UNIVERSITY OF LJUBLJANA

FS

Faculty of Mechanical Engineering

Vid Tilen Ratajec

Digital filters

A short introduction

Ljubljana, 2024

1 Introduction

In this short introduction to digital filtration, we will be describing digital filtration of deterministic signals. In signal processing, we try to transfer a signal from the *source* (e.g. sensors) to the *sink* (e.g. data logging device), that signal is transmitted via a *channel* (e.g. wire, RF). Unwanted *noise* is usually added to information. Filtration of the information is used to remove or minimise the effect of the noise.
Example of such a simple system is shown on Figure 1.

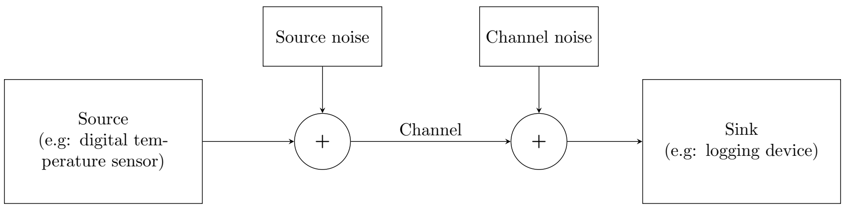


Figure 1: Simplified signal transfer chain

In this article, we will limit ourselves to discrete signals or data and simple linear filters. Most of the data is one dimensional and expressed with numbers. At first we will define some used terms, explain what filtration is, and why we need it in signal processing. Later we will discuss simple mathematical theory of filters and their types and show two examples of filtration, on a sound signal and an image. The reader should be somewhat familiar with signal processing.

1.1 Definitions

Deterministic signal	A signal in which each value is fixed.
Discrete signal	Value of the signal is known for only certain moments in time. Values in between of such moments are not defined.
Continuous signal	Value of the signal is known for any moment in time.
Noise	General term for unwanted modifications a signal may suffer during capture, transmission, storage, etc. Noise can be additive or it can occupy a completely different frequency.
Source	Source of the signal/information. Example: temperature sensor, light sensor
Sink	Destination of the signal/information. Example: micro-controller, thermostat
Channel	Something that transfers signal/information from source to the sink. Example: wire, RF frequency, sound
SNR	Signal to noise ratio, compares level of signal to noise (which is unwanted). Often defined by ratio of signal to noise power, often expressed in decibels (dB). If SNR is greater than 0, there is more signal than noise.$SNR = \frac{P_{signal}}{P_{noise}}$
Recursion	A process that calls itself.
Digital filter	An algorithm that allows some data to pass while blocking others.
Linear filter	A filter linear when the output is a linear combination of the input to the filter.
Window	A mathematical function, which is zero outside of a certain interval and non-zero inside of this interval.
Z transform	Transforms a discrete signal in to series of real or complex numbers.
DSP	Digital signal processing.
LTI	Linear Time Invariant. Output of the system only depends on the input, not the time of the input.
ADC	Analog to digital converter.
Order	Amount of previous data used to calculate current result.

Transfer function $\mathcal{T}$	Mathematical function that models system response.
Impulse response	Output of a system when input is short.
DTFT	Discrete time Fourier transform.
Causal system	System only dependant on past and present input.
Group delay	Relation between frequency and phase delay.

1.2 Filtering

Function of a filter is to remove unwanted noise from a signal. Filters can be *digital* or *analog*. Analog filters use analog devices (e.g. capacitors, amplifiers) to filters analog signals. Digital filters are mathematical models that use algorithms and parameters. Compared to analog filters, digital filters are easier to design and simulate, however they cannot completely replace analog filters due to the delay that is caused by computation. In order to use a digital filter, analog signal must first be converted, using ADC, to a digital signal. Flowchart, shown in Figure 2, displays the basic idea of filtering.

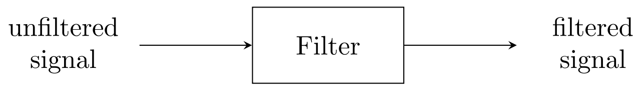


Figure 2: Filter flowchart

Advantages of digital filters, compared to analog, are:

1. Digital filters are programmable, their properties are not defined by hardware. They can be changed with software.

2. Digital filers are easier to design, simulate and test.

3. Environmental variables such as temperature, humidity have no influence on filtration.

4. They can operate on more than one type of signal.

Disadvantages of digital filters are:

1. Digital filters introduce a delay.

2. Processing speed is lower compared to analog filters.

3. Less smooth response compared to analog filters.

In practice, digital filtering is used in a variety of applications for various purposes, such as equalization, image sharpening, and system control. They help improve the quality of transmitted signals, reduce interference, and increase the efficiency of systems.

2 Theoretical background

2.1 Discrete signals

A discrete signal is defined at discrete points in time, not continuously. A discrete signal is a sequence of values at specific time intervals. Figure 3 shows the difference between discrete (red squares) and continuous signal (black line).

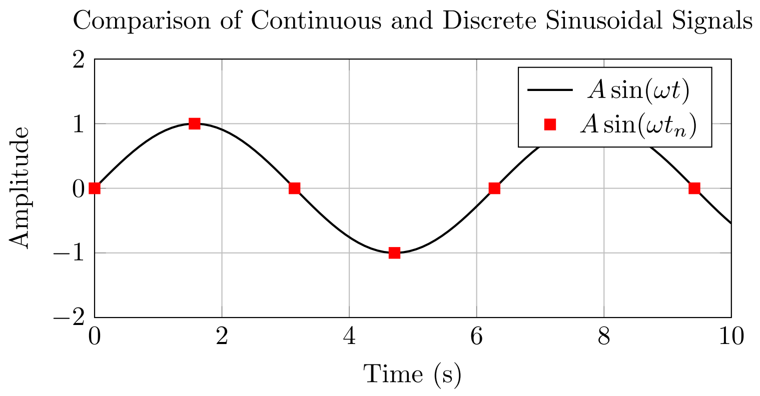


Figure 3: Comparison of Continuous and Discrete signals

Discrete signals are used in digital processing, as they can be represented as sets or arrays of numbers where one represents the value, and the other represents the time. Two signals shown in Figure 3 can be written as:

Continuous	$y = sin(t)$
Discrete (with formula)	$y = sin(t_n)$ where $t_n = i \cdot \Delta t$, $\Delta t = \pi$, $i \in \mathbb{N}$
Discrete (with array)	value y = [0,1,0,-1,1,0]
	time t = [1.57,3.14,4.71,6.28,7.85,9.42]

Discrete signals can be represented in time or frequency domain.

