]$ kHz and the stopband $[1, 5]$ kHz. The normalized (by the sampling rate 10 kHz) transition band is $[0.09, 0.1]$ and the required number of coefficients $N_2 = 3.3/0.01 \approx 331$. In total the required number of coefficients **348**.

Thus, it is wisest to perform the task in two steps with coefficients 5 and 2 (in this order). Generally it is true that the decimation factors should be placed in the descending order. Therefore, the order $x(n)\rightarrow\boxed{H_1(z)}\rightarrow\boxed{\downarrow 2}\rightarrow\boxed{H_2(z)}\rightarrow\boxed{\downarrow 5}\rightarrow y(n)$always produces more coefficients than the order $x(n)\rightarrow\boxed{H_1(z)}\rightarrow\boxed{\downarrow 5}\rightarrow\boxed{H_2(z)}\rightarrow\boxed{\downarrow 2}\rightarrow y(n)$, and it could have been ignored in the first place.

8.3 Interpolation

The purpose of the signal interpolation is to obtain a signal that is equivalent to the original, but has a higher sampling rate. This diagram shows the steps of interpolating the signal $x(n)$.



Unlike in the decimation, no pre-filtering is required now, because when the sampling rate is increased all the original frequencies can be produced. The first operation is to increase the sampling rate to L times the original. This operation is indicated by an arrow pointing upwards and the sampling rate interpolation factor L, i.e. $\boxed{\uparrow L}$. There are numerous options for implementing this operation, i.e. determining the new extra values? In this case, a mathematician would probably fit the polynomials or splines to obtain new values between the existing ones. In signal processing, however, the situation is treated differently: zeros are placed for the unknown values and the resulting signal is filtered by a low-pass filter. When we add zeros, we add very high frequencies to the signal.

These high frequencies should be removed and naturally that is done by low-pass filtering. The highest frequency in the original signal is $F_s/2$. When interpolated to the sampling rate LF_s, the highest available frequency is $LF_s/2$. The frequencies above $F_s/2$ must be removed because they are resulting from the addition of zeros.

The signal obtained by adding the zeros (sampling rate LF_s) is filtered by a low-pass filter that preserves the frequencies on the interval $[0, F_s/2]$ and removes the higher frequencies. Since the new sampling rate is LF_s, the filter requirements need to be normalized by this number. So we need to design a low-pass filter satisfying these requirements:

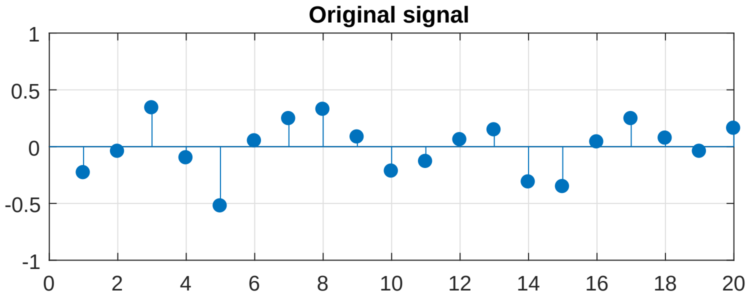
- The passband in normalized frequencies is the interval $[0, (F_s/2)/(LF_s) - \Delta f]$ $= \left[0, \frac{1}{2L} - \Delta f\right]$.
- The width of the transition band Δf depends again on the application where the interpolation is to be used (Δf determines the number of coefficients).
- The stopband is the interval $[(F_s/2)/(LF_s), 1/2] = \left[\frac{1}{2L}, \frac{1}{2}\right]$.

The attenuation requirements are also largely determined by the application and no general guidelines can be given for choosing them. Since $L-1$ zeros are placed for each signal value in the original signal, the amplitude of the signal decreases in the filtering to one Lth of the original. Therefore, after (or before) the filtering the signal values should be multiplied by L.

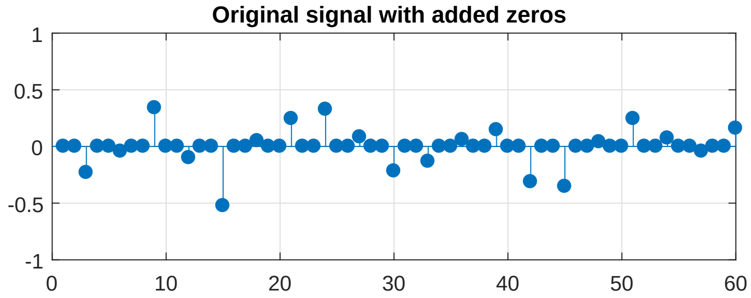
Mathematically the interpolation process of the signal $x(n)$ by the factor L into the signal $y(n)$ can be expressed as follows:

$$\begin{aligned} w(n) &= \begin{cases} x(n/L), & \text{when } n = 0, \pm L, \pm 2L, \ldots, \\ 0, & \text{otherwise}, \end{cases} \\ y(n) &= \sum_{k=-\infty}^{\infty} h(k) w(n-k). \end{aligned}$$

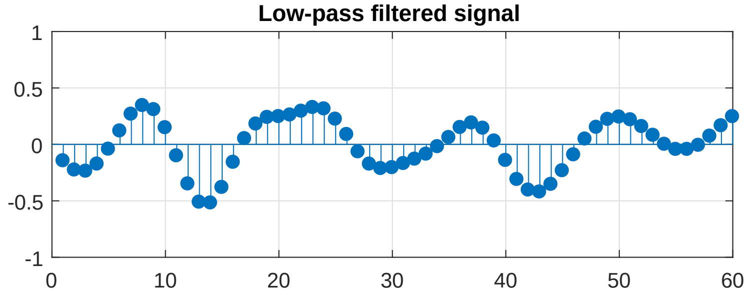
Let's have an example where the signal is interpolated by factor 3 and thus the sampling frequency is tripled ($L = 3$).



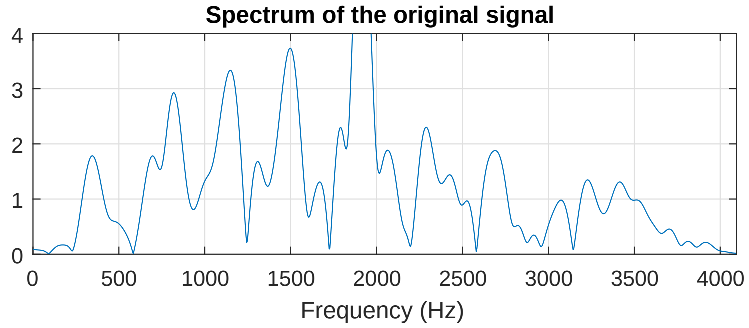
First $L - 1 = 2$ zeros are placed between each two consecutive samples in $x(n)$, which gives the signal $w(n)$.



