Digital Signal Processing

Release 0.0

Sascha Spors

February 13, 2016

Table of Contents

This collection contains the lecture notes to the masters course Digital Signal Processing, Institute of Communications Engineering, Universität Rostock read by Sascha Spors. The notes are provided as Jupyter notebooks using IPython 3 as Open Educational Resource. Feel free to contact me if you have questions or suggestions.

- Reference Card Discrete Signals and Systems
- Reference Card Random Signals and LTI Systems

CHAPTER 1

Spectral Analysis of Deterministic Signals

The analysis of the spectral properties of a signal plays an important role in signal processing. Some application examples are

- Spectrum analyzer
- Detection of (harmonic) signals
- Estimation of fundamental frequency and harmonics
- Spectral suppression: acoustic echo suppression, noise reduction, ...

Spectral analysis often applies the discrete Fourier transformation (DFT) onto discrete finite-length signals in order to determine the spectrum and its magnitude.

1.1 The Leakage Effect

Spectral leakage is a fundamental effect of the DFT. It limits the ability to detect harmonic signals in signal mixtures. In order to discuss the properties of the DFT, the transition from the Fourier transform applied to an analytic continuous signal to the DFT applied to a sampled finite-length signal is investigated.

1.1.1 Continuous Signals and the Fourier Transformation

We first consider the spectrum of one single harmonic signal. For the continuous case this is given by the complex exponential function

$$x(t) = \mathrm{e}^{\,\mathrm{j}\,\omega_0\,t}$$

where $\omega_0 = 2\pi f$ denotes its angular frequency. The Fourier-Transformation of the exponential function is

$$X(\mathrm{j}\,\omega) = \int_{-\infty}^{\infty} x(t)\,\mathrm{e}^{\,-\mathrm{j}\,\omega\,t}\mathrm{d}t = 2\pi\;\delta(\omega - \omega_0)$$

The spectrum consists of a single Dirac impulse, hence a clearly isolated and distinguishable event.

1.1.2 Sampled Signals and the Discrete-Time Fourier Transformation

Now lets consider sampled signals. The discrete exponential signal is derived from its continuous counterpart by equidistant sampling $x[k] := x(kT)$ with the sampling interval T

$$x[k] = \mathrm{e}^{\,\mathrm{j}\,\Omega_0\,k}$$

where $\Omega_0 = \omega_0 T$ denotes the normalized angular frequency. The discrete-time Fourier transform (DTFT) is the Fourier transformation of a sampled signal. For the exponential signal it is given as

$$X(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{k=-\infty}^{\infty} x[k]\,\mathrm{e}^{\,-\mathrm{j}\,\Omega\,k} = 2\pi \sum_{n=-\infty}^{\infty} \delta((\Omega - \Omega_0) - 2\,\pi\,n)$$

The spectrum of the DTFT is periodic due to sampling. As a consequence, the transformation of the discrete exponential signal consists of a series Dirac impulses. For the region of interest $-\pi < \Omega \leq \pi$ the spectrum consists of a clearly isolated and distinguishable event, as for the continuous case.

The DTFT cannot be realized in practice, since is requires the knowledge of the signal $x[k]$ for all time instants k. The DFT can be derived from the DTFT in two steps

1. truncation (windowing) of the signal
2. sampling of the DTFT spectrum

The consequences of these two steps are investigated in the following two sections.

1.1.3 Finite-Length Discrete Signals and the Discrete-Time Fourier Transformation

Truncation of the signal $x[k]$ to a length of N samples is modeled by multiplying the signal with a window function $w[k]$ of length N

$$x_N[k] = x[k] \cdot w[k]$$

where $x_N[k]$ denotes the truncated signal. Its spectrum $X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ can be derived from the multiplication theorem of the DTFT as

$$X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{1}{2\pi} X(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \circledast W(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

where $\circledast$ denotes the cyclic/circular convolution. For a hard truncation of the signal to N samples the window function $w[k] = \mathrm{rect}_N[k]$ yields

$$W(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathrm{e}^{\,-\mathrm{j}\,\Omega\,\frac{N-1}{2}} \cdot \frac{\sin(\frac{N\,\Omega}{2})}{\sin(\frac{\Omega}{2})}$$