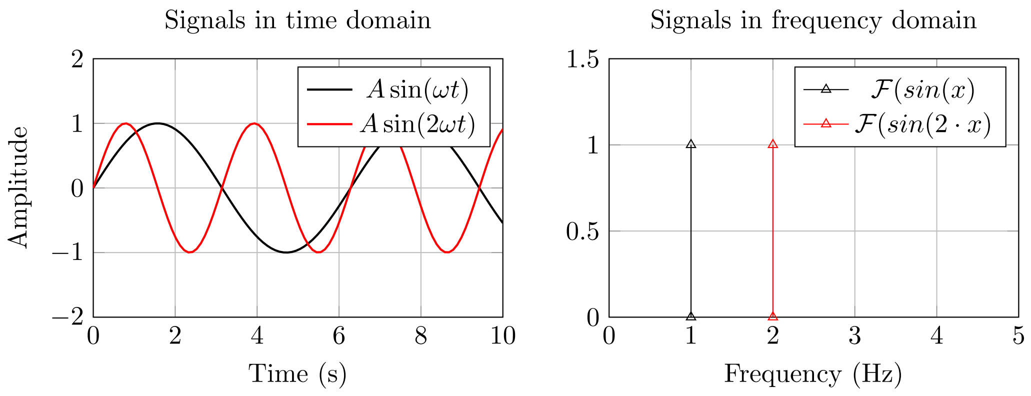


Figure 4: Representation of signals in time and frequency domain.

2.2 Discrete time systems

When a signal travels through a system it is transformed. System can be an algorithm or a hardware device. Mathematically this transformation is

$$y(n) = \mathcal{T}(x(n)) \tag{1}$$

where $y(n)$ is the result of the transformation $\mathcal{T}(\dots)$ of data $x(n)$. Transformation $\mathcal{T}$ represents a system. System can be *static* or *dynamic*,which can be described as functions whose output only depends on the current input - they have no memory. Output of a dynamic system depends on previous inputs - such function has memory. Equation 2 represents a static system, equation 3 is a dynamic system - previous value x(n-1) is needed to calculate the result.

$$y(n) = 3 \cdot x(n) \tag{2}$$

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

Discrete time systems can be time *variant* or *invariant*. Output of a time variant system depends both on the input and time of observation, output of time invariant system depends only on the input. Systems are *linear* if the principle of superposition can be applied to them - if the input is multiplied by a, then the output is multiplied by a. and if two inputs are added, their output is equal to the output of added inputs as shown on Figure 5.

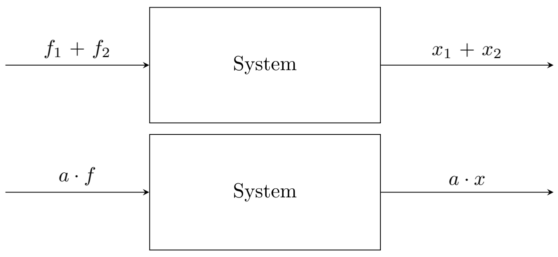


Figure 5: Linear system

2.3 Noise

Noise is any unwanted modification to the signal or information during capture, transmission or storage. Noise is usually random and carries no useful information. The type of noise is determined by how it affects the signal (e.g. additive, multiplicative, etc.) and by its statistical properties. Figure 6 shows signal with and without random noise.

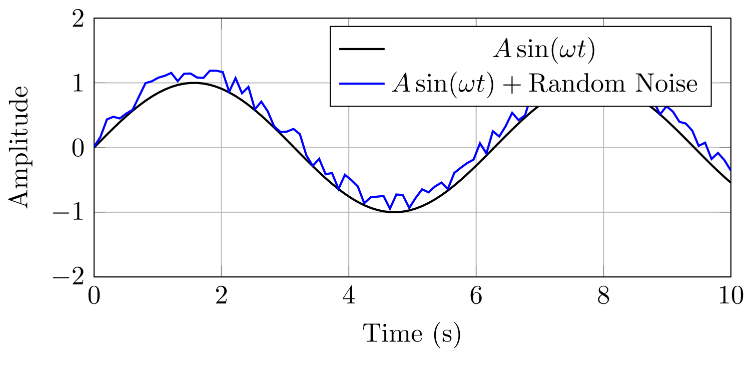


Figure 6: Continuous signal with and without added noise

2.3.1 Signal-to-noise ratio

Signal to noise ratio (*SNR*) is a measure used to describe the ratio of power in a signal. P represents the average power of signal or noise at the same point and time. SNR higher than 1 (0 dB) indicates more signal than noise.

$$\text{SNR} = \frac{P_{signal}}{P_{noise}} \tag{4}$$

2.4 Transfer function - Z transform

Z-transform is a mathematical transformation used to analyze LTI systems. It transforms a sequence of complex numbers to a complex function with a complex variable *z*. It enables us to preform operations on the sequence in the frequency domain. Z-transform is a generalization of *discrete time Fourier transform* (DTFT), in comparison to it, Z-transform can be calculated for more signals.

Z-transform is defined as 5, inverse z transform is defined as 6.

$$X(z) = \sum_{-\infty}^{\infty} x(n) \cdot z^{-n} \tag{5}$$

$$x(n) = \frac{1}{2\pi i} \oint_r X(z) z^{n-1} dz \tag{6}$$

where r is a contour in a complex plain. Inverse Z-transform is used to convert the complex function back to the original sequence. In order to solve equations with nonzero initial conditions the lower bound of 5 is changed to 0. This formulation 7 is called *unilateral* z transform.

$$X(z) = \sum_{-\infty}^{0} x(n) \cdot z^{-n} \tag{7}$$

Z-transform is rarely solved by integration, tables and builtin program functions (e.g. Matlab ztrans(x))are used more often. Table 1 shows some commonly used mathematical functions and their z-transform.

$f(n)$	$F(z)$
$\delta[n]$	1
a^n	$\frac{1}{1-az^{-1}}$
n	$\frac{z}{z-1}^2$
n^2	$\frac{z+1}{z^2-2z+1}$
e^{an}	$\frac{1}{1-aze^{-a}}$
$\sin(an)$	$\frac{aze^{-a}}{1-2aze^{-a}+aze^{-2a}}$
$\cos(an)$	$\frac{1-aze^{-a}}{1-2aze^{-a}+eze^{-2a}}$

Table 1: Z-transform lookup table of some common mathematical functions

2.5 Linear filters

Linear filters are all filters whose output is a linear transformation of input and are usually time invariant and obey the principle of superposition. They can be analysed using *linear time invariant* (LTI) systems theory. Every linear filter has a *transfer function* and a *impulse response.* Transfer function $\mathcal{T}$ is a mathematical function that transforms input, impulse response is the response system outputs if the input is an infinitely short impulse, usually represented with Dirac delta function ($\delta(t)$). Each filter has an *impulse response* time, which can be finite - filter stops responding after a some amount of time - or infinite.