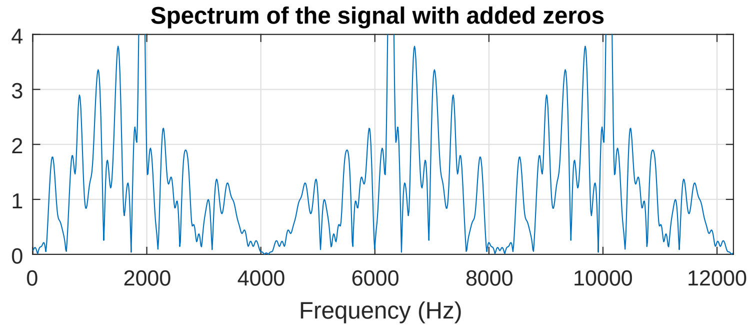
This is filtered by a low-pass filter with the stopband $[\frac{1}{6}, 0.5]$, the passband $[0, 0.1526]$, the passband ripple 0.09 dB and the minimum stopband attenuation 40 dB. The result is the output signal $y(n)$. As in the case of decimation, this signal must finally be multiplied by $L = 3$ to preserve the energy of the signal. Alternatively, the coefficients of the low-pass filter may be multiplied by L.



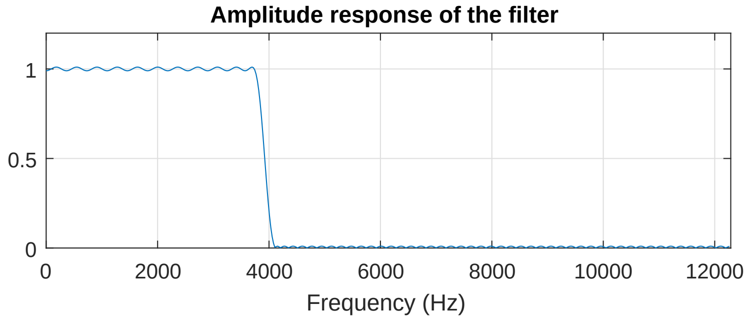
In the frequency domain, the original signal with the sampling rate 8192 Hz is this.



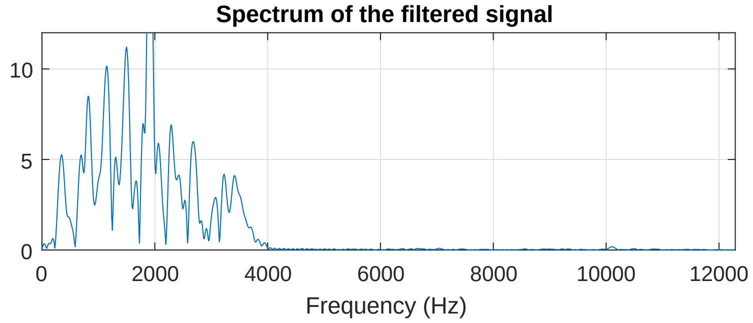
When zeros are added to the signal, we obtain a signal with the following spectrum $|W(e^{i\omega})|$.



The frequencies between 4096 Hz – 12288 Hz must be removed e.g. by a filter with this amplitude response.

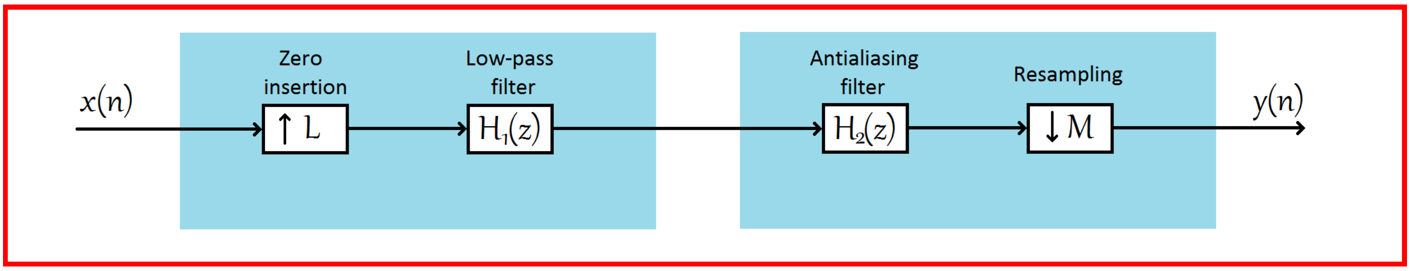


The result is the signal $y(n)$ whose spectrum is this.

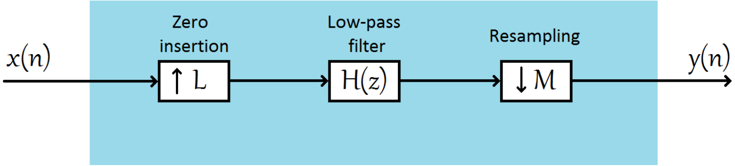


8.4 Resampling

As mentioned earlier, the sampling rate conversion by a rational factor is obtained by combining interpolation and decimation. If the sampling rate is wanted to be $\frac{L}{M}$ times the original, then the signal is first interpolated by the factor L and then decimated by the factor M. Here, in order to save computing resources, you should simplify the fraction $\frac{L}{M}$ as much as possible. The block diagram of the system is as shown below.



The diagram shows that the system has two low-pass filters in series. One of these can be removed if the remaining one has stricter requirements. For example, if $\frac{L}{M} = \frac{4}{3}$ the passband of the filter $H_1(z)$ is $[0, \frac{1}{2L} - \Delta f] = [0, \frac{1}{8} - \Delta f]$ and the stopband $[\frac{1}{8}, \frac{1}{2}]$. The corresponding values of the filter $H_2(z)$ are $[0, \frac{1}{2M} - \Delta f] = [0, \frac{1}{6} - \Delta f]$ and $[\frac{1}{6}, \frac{1}{2}]$. The filter $H_1(z)$ therefore removes a wider bandwidth than $H_2(z)$, so only the filter $H_1(z)$ alone is enough. Of course we must also ensure that all the filtering requirements are fulfilled The diagram below shows the obtained simplified system.



8.5 Applications of Multirate DSP

Some applications of multirate signal processing are listed below:

1. Various systems in digital audio signal processing often operate at different sampling rates. Thus changes from one system to another require conversion of the sampling rate.

2. Signal has been sampled at a too high frequency and it does not contain frequencies up to the Nyquist frequency. This makes designed filters to have unnecessarily high order and down-sampling of the signal decreases the order of the filters.