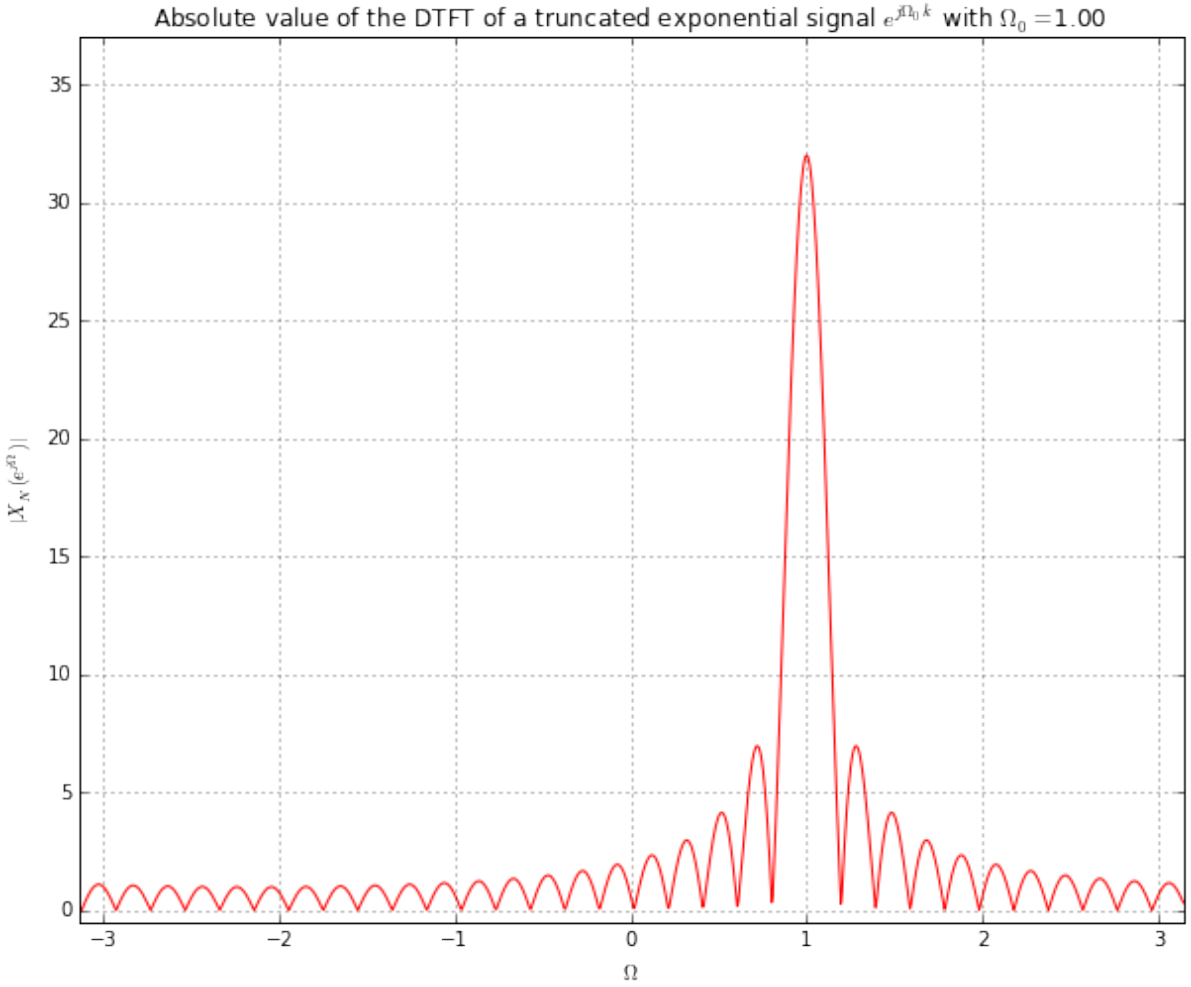
Introducing the DTFT of the exponential signal into above findings, exploiting the properties of Dirac impulses and the cyclic convolution allows to derive the DTFT of a truncated exponential signal

$$X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathrm{e}^{\,-\mathrm{j}\,(\Omega-\Omega_0)\,\frac{N-1}{2}} \cdot \frac{\sin(\frac{N\,(\Omega-\Omega_0)}{2})}{\sin(\frac{(\Omega-\Omega_0)}{2})}$$

Above equation is evaluated numerically in order to illustrate the properties of $X_w(\mathrm{e}^{\,\mathrm{j}\,\Omega})$

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        Om0 = 1  # frequency of exponential signal
        N = 32  # length of signal


        # DTFT of finite length exponential signal (analytic)
        Om = np.linspace(-np.pi, np.pi, num=1024)
        XN = np.exp(-1j*(Om-Om0)*(N-1)/2)*(np.sin(N*(Om-Om0)/2))/(np.sin((Om-Om0)/2))