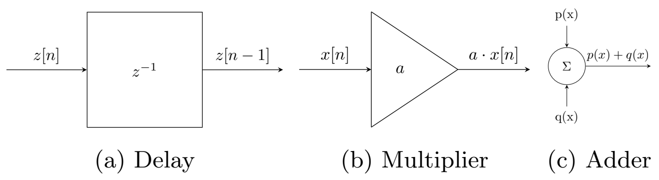


Figure 7: Basic building blocks of a linear filter

Filters can be used on *spatial* and *temporal* signals. Temporal signals are a function of time, spatial are time independent, an example of time independent signals are images, instead of time, data or signal depends on location - coordinates or pixels.

Output $y(n)$ of a filter is a linear combination of input $x(n)$ and a transformation. Most basic building blocks of filters are a delay element (Figure 7a) a multiplier (Figure 7b) and an adder (figure 7c). Each delay element has an *order*, which is equal to the exponent n. Number n is equal to the amount of previous samples filter is using to calculate the output. In real time applications the exponent is smaller or equal to zero. Delay elements cause a phase shift, which also depends on the frequency of the input. Multipliers simply multiply the input with the coefficient, adder ads two inputs.

2.5.1 Feedback and feedforward filters

Feedforward filters rely only on the current and previous inputs (e.g. x[n], x[n-1]) to calculate the result. Figure 8a shows a simple feedforward filter with order of *n = 1*. *Feedback* filters use a previous outputs (e.g. y[n-1]) to calculate the current result. Coefficient b_1 has a negative sign $(-)$ meaning it is subtracted not added. Feedback filters can have a non-linear response, and non steady state output, value of b_1 will influence stability and convergence. Feedforward filter always has a linear response, feedback filters have infinite impulse response. Both filters can have finite or infinite impulse response.

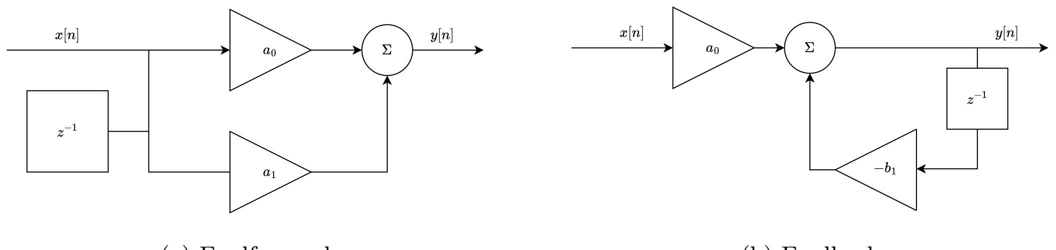


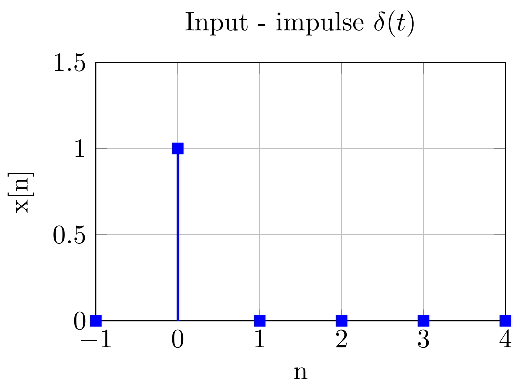
Figure 8: Feedforward and feedback filter designs

Compared to feedback , feedforward filters have higher computational cost, as they need both more time to calculate and store all of the needed data.

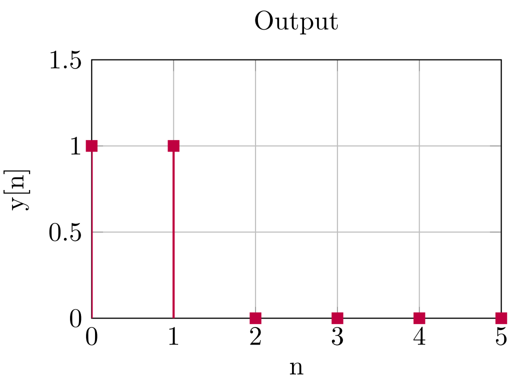
2.6 FIR - Finite impulse response

Finite impulse response time filters stop outputting a response after a certain duration of time. Output settles to zero after that. FIR filters can work both on continuous and discrete signals. For a *causal* filter, the output is depends on the current and past input. Order of the filter n determines how many past inputs are included in the current output. Figure 9a shows an impulse that is used as an input to some FIR filter and 9b shows filters output. After two cycles all y[n] are zero. Filter shown on 8a has a finite impulse response.

FIR filters are easier to design and implement, they have no stability issues and a linear phase response. However they are more computationally complex because they require more additions



(a) Input to the filter



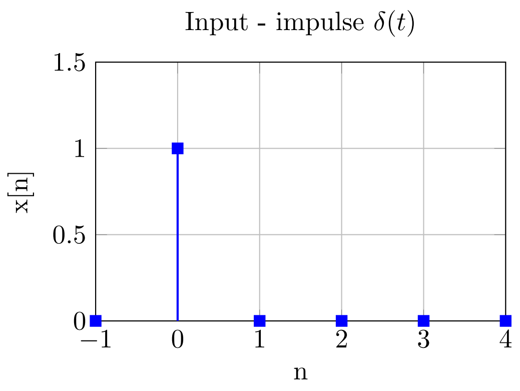
(b) Output of the filter

Figure 9: Response to an impulse of a FIR filter

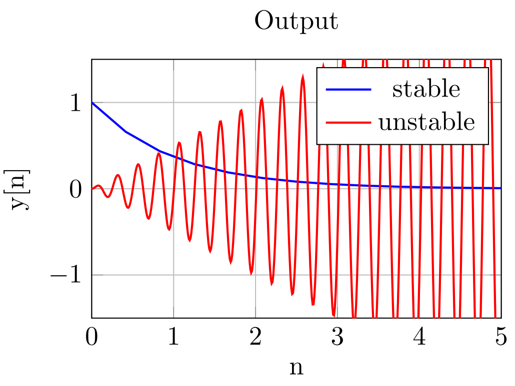
and multiplications. FIR filters have a longer delay or latency and higher memory consumption. FIR filters can only be digital, all analog filters have an infinite response.

2.7 IIR - Infinite impulse response

Infinite impulse response filters never stop responding to an impulse. Response can converge towards a steady state or it can diverge from it. Figure 10b shows an example of a stable output that is approaching steady state and an unstable output, that is diverging from a steady state.