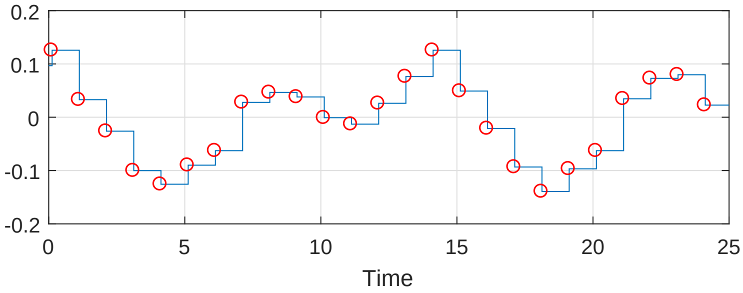
3. Decomposition of a signal into components that contain different frequency bands. Every component can then be represented using much smaller sampling rate.

4. In the implementation of high-performance filtering operations, a very narrow transition band is required. This leads to very high filter order which can be avoided by decomposing the signal into a number of subbands where each component can be processed at a lower rate, and the transition band will be less narrow.

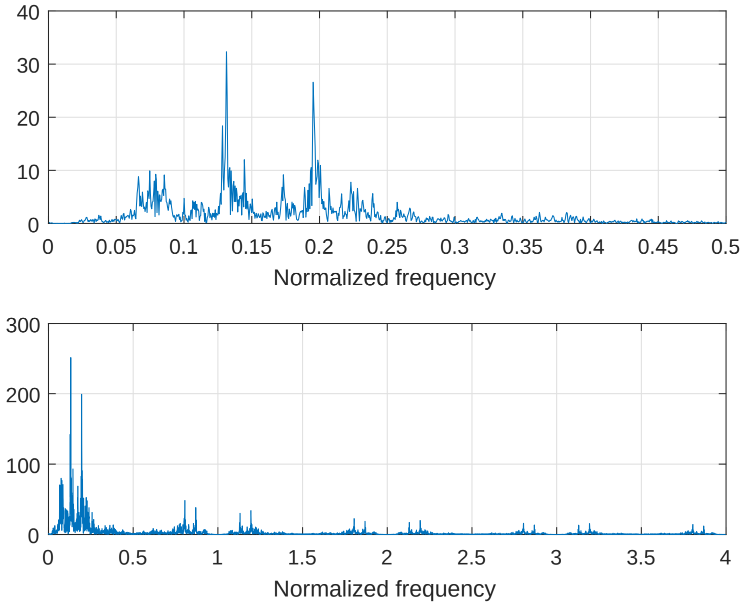
5. Interpolation before D/A conversion in order to relax the requirements of the analog lowpass antialiasing filter. Next we consider D/A conversion in the CD player with an implementation method called *zero-order hold*.

8.5.1 Zero-Order Hold

In its simplest form, the conversion from a digital signal to an analog is done using a *zero-order hold* (ZOH, sample-and-hold, S/H). In this case, the latest value of the digital signal is set as the value of the analog signal. The following figure shows the D/A conversion using a zero-order hold circuit. The circles represent the values of the digital signal and the line is the analog approximation.



In the case of an audio signal, the most interesting thing is to see whether the spectrum of the analog signal corresponds to the spectrum of the original digital signal. The answer is in the following simulated spectra where the upper figure shows the spectrum of the digital signal. In the horizontal axis are the frequencies normalized by the sampling frequency. The lower figure represents the spectrum of the analog signal after the zero-order hold circuit. Now the energy of the signal is spread into a much wider area so that the original spectrum (interval $[0, 0.5]$) duplicates to larger frequencies. On the interval $[0.5, 1]$ is an attenuated mirror image of the original spectrum, on the interval $[1, 1.5]$ an attenuated copy of the original spectrum, on the interval $[1.5, 2]$ attenuated mirror image, etc. The figure shows only the original spectrum and seven copies of it, but in fact there are infinitely many copies (with smaller and smaller energies).



The behavior of the zero-order hold circuit in the frequency domain can be presented analytically. Since this is a continuous-time filter, the analytical method differs slightly from the ones we have studied. The holding circuit can be thought of as a filter whose impulse response is a continuous function

$$h(t) = \begin{cases} 1, & \text{when } 0 \leq t < T, \\ 0, & \text{otherwise,} \end{cases}$$

where T is the difference between the two sampling times (e.g. for CD player 1/44100 s). In the case of an analog filter, the convolution is defined by the formula

$$y(t) = \int_{-\infty}^{\infty} h(u)x(t-u)du,$$

which in this case is

$$y(t) = \int_{0}^{T} x(t-u)du.$$

The frequency response of such a filter is

$$\begin{aligned} H(e^{i\omega}) &= \int_{-\infty}^{\infty} h(t)e^{-i\omega t}\,dt \\ &= \int_{0}^{T} e^{-i\omega t}\,dt \\ &= \Big/_{\!0}^{\!T} \frac{1}{-i\omega}e^{-i\omega t} \\ &= \frac{e^{-i\omega T}-1}{-i\omega}. \end{aligned}$$

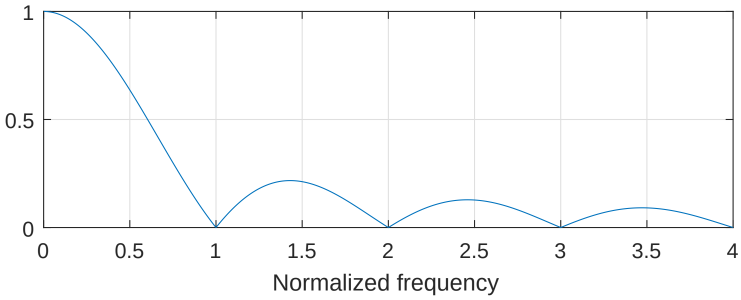
The last expression can be simplified into the form

$$H(e^{i\omega}) = Te^{-i\omega T/2}\,\mathrm{sinc}(\omega T/2),$$

whereby the amplitude response becomes

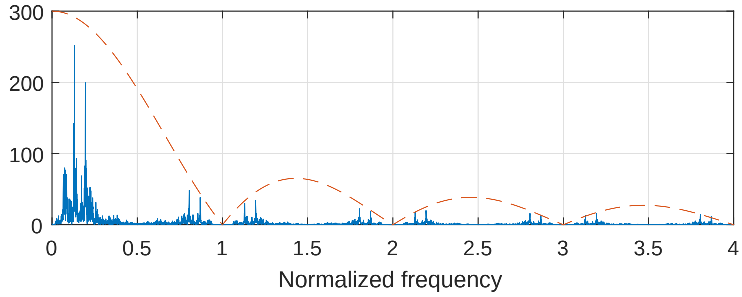
$$|H(e^{i\omega})| = T|\,\mathrm{sinc}(\omega T/2)|.$$

The graph of this function is below (now $T = 1$).