        # plot spectrum
        plt.figure(figsize = (10, 8))
```

```
plt.plot(Om, abs(XN), 'r')
plt.title(r'Absolute value of the DTFT of a truncated exponential signal $e^{j \
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$|X_N(e^{j \Omega})|$')
plt.axis([-np.pi, np.pi, -0.5, N+5])
plt.grid()
```



Excercise

- Change the frequency `Om0` of the signal and rerun the cell. What happens?
- Change the length `N` of the signal and rerun the cell. What happens?

The maximum absolute value of the spectrum is located at the frequency Ω_0. It should become clear that truncation of the exponential signal leads to a broadening of the spectrum. The shorter the signal the wider the mainlobe becomes.

1.1.4 The Leakage Effect of the Discrete Fourier Transformation

The DFT can be derived from the DTFT $X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the truncated signal by sampling the DTFT equiangularly at (angles) $\Omega = \mu \frac{2\pi}{N}$

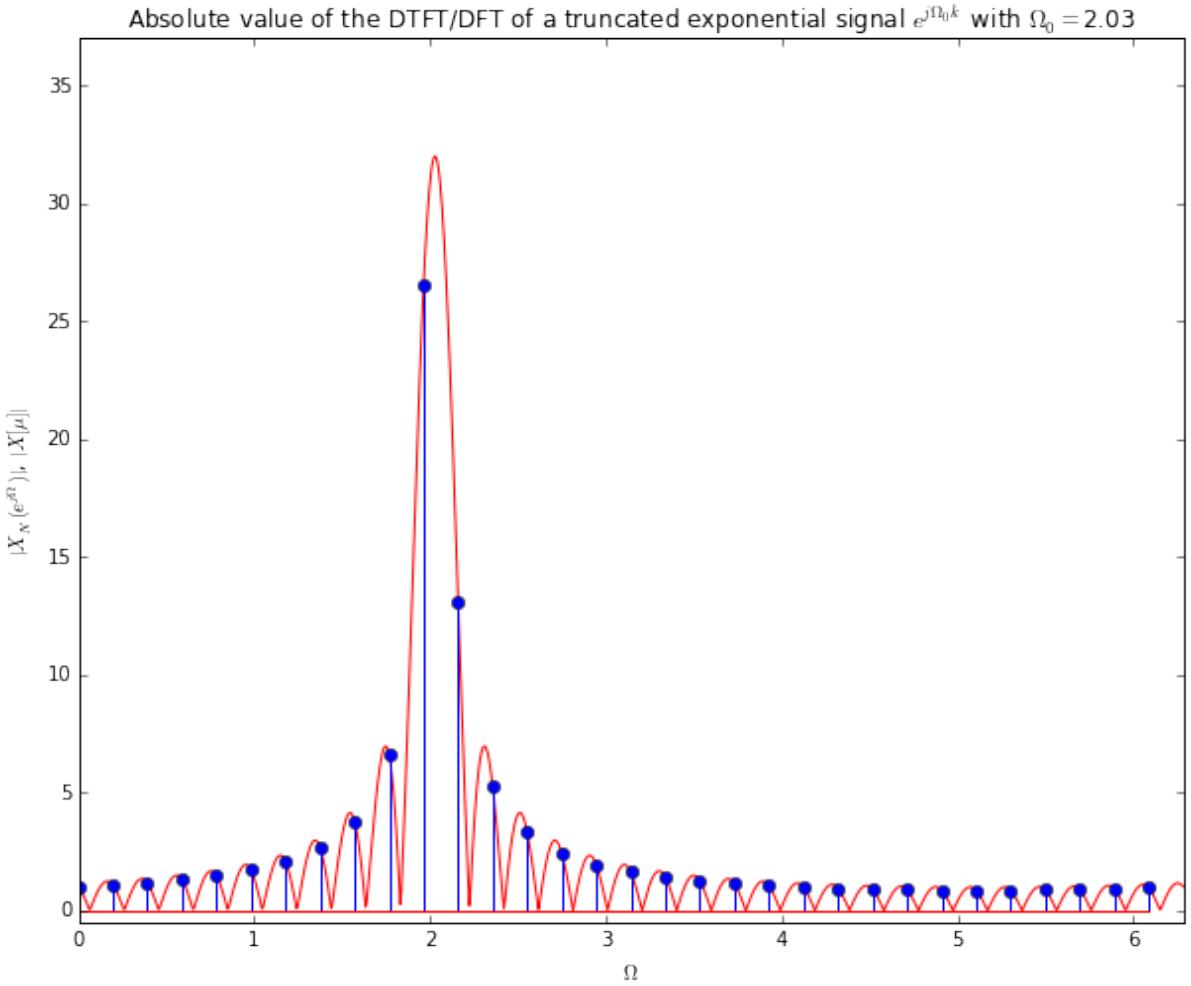
$$X[\mu] = X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega})\big|_{\Omega=\mu\frac{2\pi}{N}}$$

For the DFT of the exponential signal we finally get

$$X[\mu] = \mathrm{e}^{\,\mathrm{j}\,(\Omega_0-\mu\frac{2\pi}{N})\frac{N-1}{2}} \cdot \frac{\sin(\frac{N\,(\Omega_0-\mu\frac{2\pi}{N})}{2})}{\sin(\frac{\Omega_0-\mu\frac{2\pi}{N}}{2})}$$

The sampling of the DTFT is illustrated in the following example. Note, the normalized angular frequency Ω_0 has been expressed in terms of the periodicity P of the exponential signal $\Omega_0 = P \frac{2\pi}{N}$.