(a) Input to the filter



(b) Output of the filter

Figure 10: Response to an impulse of a IIR filter

Filter shown on 8b has an infinite impulse response, if the response is stable or unstable depends on the coefficient b_1. All analog filters have an infinite impulse response. Compared to FIR filters, IIR have lower latency and lower computational cost. Disadvantages of IIR filters are non linear responses and possible issued with stability. They are commonly used in real time audio applications.

2.8 Windows

In signal processing, windows are used to extract a segment of a signal from the a longer time series. A window is applied to the entire signal, but only a smaller portion is used for analysis. Figure 11 shows a signal that has been recorded and a rectangular window, which multiplies the value of a signal at a certain time with 1 or 0. If the time value is outside of the window bounds value of the signal at that time is multiplied by 0, otherwise it is multiplied by 1.

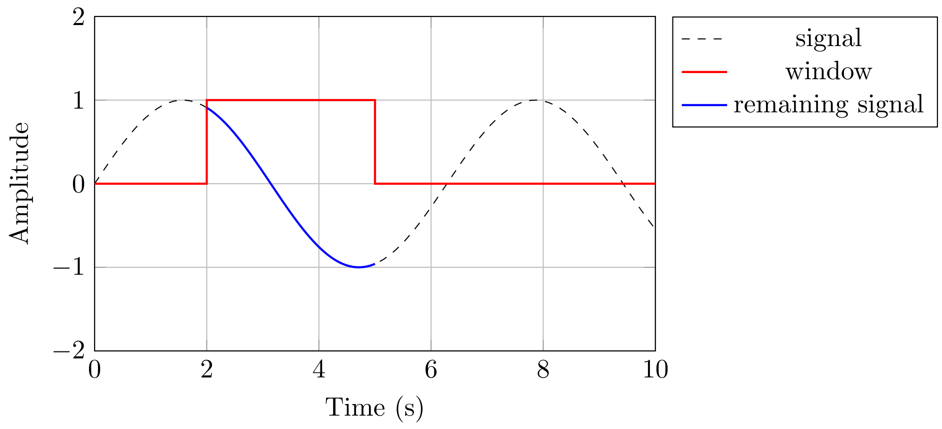


Figure 11: Signal with window

The same window can be implemented using python code. A function called *window* is defined, and it takes four parameters as input. If time $t \in [t_{lower}, t_{upper}]$, where t_{lower}, t_{upper} are the boundaries, is multiplied with c, which is equal to 1.

Code snippet 1: Rectangular window function in Python

```python
def window(t,x, lower, upper):
    """
    t -> current time;
    x -> current value;
    lower -> lower time boundary - t_lower;
    upper -> upper time boundary - t_upper;
    """
    c = 1;
    if lower < t and t < upper:
        #return signal values of x*c for all values inside the window
        return x * c
    else:
        #retun 0 for all values outside the window
        return 0
```

If c would be different or a function of time, it would change the shape of the window. Other windows shapes are exponential, Hamming, etc. Figure 12 shows different window shapes created using *scipy.signal.windows*. Rectangular windows can cause problems due to sharp transitions, which can cause ringing effects such as oscillations. Ringing can cause signal degradation and artefacts. In image processing artefacts can be seen as halos around edges and in audio artefacts can be heard as hissing or frequency distortions. To avoid there issues, windows with smoother transitions should be used.

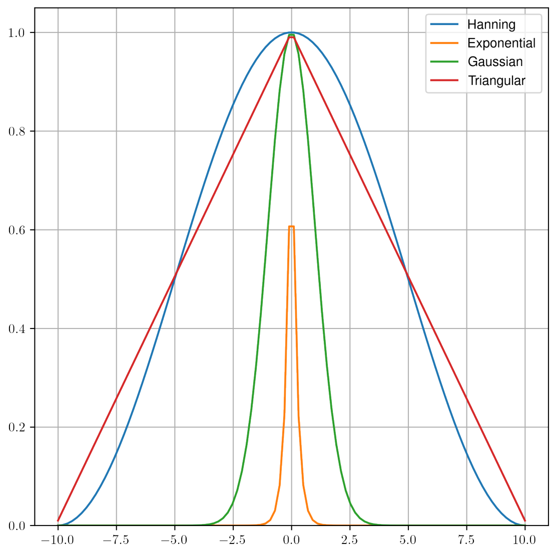
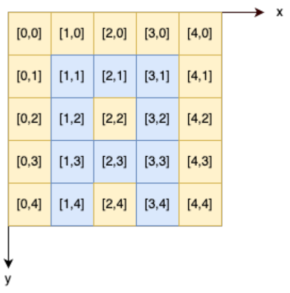


Figure 12: Examples of different window shapes

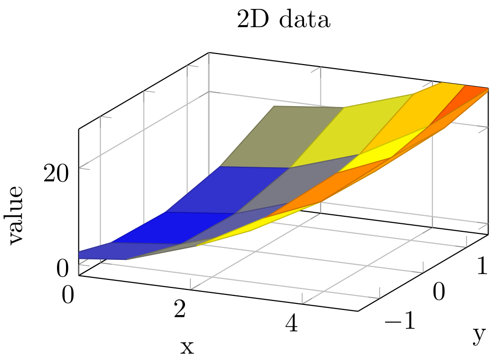
2.9 Discrete filters

2.9.1 Filter dimensions

Note: in some literature, filter dimensions are used instead of ordes.
Filters can operate on data that has more than one dimension. An example of two dimensional data is an image. Figure 13a shows an example of two dimensional data, each pixel has two coordinates $[x, y]$ and the value of it is its color. Figure 13b shows two dimensional data where the value is abstract. Example of one dimensional data is 3. It is possible to have data with even more dimensions. An example of that would be data with coordinates $[x, y, z]$ where a filtered value would be density.



(a) Sample 4×4 image



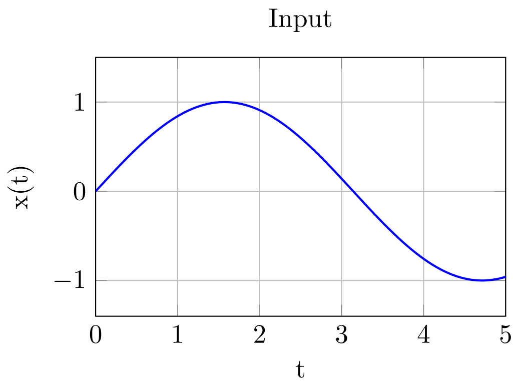
(b) 2D data

2.9.2 Gain

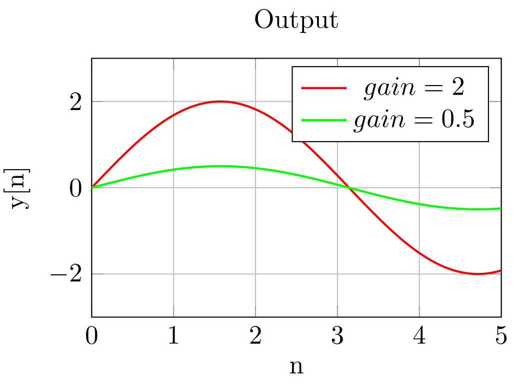
A gain filter is used to amplify or attenuate a signal by a certain factor. Gain is defined as the ratio of output and input amplitudes. Gain filters can be used in time or frequency domain. In time domain, output of a gain filter is a multiplied input with the gain factor. In the frequency domain input is modified by using the transfer function. Gain filter can be used to adjust the amplitude of the signal. Code snippet 2 is a python implementation of a gain filer in time domain. Output of this filter is shown in Figure 14.