The sampling frequency of the digital signal is on this scale at 1 and the Nyquist's frequency is at 0.5. Each peak of the graph produces one attenuated copy of the spectrum of the digital signal.

The graph of the amplitude response fitted to the signal spectrum is below. For the visualization purposes the amplitude response has been multiplied by three hundred.



Thus we found out that the holding circuit produces additional frequencies for the analog signal. In the case of a CD player, extra energy is at frequencies above 22.05 kilohertz, which the human ear cannot hear. Extra energy at high frequencies may, however, strain the amplifier, so it is best to remove it.

High frequencies could be removed by an analog low-pass filter, for example, having the interval $[0, 20]$ kHz as the passband and $[22.05, \infty)$ kHz as the stopband (there is no upper limit on frequencies being processed by analog filters). In that case, the transition band becomes relatively narrow, making the analog filter difficult to design and expensive.

Using the multirate methods, the filtering can be done in two stages, making the filters simpler. In the first stage, the sampling rate of the digital signal is quadrupled (176.4 kHz). This is done by adding three zeros between each two consecutive samples and filtering the result with a low-pass filter having the passband on the interval $[0, 20]$ kHz and the stopband on the interval $[22.05, 88.2]$ kHz. The implementation of this filter depends on the desired accuracy. In the passband, the amplitude response of the filter is rising so that near the zero frequency it is slightly below zero dB and near 20 kHz it is slightly over zero dB. This is to compensate for the small attenuation of higher frequencies in the zero-order hold circuit.

In the second stage, the digital signal is converted to an analog signal with a zero-order hold circuit and finally filtered by an analog filter that eliminates the extra energy appearing in the high frequencies. Now the requirements for the analog filter are easy to implement: the passband $[0, 20]$ kHz and stopband $[88.2, \infty)$ kHz. The width of the transition band is now more than 30 times the previous one. In fact, the transition band could be stretched to more than 150 kHz without any problems (why?).

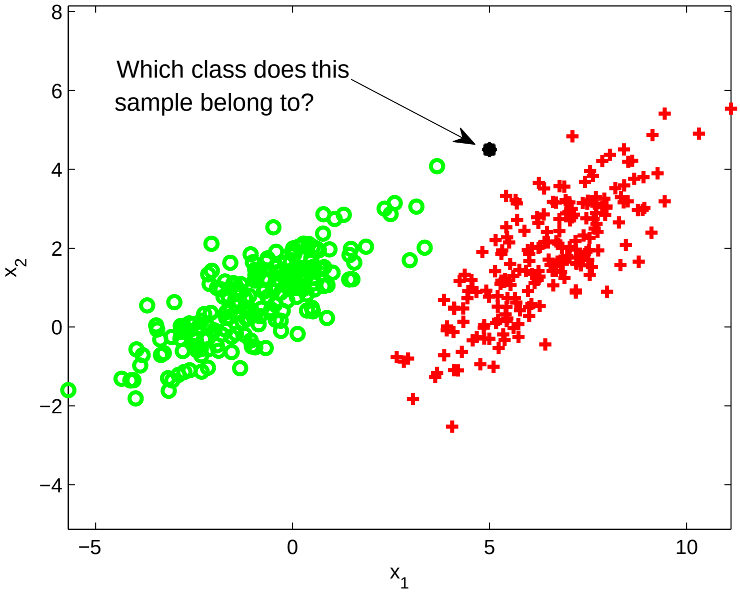
Chapter 9

Pattern Recognition

In this chapter we explore *learning systems*, for which, a computational problem is taught via examples. One significant part of the systems in this category are the *classifiers* which, by using examples, have to learn how to categorize multi-dimensional data ($\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N \in \mathbb{R}^n$) to a finite number of classes: $y_1, y_2, \ldots, y_N \in \{1, 2, \ldots, M\}$.

Applications of learning systems for signal processing are for example text recognition and speech recognition. In addition to these, the methods are used for example in the search engines, medical diagnostics and email filters. For all these systems the common factor is that they learn the classification by means of examples: for example, in text recognition hundreds or thousands of examples of each character with the correct classification are given to the classifier. Based on these examples the learning algorithm selects the appropriate parameters for the classifier. For example, the MNIST database commonly used to identify handwritten digits consists of a total of about 70000 bitmaps that represent handwritten digits.

We will consider next three commonly used classification methods: *nearest neighbor classifier*, *linear discriminant analysis* and *support vector machine.* We use an example where by means of 2-dimensional sets (red plus signs and green circles in the following figure) an inference rule should be learned in order to classify new samples (such as the black ball in the figure).



On the face of it, the problem may seem easy because in this 2-dimensional case anyone can draw a fairly good line between the classes. However, the problem becomes more challenging when the dimension of the classes increases and they partly overlap. For example, the samples of the above-mentioned MNIST database are images of size 20 × 20 pixels, so the space is 400-dimensional.

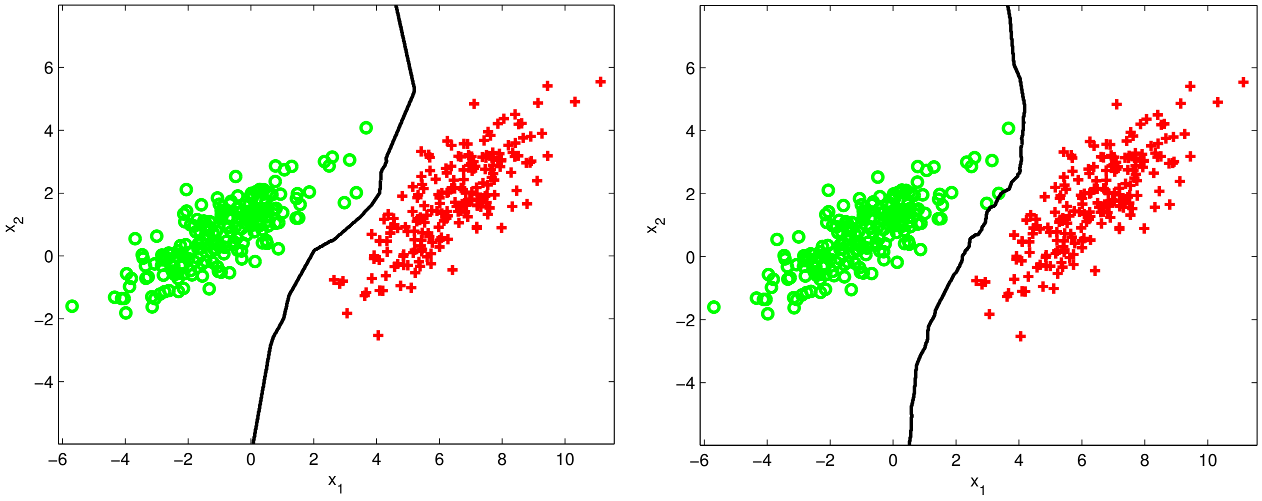
Important concepts for classification, and in machine learning in general, are *training set* and *test set*. The training set is used in the training of the system, but the performance of the trained system is evaluated by a separate test set. In reality, the classifier classifies mostly samples that do not belong to the training set, so the classification error in the training set is not a reliable performance indicator. For example, the aforementioned 70000 bitmaps of digits in the MNIST database are divided into 60000 training set images and 10000 test set images.