```
In [2]: N = 32  # length of the signal
        P = 10.33  # periodicity of the exponential signal
        Om0 = P*(2*np.pi/N)  # frequency of exponential signal


        # DTFT of finite length exponential signal (analytic)
        Om = np.linspace(0, 2*np.pi, num=1024)
        Xw = np.exp(-1j*(Om-Om0)*(N-1)/2)*(np.sin(N*(Om-Om0)/2))/(np.sin((Om-Om0)/2))

        # DFT of the exponential signal by FFT
        x = np.exp(1j*Om0*np.arange(N))
        X = np.fft.fft(x)
        mu = np.arange(N) * 2*np.pi/N

        # plot spectra
        plt.figure(figsize = (10, 8))
        plt.hold(True)

        plt.plot(Om, abs(Xw), 'r')
        plt.stem(mu, abs(X))
        plt.title(r'Absolute value of the DTFT/DFT of a truncated exponential signal $e^
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$|X_N(e^{j \Omega})|$, $|X[\mu]|$')
        plt.axis([0, 2*np.pi, -0.5, N+5]);

        plt.show()
```



Exercise

- Change the periodicity `P` of the exponential signal and rerun the cell. What happens if the periodicity is an integer? Why?
- Change the length `N` of the DFT? What happens?
- What conclusions can be drawn for the analysis of exponential signals by the DFT?

You should have noticed that for an exponential signal whose periodicity is an integer $P \in \mathbb{N}$, the DFT consists of a discrete Dirac pulse $X[\mu] = \delta[\mu - P]$. In this case, the sampling points coincide with the maximum of the main lobe or the zeros of the DTFT. For non-integer P, hence non-periodic exponential signals with respect to the signal length N, the DFT has additional contributions. The shorter the length N, the wider these contributions are spread in the spectrum. This smearing effect is known as *leakage effect* of the DFT. This effect limits the achievable frequency resolution of the DFT when analyzing signals mixtures with more than one exponential signal. This is illustrated in the following.

1.1.5 Analysis of Signal Mixtures by the Discrete Fourier Transformation

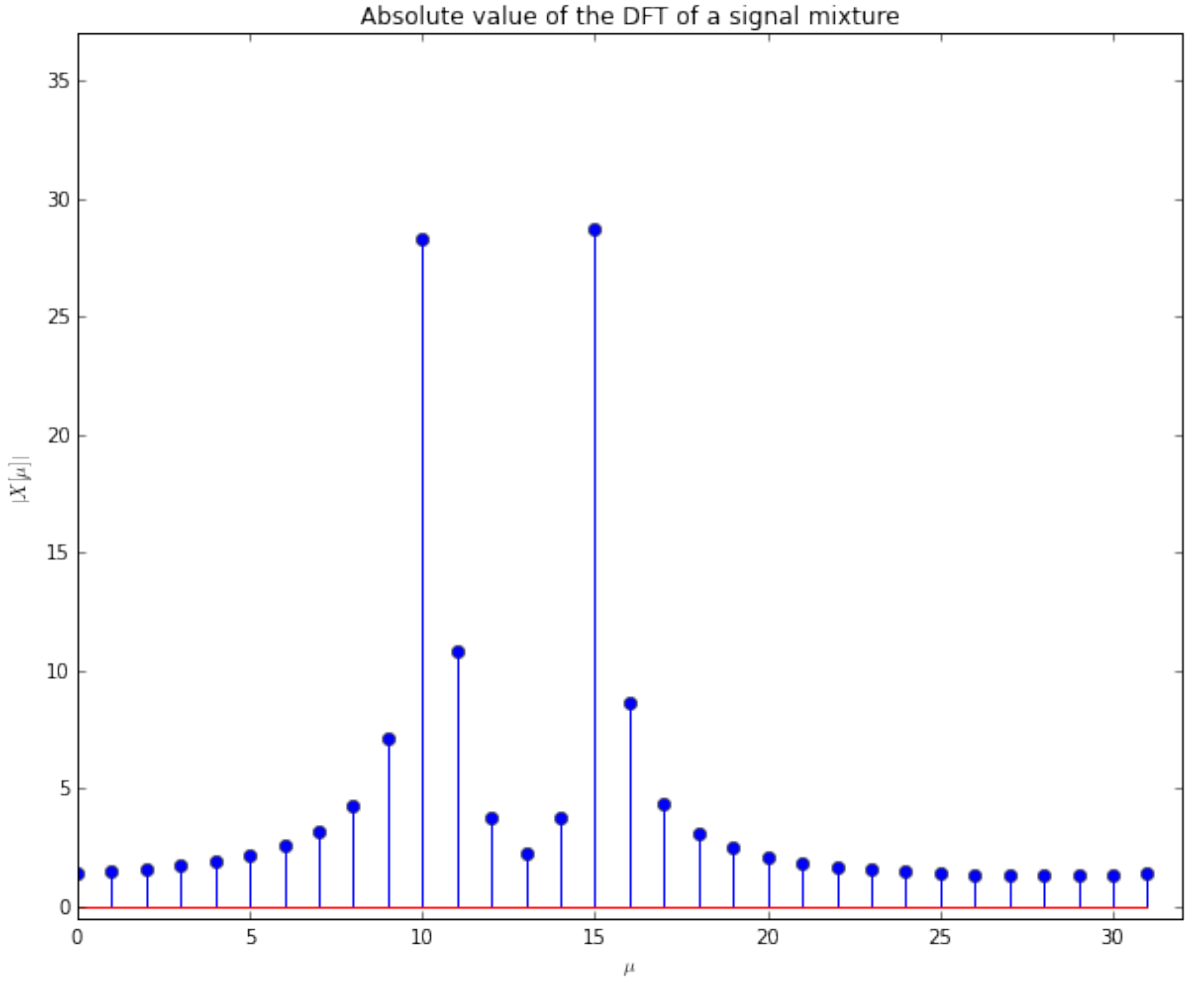
In order to discuss the implications of the leakage effect when analyzing signal mixtures, the superposition of two exponential signals with different amplitudes and frequencies is considered. For convenience, a function is defined that calculates and plots the magnitude spectrum