Code snippet 2: Gain filter

```
def gain_filter(x, gain):
    """
    Gain filter in time domain
    x -> current value;
    gain -> gain factor;
    """
    return x * gain
```



(a) Input to the filter



(b) Output of the filter

Figure 14: Response

2.9.3 Delay

Delay filter delays the input signal for a certain amount n. It is made of one element, which is shown on figure 7a. Code snippet 3 show a simple delay filter created in Python. Use of this function is shown in Code snippet 4.

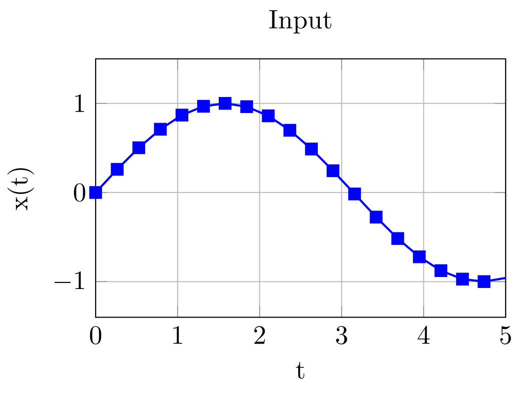
Code snippet 3: Delay filter

```
import numpy as np

def delayfilter(x,n):
    """
    x -> data shaped as an array/list;
    n -> delay;

    returns a delayed input as a list;
    """
    y = np.zeros(x.shape);
    y[n:] = x[:-n]
    return y
```

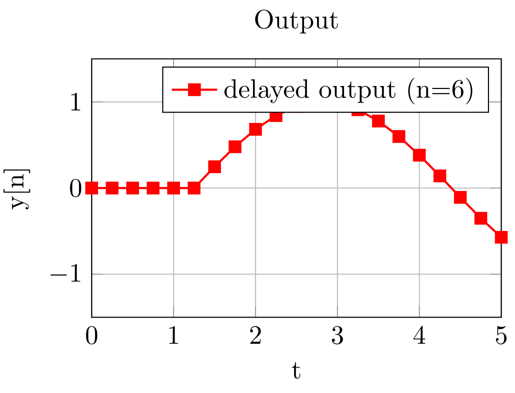
Code snippet 4: Use of delay function

```
input = np.array([1,2,3,4]);
print(delayfilter(input,2)) # prints -> array([0., 0., 1., 2.])
```

Figure 15 shows use of this filter on a signal shown on Figure 15a, Figure 15b shows a filtered/delayed signal.



(a) Input to the filter



(b) Output

Figure 15: Response of a delay filter

2.9.4 Moving Average

Moving average is a statistical method used to smooth a time series or a sequence of data points. Moving average is calculated by averaging data within a certain range or window. Moving average can be used to reduce noise or fluctuations. Code snippet 5 was used to calculate averages shown in Figure 16.

Code snippet 5: Simple moving average in Python.

```
import math
def mean(data):
    sum = 0;
    for i in data:
        sum += i
    return sum/len(data)

def movingaverage(signal, window_size):
    """
    signal -> array of numbers
    window_size -> size of window, must be integer
    """
    i = math.ceil(window_size/2);
    x_axis = np.zeros(len(signal))
    r = np.zeros(len(signal))
    while i < len(signal) - math.ceil(window_size/2):
        cur_data = signal[i:i+window_size]
        r[i] = mean(cur_data)
        x_axis[i] = i;
        i += 1;
    return r,x_axis # returns averages @ points on x axis
```

Function *movingaverage()* returns moving average at each point. Edges of data are not handled.

2.9.5 High/Low pass

Figure 17 shows a high/low pass feedforward filter, made out of components shown on Figure 7. Each multiplier has one coefficient - a_0,a_1. Delay element has an order of one, and its initial value is set to zero. Filtered result is a sum of current value $x[n] \cdot a_0$ and $x[n-1] \cdot a_1$. Transfer function of this filter is show in shown in equation 8.

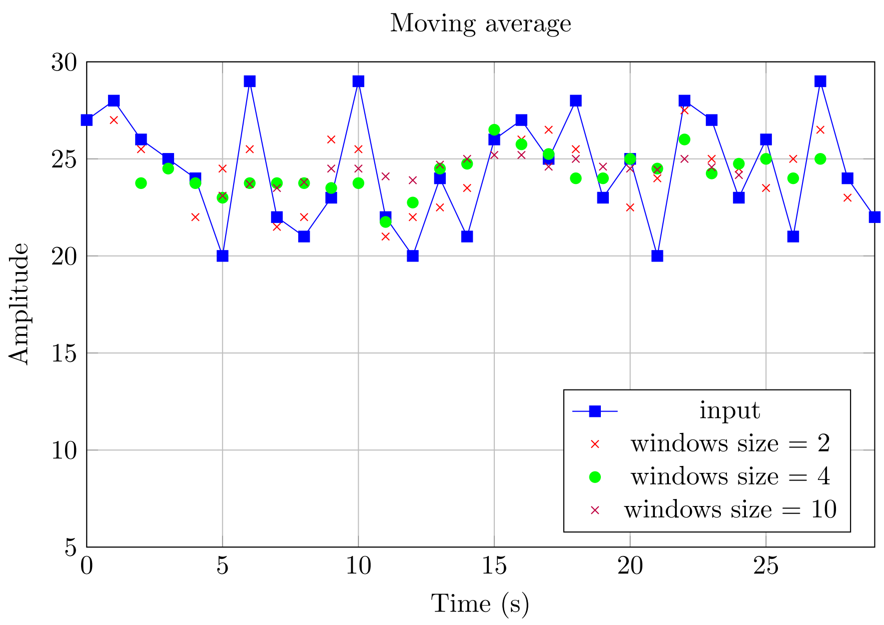


Figure 16: Example of a moving average on random data

$$y[n] = x[n] \cdot a_0 + x[n-1] \cdot a_1 \tag{8}$$

Values of coefficients a_0 and a_1 determine if high or low frequencies pass. In Python code, this