9.1 Nearest Neighbor Classifier

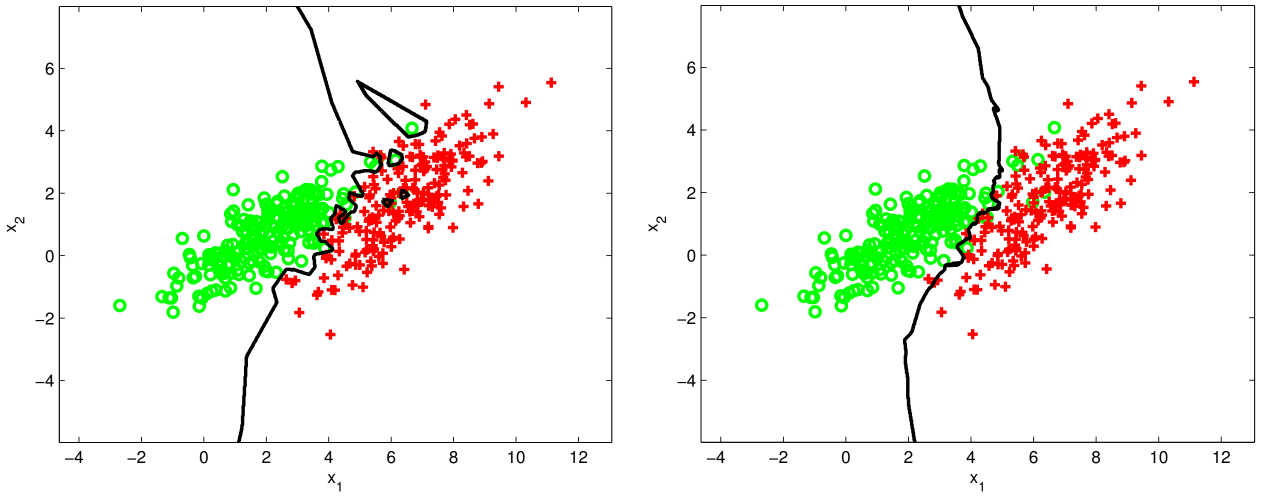
A simple and easy to implement solution to the classification problem is the nearest neighbor (NN) classifier. The NN classifier looks for the most similar sample of the training set and gives the same class for the sample to be classified. A more general version of this rather intuitive classifier is the k-NN classifier which searches for k most similar samples from the training set and selects the most common class among them.

Thus, the method does not require a separate training phase, but it keeps in mind the whole training data as such. This is also the biggest problem of the method: when the class of a new sample is to be decided, the whole data must be gone through in order to find the nearest data samples. In addition, the entire data must be in memory, which can be a problem when the data set is large. For small training data sets k-NN works just as well as any other classifier, and is often used in simple classification problems where the classes are clearly distinguishable.

Let's see how the k-NN classifier works for our two-class example data. The left figure shows the result of the 1-NN and the right 9-NN classifier for our two-class example data. It is seen from the figure that with higher values of the parameter k the method becomes less sensitive to individual outlying samples. In the right-hand figure, the right-most green circles affect the class limit less as they are quite separate from the other samples.



If the classes are overlapping more, the 1-NN classifier's class boundary is pretty fragmented. Below is an example of such a situation. In the 1-NN classifier result (left), each sample point affects the outcome, and the class boundary consists of many separate parts. In the 9-NN classifier result (right), the class boundary is continuous and probably works better.



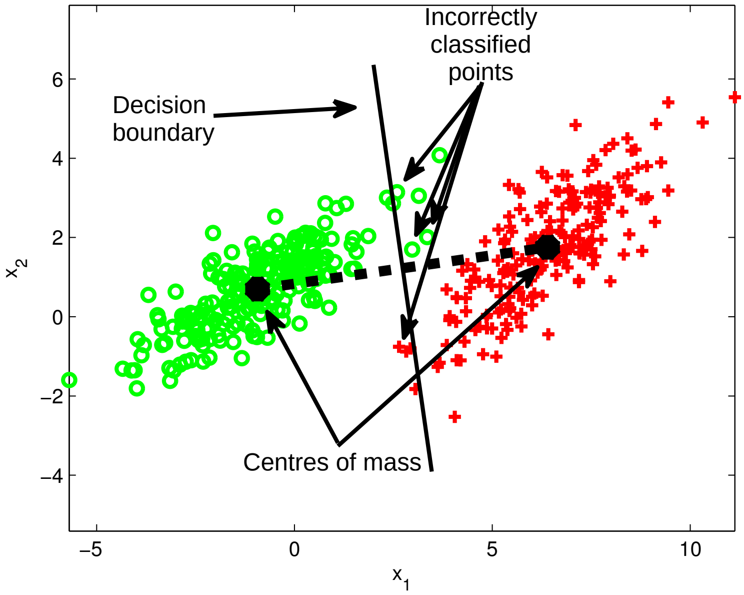
Next, we will get to know more complex classifiers that have more compact representation than the k-NN classifier. Compact representation can be, for example, a function $f(\mathbf{x})$ whose value (e.g. positive or negative) determines the class of the sample $\mathbf{x}$. In this case, the class selection is faster, but the inference of good function f takes more time. This is usually desirable, because the training phase is not real-time and the duration of the training is not as critical as the duration of the classification itself.

9.2 Linear Discriminant Analysis

Linear discriminant analysis (LDA, Fisher's linear discriminant) was published 1936, but is still widely used, mainly due to its simplicity. The basic idea in 2-dimensional case is to find such a line (or a hyperplane in multi-dimensional case) that when we project the data on the line the inter-class variance is maximized (i.e. the classes are as far apart as possible).