```
In [3]: def dft_signal_mixture(N, A1, P1, A2, P2):
            # N: length of signal/DFT
            # A1, P1, A2, P2: amplitude and periodicity of 1st/2nd complex exponential
```

```
    # generate the signal mixture
    Om0_1 = P1*(2*np.pi/N)  # frequency of 1st exponential signal
    Om0_2 = P2*(2*np.pi/N)  # frequency of 2nd exponential signal
    k = np.arange(N)
    x = A1 * np.exp(1j*Om0_1*k) + A2 * np.exp(1j*Om0_2*k)

    # DFT of the signal mixture
    mu = np.arange(N)
    X = np.fft.fft(x)

    # plot spectrum
    plt.figure(figsize = (10, 8))
    plt.stem(mu, abs(X))
    plt.title(r'Absolute value of the DFT of a signal mixture')
    plt.xlabel(r'$\mu$')
    plt.ylabel(r'$|X[\mu]|$')
    plt.axis([0, N, -0.5, N+5]);
```

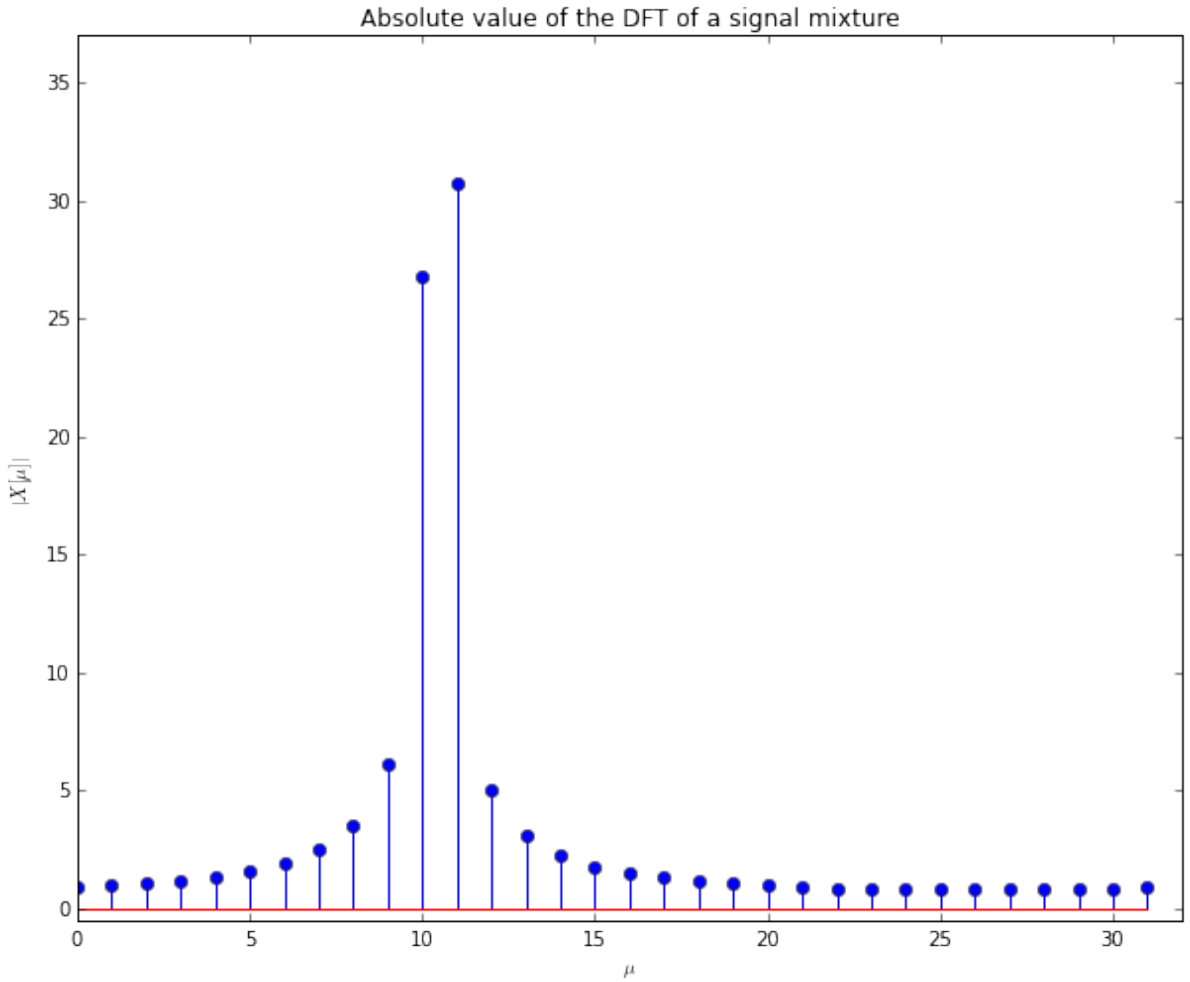
Lets first consider the case that the frequencies of the two exponentials are rather apart