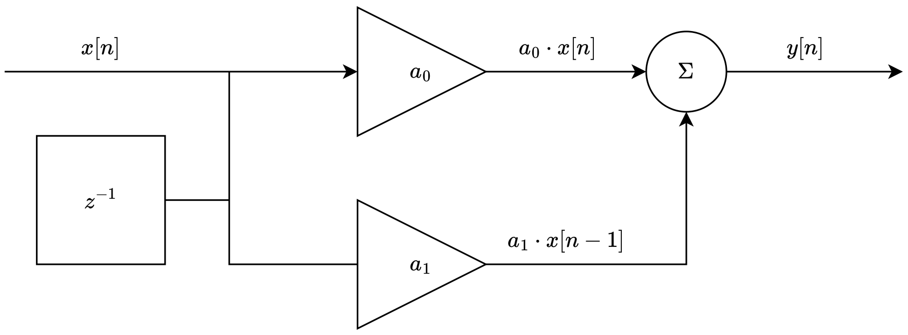


Figure 17: Low/High pass filter

filter will be implemented as shown in code snippet 6.

Code snippet 6: Feedforward filter

```
def filter(x, a0,a1):
    y = np.zeros(len(x))
    for n in range(0,len(x)):
        y[n] = x[n] * a0 + x[n-1] * a1 # transfer
    return y
```

Impulse response
In order to determine how a filter responds to an impulse, we can simulate it using Python. Code snippet 7 generates an impulse of a certain length, where all but first values are zero and passes it to the function *filter()*. Filters response is shown in figure 18

Code snippet 7: Generating an impulse response

```
samples = 1024 # size of data
impulse = np.zeros(samples) #impulse -> all initially zeros
impulse[0] = 1 # first value

a0,a1 = 0.5,0.5
impulse_response = filter(impulse,a0,a1)
```

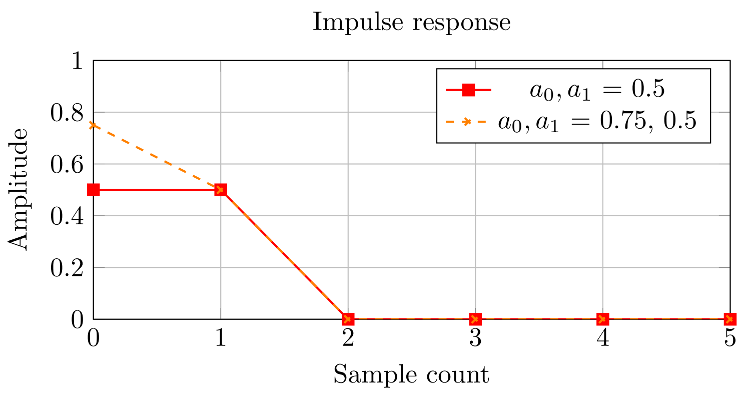


Figure 18: Impulse response

Impulse response depends on the value of a_0 and a_1.

Frequency and phase response

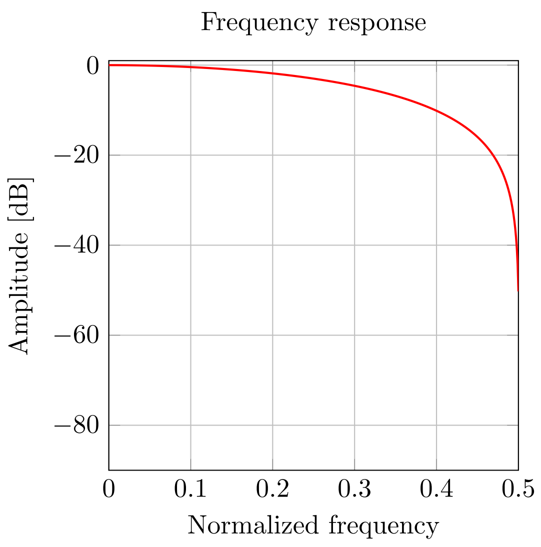
To determine filters frequency and phase response, Fourier transformation is preformed on the impulse response. Code snippet 8 shows this transformation implemented in python using *scipy.fftpack*.

Code snippet 8: Fourier transformation of the impulse response

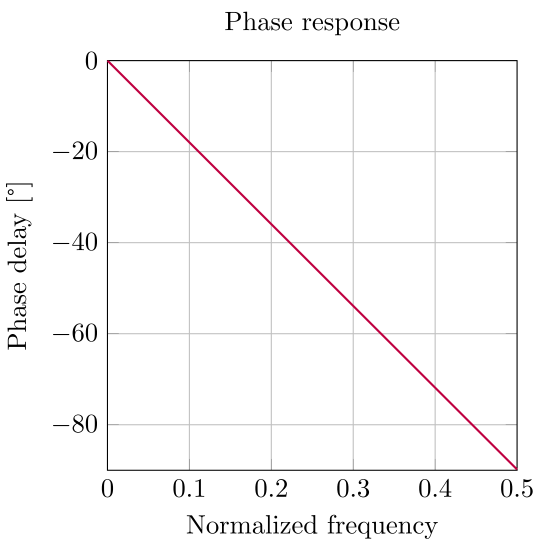
```
from scipy.fftpack import fft
a0 = a1 = 0.5
fft_response = fft(filter(impulse,a0,a1)) #fft
freq = np.linspace(0, 0.5, fft_response.size//2)
# amplitude in dB
amplitude=(20*np.log10(np.abs(fft_response)))[:fft_response.size//2]
```

Figure 19 shows the response of a low pass filter. Coefficients a_0, a_1 determine if the filter is low or high pass. Changing the coefficients to $a_0 = 0.5$ and $a_1 = -0.5$. changes from a low to a high pass filter

Both low and high pass responses have similar shapes. Frequency responses are mirrored. The rate of change of the phase delay, figures 19b and 20b, is the same. This rate of change, is often called *group delay*. Frequency axis on figures 19a and 20a is normalized, value of one would be equal to the sampling frequency, value of 0.5, which is shown on the graph, represents *Nyquist frequency*, which is the maximum frequency which can be accurately represented or transmitted in a digital signal.