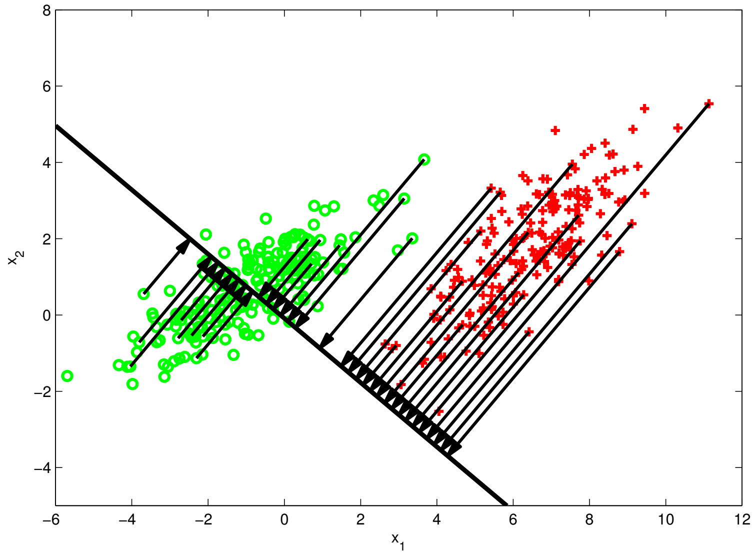
The first trial might be to place the decision boundary roughly to the midpoint of the classes. For 2-dimensional data, this can be determined by hand, but for example, for 100-dimensional data an automatic method is required to determine the decision boundary.

The simplest solution is to draw a line segment between the centres of mass of the classes and draw (an orthogonal) decision boundary midpoint between them. This solution for our example is shown in the next figure.



From the result we see that the solution is not the best possible if the classes overlap at all. Wrong decisions remain however we choose the location of the boundary.

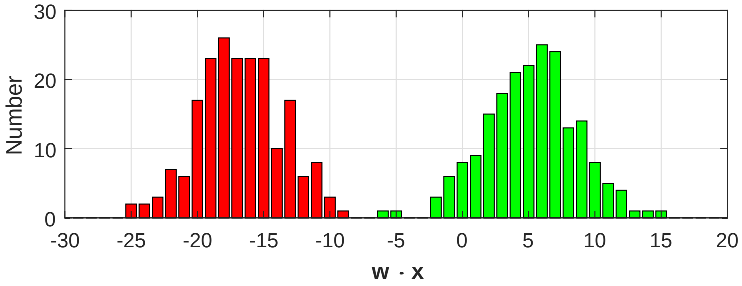
A better solution would be such a line, that when we project the point sets on the line they become separate 1-dimensional sets. For example, in the next figure, the two point clouds are projected on one such line (the projection of a point maps it to the closest point on the line).



The figure shows that for the classification only the direction of the line given by vector $\mathbf{w}$ matters. The projection of the sample $\mathbf{x}$ on a line in the direction $\mathbf{w}$ is defined by the formula

$$\text{proj}_{\mathbf{w}}(\mathbf{x}) = \frac{\mathbf{w}\cdot\mathbf{x}}{|\mathbf{w}|^2}\mathbf{w}.$$

The location of the projected point on the line is determined by the dot product $\mathbf{w}\cdot\mathbf{x}$, so for simplicity it is sufficient to look at it alone. The distribution of $\mathbf{w}\cdot\mathbf{x}$ is shown in the figure below.



It would seem that this line is a good choice and for example, the following simple algorithm would be a good classifier for the data in question:

$$\text{The class of the point } \mathbf{x} = \begin{cases} \text{green circle}, & \text{if } \mathbf{w}\cdot\mathbf{x} \geq -7.5, \\ \text{red plus sign}, & \text{if } \mathbf{w}\cdot\mathbf{x} < -7.5. \end{cases}$$

How is the vector $\mathbf{w}$ then selected? In this simple case, it can be done manually, but in the case of 100-dimensional partly overlapping classes, an automatic method is required. Fisher formulated the question as an optimization problem: what is the vector $\mathbf{w}$, for which the inter-class variance is maximized while the intra-class variance is minimized. By noting that $\mathbf{w}\cdot\mathbf{x} = \mathbf{w}^T\mathbf{x}$, this Fisher's class separation measure is defined as

$$J(\mathbf{w}) = \frac{\text{intra-class variance}}{\text{inter-class variance}} = \frac{\mathbf{w}^T(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T\mathbf{w}}{\mathbf{w}^T(\mathbf{C}_1 + \mathbf{C}_2)\mathbf{w}},$$

where $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are the means of the classes and $\mathbf{C}_1$ and $\mathbf{C}_2$ their sample covariances. This can be maximized by differentiating w.r.t. $\mathbf{w}$, setting the result equal to zero and solving for $\mathbf{w}$ which is found to be

$$\mathbf{w} = (\mathbf{C}_1 + \mathbf{C}_2)^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2).$$

For our example, the covariance matrices and means have the following values

$$\mathbf{C}_1 = \begin{pmatrix} 2.4775 & 1.3271 \\ 1.3271 & 1.1239 \end{pmatrix}, \mathbf{C}_2 = \begin{pmatrix} 2.2666 & 1.7515 \\ 1.7515 & 2.1657 \end{pmatrix}, \boldsymbol{\mu}_1 = \begin{pmatrix} -0.9330 \\ 0.6992 \end{pmatrix}, \boldsymbol{\mu}_2 = \begin{pmatrix} 6.3992 \\ 1.7508 \end{pmatrix}.$$

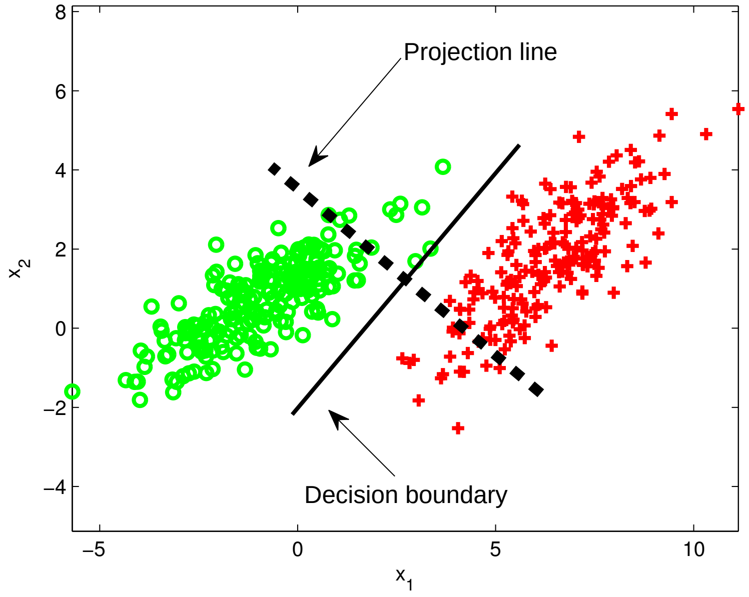
Thus $\mathbf{w} = [-3.4077, 2.8695]^T$, which is the direction vector of the projection line. Note that all the vectors with the same or opposite direction are equally valid. Often, it is customary to normalize the vector to unit vector: $\mathbf{w}' = \mathbf{w}/||\mathbf{w}||$.