```
In [4]: dft_signal_mixture(32, 1, 10.3, 1, 15.2)
```



Investigating the magnitude spectrum one could conclude that the signal consists of two major contributions at the frequencies $\mu_1 = 10$ and $\mu_2 = 15$. Now lets take a look at a situation when the frequencies are closer together

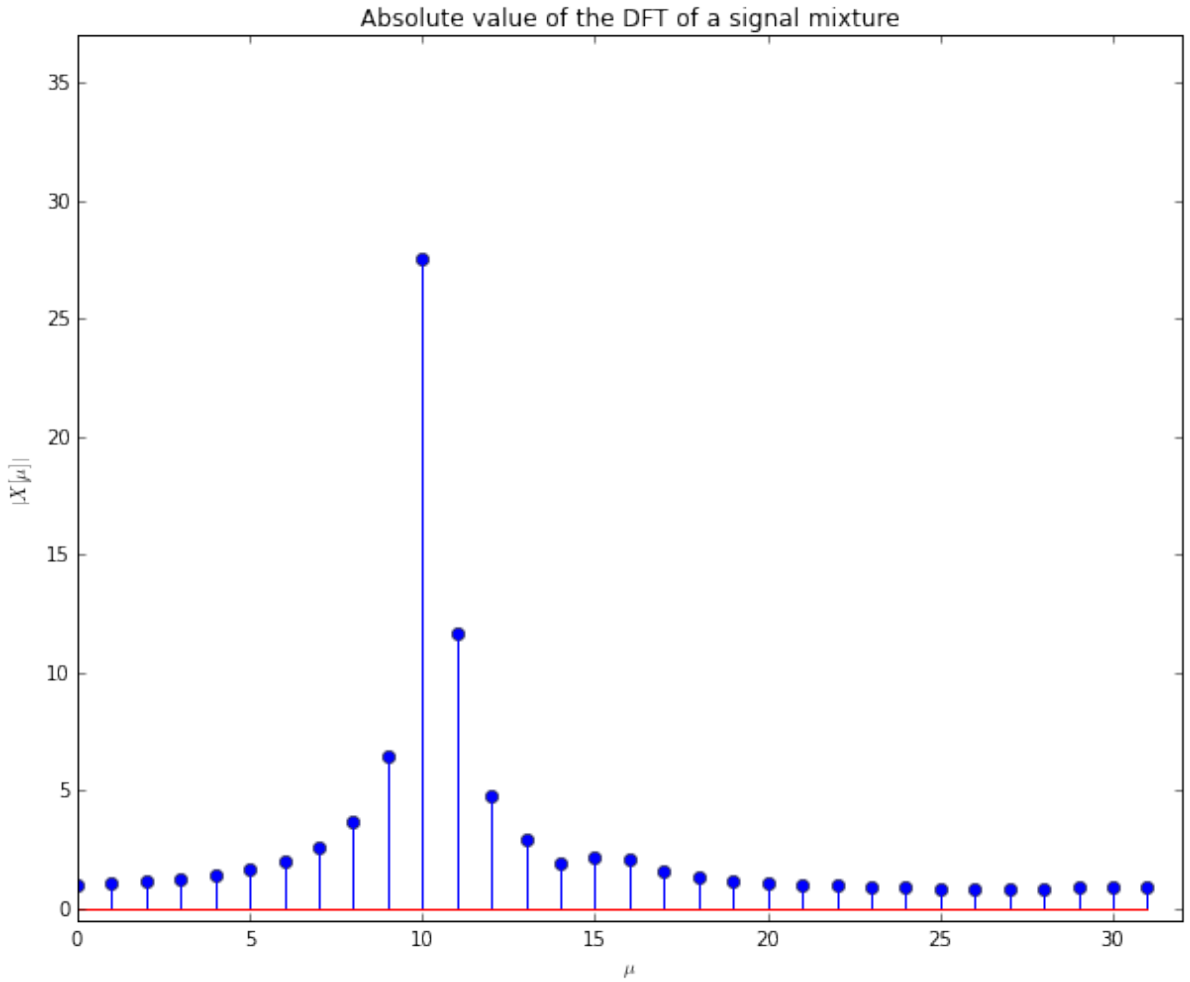
```
In [5]: dft_signal_mixture(32, 1, 10.3, 1, 10.9)
```



From visual inspection of the spectrum it is rather unclear if the mixture consists from one or two exponential signals. So far the levels of both signals where chosen equal.

Lets consider the case that the second signal has a much lower level that the first one. The frequencies have been chosen equal to the first example

```
In [6]: dft_signal_mixture(32, 1, 10.3, 0.1, 15.2)
```



Now the contribution of the second exponential is hidden in the spread spectrum of the first exponential.

1.2 Window Functions

For the discussion of the leakage effect in the previous section, a hard truncation of the signal $x[k]$ by a rectangular window $w[k] = \text{rect}_N[k]$ was assumed. Also other window functions are used for spectral analysis. The