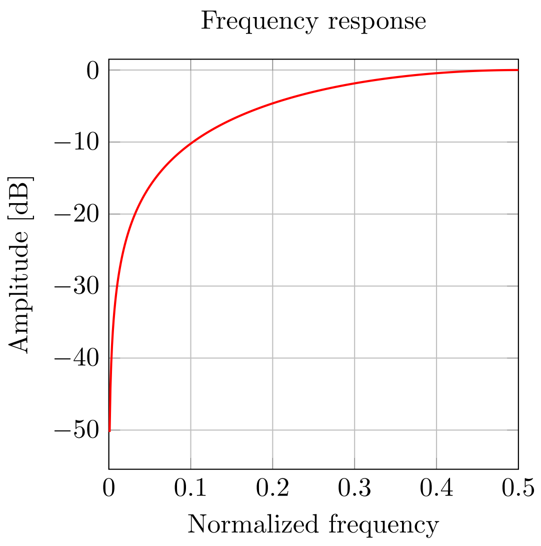


(a) Frequency response

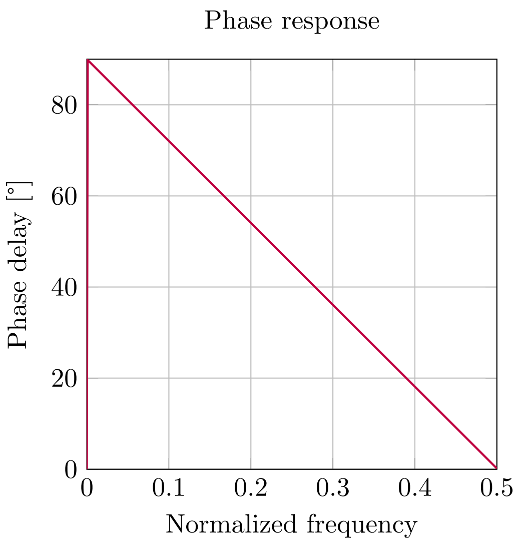


(b) Phase response

Figure 19: Frequency and phase response of a low pass filter



(a) Frequency response



(b) Phase response

Figure 20: Frequency and phase response of a high pass filter

3 Examples

3.1 Image

3.1.1 Grayscale

In order to apply a grayscale filter to an image, value of each pixel is replaced with an average of it. To a computer, an image is a two dimensional array of smaller arrays, which represent pixels. Each RGB pixel can be described by three values, each for one color. Values range from 0 to 255. To convert an image from RGB to grayscale, RGB values of each pixel must be replaced by one value - a weighted average of all three color. Code snippet 9 shows used libraries and opening and converting the imported image to an array.

Code snippet 9: Reading the image.

```
import numpy as np
from PIL import Image

#open
img = Image.open("image.jpg") #example image
nd = np.asarray(img) # image saved as array
```

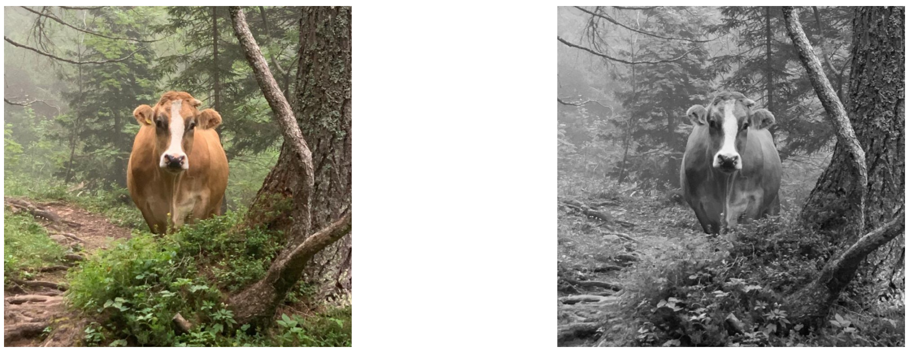
Code snippet shows the conversion to grayscale. In the imported image, each pixel is represented by a one dimensional array, formatted as $[R, G, B]$. New values are calculated by the equation 9. To convert the pixel to gray, all values R, G, B are replaced by $GRAY$.

$$GRAY = R \cdot 0.07 + G \cdot 0.72 + B \cdot 0.21 \tag{9}$$

Code snippet 10: Conversion to grayscale.

```
#converion
greyscale = np.zeros((nd.shape[0], nd.shape[1],3), dtype=np.uint8)
for i in range(0,numdata.shape[0]): # x axis
    for j in range(0, nd.shape[1]): #y axis
        gray = nd[i][j][0]* 0.07+ nd[i][j][1]* 0.72+ nd[i][j][2]* 0.21
        greyscale[i][j] = gray
#export
im = Image.fromarray(greyscale)
im.save("grayscale.png")
```

Figure 21 shows the result.



(a) Imported image (b) Grayscale image

Figure 21: Conversion to grayscale

3.1.2 Low/High pass filter

To apply a low/high pass filter to the imported image, each pixel has to be written as a single number. It is first converted to grayscale, then each value is copied to a new array,this is shown in code snippet 11.

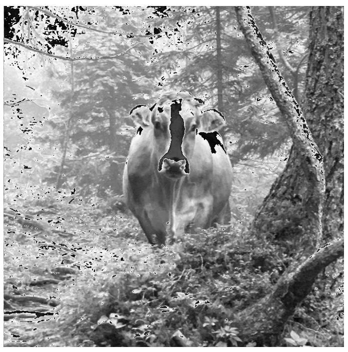
Code snippet 11: Converison of image to 8bit grayscel image.

```
#Convert pixel shape from [GRAY, GRAY, GRAY] to array [GRAY,... , GRAY]

im2 = np.zeros((greyscale.shape[0],greyscale.shape[1]), dtype=np.uint8)
for i in range(0,greyscale.shape[0]):
    for j in range(0, greyscale.shape[1]):
        im2[i][j] = greyscale[i][j][0]
```

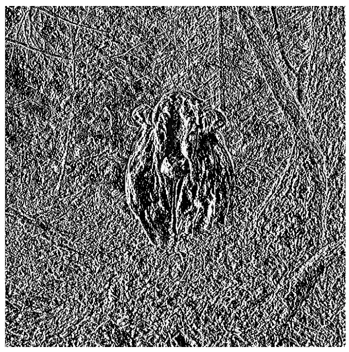
To filter out low or high values, each line of pixels is passed to the function *filter(array, a0,a1)*, shown in code snippet 6. Code snippet 12 shows the process of filtering high or low values from the image. Each line of pixels is filtered and written to a new image. Results of filtration are shown in Figure 22. Figure 21b was used as the input.

Code snippet 12: Filtering high/low values from the image.

```
filtered_image = np.zeros((im2.shape[1],im2.shape[0]), dtype=np.uint8)
for i in range(0,im2.shape[1]):
    filtered_image[i] = filter(im2[i], 0.7,-0.7).astype(np.uint8)
```



(a) Low-pass filter



(b) High-pass filter

Figure 22: Low/high pass filtration of imported image.

Coefficients show in table 2 were used.