This classifier is extremely simple to implement as a program. The class of the new sample $\mathbf{x} = (x(1), x(2))$ is found by calculating the dot product

$$\mathbf{x} \cdot \mathbf{w} = -3.4077x(1) + 2.8695x(2)$$

and comparing the result with the threshold value c (for example, the value $c = -7.5$ above). In a realistic situation, choosing a threshold by "looking at the figure" is too inaccurate, so an automatic method also for this is needed. The best way is to choose c so that the percentage of incorrectly classified training samples is as small as possible (in our example, all thresholds between $c \in [-8.9267, -5.6639]$ will produce zero error so they would all be eligible). A simpler way is to select the threshold to be the midpoint between the means of the classes (see Exercise 12, Tasks 2 and 4), in which case we obtain the value $c = -5.7984$.

The decision boundary obtained with this threshold is shown in the next figure which shows that the method divides the training set into two classes correctly.

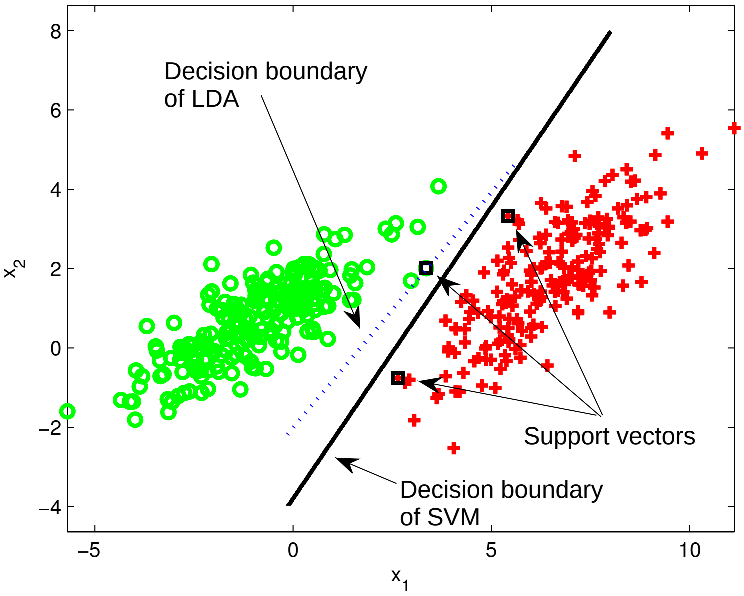


There are, however, two green points close to the decision boundary, which are classified barely correctly. A safer solution would seem to be to choose a decision boundary where the margin on both sides of it would be as large as possible. Support vector machines are based on this idea and we will discuss them next.

9.3 Support Vector Machine

The support vector machine (SVM) was originally developed in the 1960s in Moscow, but a wider popularity emerged only in the late 1990s with the help of efficient computing methods and more sophisticated theory. The idea of the method is to choose the decision boundary that maximizes the margin between the classes (i.e. the distance from the boundary to the nearest data point on each side is maximized). For example, the above LDA decision boundary is drawn to the midpoint of the means of the classes, but it happens to be closer to some green circles than red plus signs. SVM draws the decision boundary so that the distance to the closest samples for both classes is as large as possible.

The solution of the SVM support vectors and the decision boundary is quite mathematical and is a topic of later courses. However, SVM has several implementations that can be used even if the operation of the algorithm is not completely clear. Popular implementations include LibSVM and Matlab Bioinformatics toolbox implementation. The optimum solution is illustrated in the next figure. The blue dashed line in the figure is the decision boundary of the LDA and the black solid line the decision boundary of the SVM, which is supported by the so-called *support vectors*. The distance (i.e. margin) to all three support vectors is now the same and the support vectors are marked with black squares.



Thus, the SVM maximizes the margin around the decision boundary and selects the support vectors accordingly. This feature does not fully do itself justice in the case of a linear decision boundary since the decision boundary of the LDA is often quite close to the SVM boundary. Maximizing the margins, however, is emphasized when the data is of higher dimension and the margins are typically larger.

The popularity of the SVM is based on this finding and the use of the so-called *kernel trick*, which was introduced in 1964 to be used with the LDA. The kernel trick was not very popular with the LDA because it did not work particularly well with it. The idea of the kernel trick is to map the data into a higher-dimensional space using a kernel. The kernel can be any function that satisfies certain conditions and the purpose is to classify the data in a higher dimension using a linear decision boundary (hyperplane). The solution is called a trick because the data does not need to be explicitly transferred to a higher dimension in order to implement it, but it is enough to just replace all the dot products in the algorithm by a suitable function. Thus, the kernel trick is applicable to all methods based on the dot product of vectors (such as LDA and SVM).

There are numerous different kernels. The simplest ones are the polynomial kernels that are formed by raising the dot product to an integer power. For example, the second degree polynomial kernel corresponds to the substitution of the normal dot product with its second power:

$$k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2 .$$

In our example case, the data is 2-dimensional, so the dot product of the vectors $\mathbf{x}_i = (\mathbf{x}_i(1), \mathbf{x}_i(2))$ ja $\mathbf{x}_j = (\mathbf{x}_j(1), \mathbf{x}_j(2))$ is defined by the formula

$$\mathbf{x}_i \cdot \mathbf{x}_j = \mathbf{x}_i(1)\mathbf{x}_j(1) + \mathbf{x}_i(2)\mathbf{x}_j(2).$$