Low-pass	$a_0 = 0.7, a_1 = 0.7$
High-pass	$a_0 = 0.7, a_1 = -0.7$

Table 2: Filter coefficients.

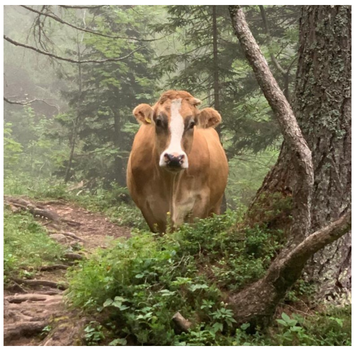
Note: Each line of the image is filtered separately. Filtering is applied to a 1D array.

3.1.3 2D Blur filter using Pillow

In order to demonstrate two dimensional filtration, a blur filter will be applied to the image. Two dimensional filtration is preformed on multiple pixels, that are not in the same line, but in a certain area of the image. Figure 23 shows the original and the blurred image. Code used to blur the input is shown in code snippet 13.

Code snippet 13: Blur filter using PIL.

```
from PIL import ImageFilter

img = Image.open("image700.jpg")
blured_image = img.filter(ImageFilter.BLUR)
blured_image.show()
```


(a) Original image

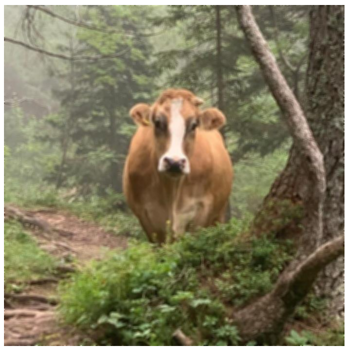

(b) Blurred image

Figure 23: Blurred image.

3.2 Sound

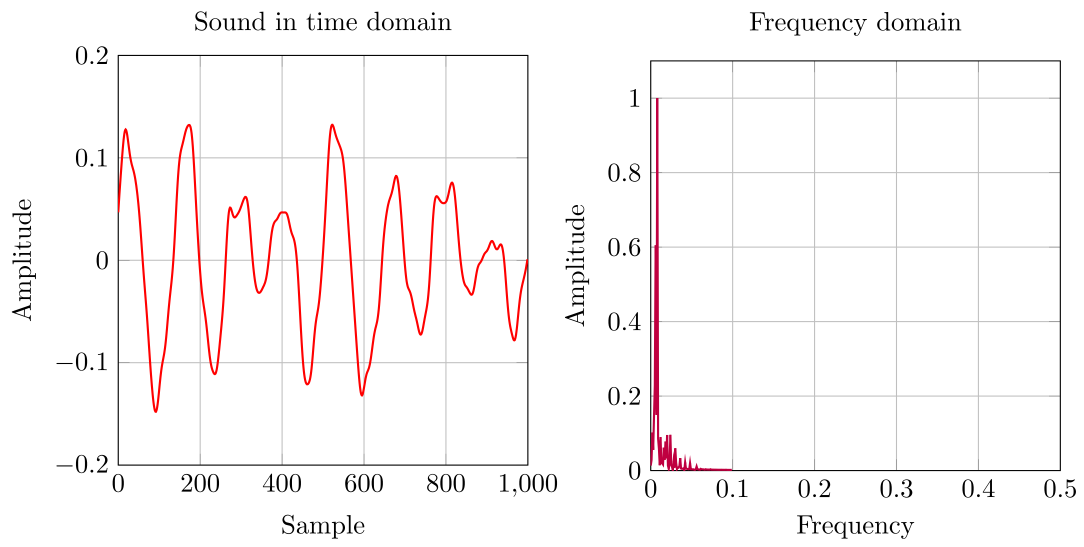
3.2.1 Low-pass filter

To remove low frequencies from a sound sample, it will be normalized and converted from time to frequency domain using FFT. Figure 24 shows the time sample in both domains. Filtration in shown in code snippet 14.

Code snippet 14: Sounnd

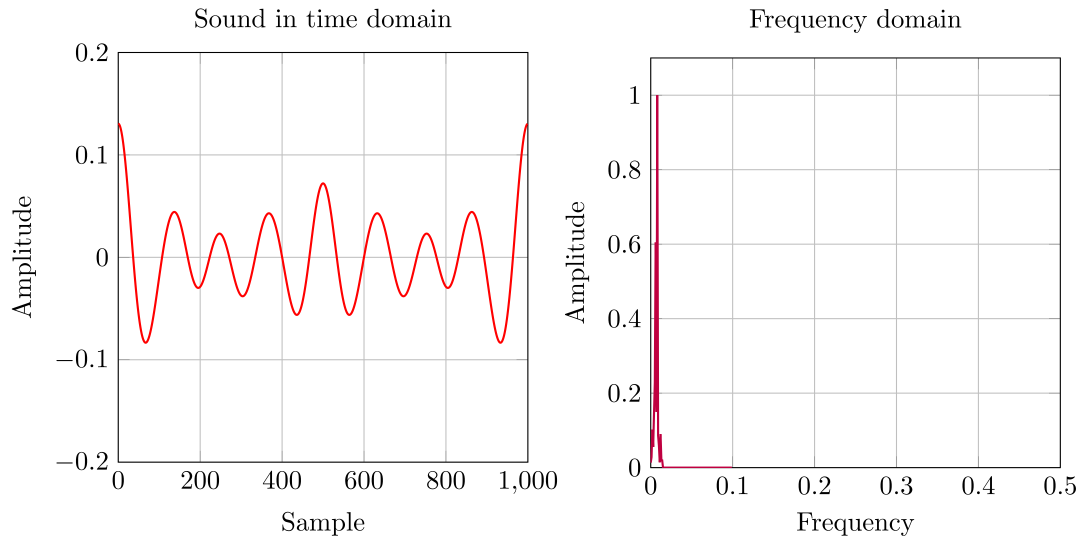
```
import numpy as np
import matplotlib.pyplot as plt

sound_sample = np.loadtxt("sound_sample.txt") # original -> 24a
fft = np.abs(np.fft.rfft(sound_sample))
fft = fft/max(fft)
freq = np.fft.rfftfreq(sound_sample.shape[0]) # fft -> 24b

fft[15:] = 0 # cuts higher frequencies -> 25b
sound_out = np.fft.irfft(fft) # reverse fft -> 25a
```

In order to remove the higher frequencies, we will replace all higher frequencies, in frequency domain, with zeros and then convert the signal back to time domain. Figure 25 shows the result of filtration. All frequencies and amplitudes are normalized.



(a) Time domain (b) Frequency domain, only a small section is shown

Figure 24: Example sound wave in time and frequency domain.



(a) Time domain (b) Frequency domain, only a small section is shown

Figure 25: Filtered sound wave in time and frequency domain.