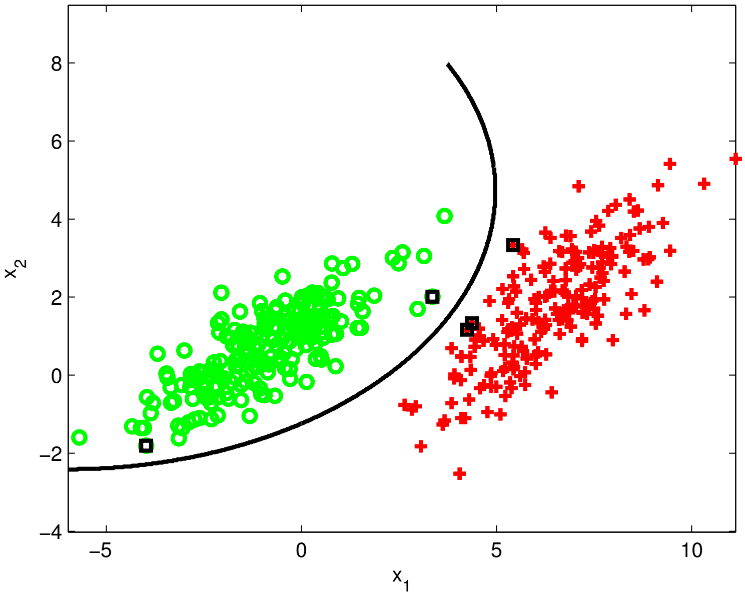
The second power of the dot product is thus the expression

$$\begin{aligned}
k(\mathbf{x}_i, \mathbf{x}_j) &= \big[\mathbf{x}_i(1)\mathbf{x}_j(1) + \mathbf{x}_i(2)\mathbf{x}_j(2)\big]^2 \\
&= \big[\mathbf{x}_i(1)\mathbf{x}_j(1) + \mathbf{x}_i(2)\mathbf{x}_j(2)\big] \cdot \big[\mathbf{x}_i(1)\mathbf{x}_j(1) + \mathbf{x}_i(2)\mathbf{x}_j(2)\big] \\
&= \mathbf{x}_i^2(1)\mathbf{x}_j^2(1) + \mathbf{x}_i^2(2)\mathbf{x}_j^2(2) + 2\mathbf{x}_i(1)\mathbf{x}_j(1)\mathbf{x}_i(2)\mathbf{x}_j(2) \\
&= \mathbf{x}_i^2(1)\mathbf{x}_j^2(1) + \mathbf{x}_i^2(2)\mathbf{x}_j^2(2) + (\sqrt{2}\mathbf{x}_i(1)\mathbf{x}_i(2)) \cdot (\sqrt{2}\mathbf{x}_j(1)\mathbf{x}_j(2)) \\
&= (\mathbf{x}_i^2(1), \mathbf{x}_i^2(2), \sqrt{2}\mathbf{x}_i(1)\mathbf{x}_i(2))^T \cdot (\mathbf{x}_j^2(1), \mathbf{x}_j^2(2), \sqrt{2}\mathbf{x}_j(1)\mathbf{x}_j(2))^T.
\end{aligned}$$

The idea behind the kernel trick is to figure out that this expression is also a dot product in the 3-dimensional space. As can be seen from the last line, the same result can be obtained by mapping the original 2-dimensional vector $\mathbf{x} = (\mathbf{x}(1), \mathbf{x}(2))$ into 3-dimensional space by the mapping

$$\begin{pmatrix} \mathbf{x}(1) \\ \mathbf{x}(2) \end{pmatrix} \mapsto \begin{pmatrix} \mathbf{x}(1)^2 \\ \mathbf{x}(2)^2 \\ \sqrt{2}\mathbf{x}(1)\mathbf{x}(2) \end{pmatrix} . \tag{9.1}$$

Consequently, it can be concluded that raising the dot products to the second power corresponds to normal SVM in a 3-dimensional case where the samples have been obtained by the formula (9.1). Next figure shows the result of the classification.



In the previous case, the same classifier could have been implemented without the kernel trick, simply by designing a 3-dimensional SVM classifier for the data mapped by (9.1). However, an explicit mapping to a higher dimension is not possible if the dimension is very high. Such a situation arises, for example, when an inhomogeneous polynomial kernel is used

$$k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j + 1)^d \,,$$

which includes all monomials up to degree d. The mapping (9.1) would change in this case to the form

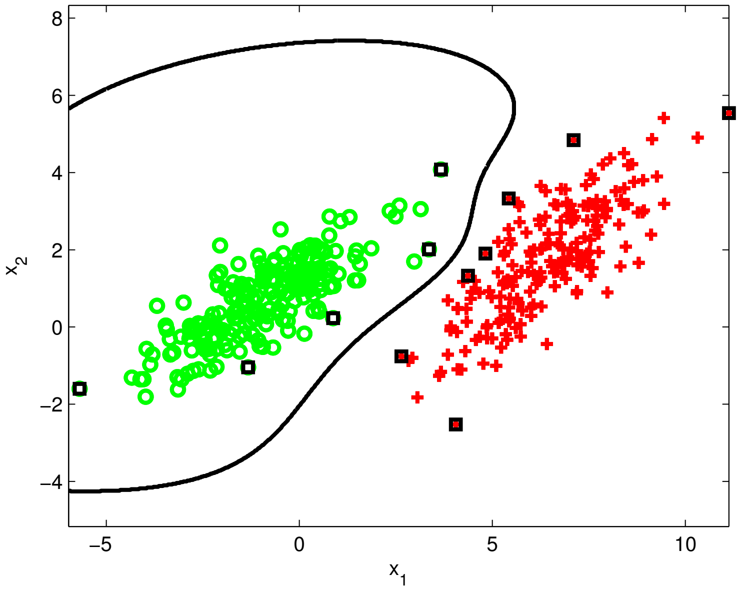
$$\begin{pmatrix} \mathbf{x}(1) \\ \mathbf{x}(2) \end{pmatrix} \mapsto \begin{pmatrix} \mathbf{x}(1)^2 \\ \mathbf{x}(2)^2 \\ \sqrt{2}\mathbf{x}(1)\mathbf{x}(2) \\ \sqrt{2}\mathbf{x}(1) \\ \sqrt{2}\mathbf{x}(2) \\ 1 \end{pmatrix} . \tag{9.2}$$

If, for example, the original data would be 5-dimensional and $d = 8$, direct mapping would have dozens of terms and the kernel trick starts to seem more attractive.

As an extreme example of the possibilities of the kernel trick is the (Gaussian) radial basis function (RBF) corresponding to the mapping into an *infinite-dimensional* space. The RBF kernel replaces the dot product of the vectors $\mathbf{x}_i \cdot \mathbf{x}_j$ by the function

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2),$$

where the parameter $\gamma > 0$. It can be shown that this trick corresponds to the search for a linear decision boundary hyperplane in an infinite-dimensional space. Next figure shows an example of the decision boundary when using the RBF kernel in the case $\gamma = 1$.



Although we have assumed that the classes are separable by a linear decision boundary (in order for the margin to exist at all), this is often not the case. For this reason, the SVM algorithm has also been generalized to situations where the classes overlap, and all reasonable implementations support this.

SVM has gained considerable popularity, primarily because of its ability to maximize the classification margin. This makes teaching easier and makes the classifier work well even for small training data. Other classifiers have a risk of so-called *overlearning* or *overfitting*. In general, SVM is considered not to be as sensitive to the size of the training data as other classifiers.