resulting properties depend on the spectrum $W(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the window function, since the spectrum of the windowed signal is given as $X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{1}{2\pi} X(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \circledast W(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. Different window functions have different properties. For instance with respect to the capability to distinguish two neighboring signals (frequency resolution) or to detect two signals where one is much weaker (sidelobe level). Since these two aspects counteract for typical window functions, the choice of a suitable window depends heavily on the application. We therefore take a look at frequently applied window functions and their properties.

In order to investigate the windows, a function is defined which computes and plots the spectrum of a given window function. The spectrum $W(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is approximated numerically by the DFT.

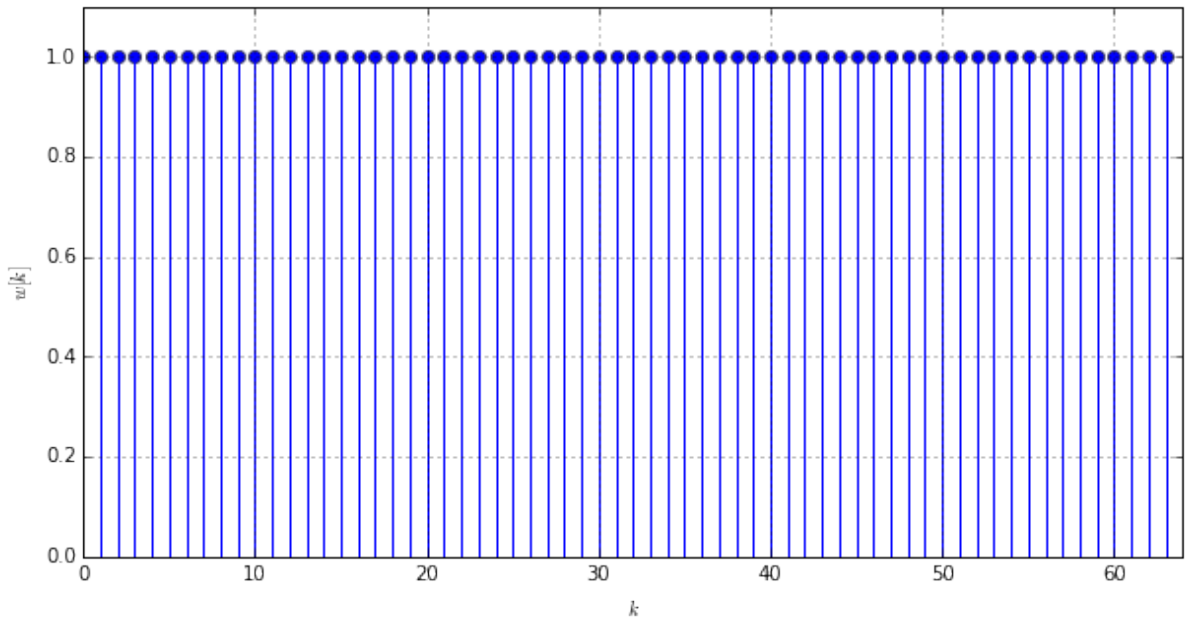
```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig


        def dft_window_function(w):

            N = len(w)
```

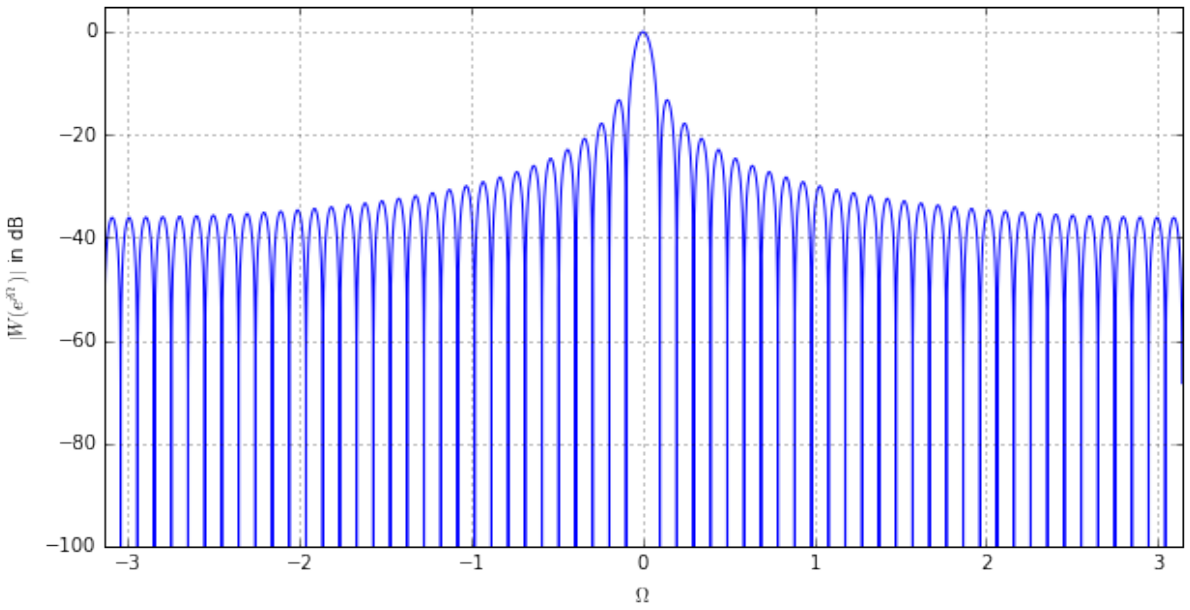
```
# DFT of window function
W = np.fft.fft(w, 8192)
W = np.fft.fftshift(W)
W = W / np.amax(W) + np.nextafter(0,1)
mu = np.linspace(-np.pi, np.pi, 8192)

# plot window function and its spectrum
plt.rcParams['figure.figsize'] = 10, 5
plt.stem(w)
plt.xlabel(r'$k$')
plt.ylabel(r'$w[k]$')
plt.axis([0, N, 0, 1.1])
plt.grid()

plt.figure()
plt.plot(mu, 20*np.log10(np.abs(W)))
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$| W(e^{j \Omega}) |$ in dB')
plt.axis([-np.pi, np.pi, -100, 5])
plt.grid()
```

1.2.1 Rectangular Window

The rectangular window $w[k] = \text{rect}_N[k]$ takes all samples with equal weight into account. The main lobe of its magnitude spectrum is narrow, but the level of the side lobes is rather high. It has the highest frequency selectivity.

```
In [2]: dft_window_function(np.ones(64))
```

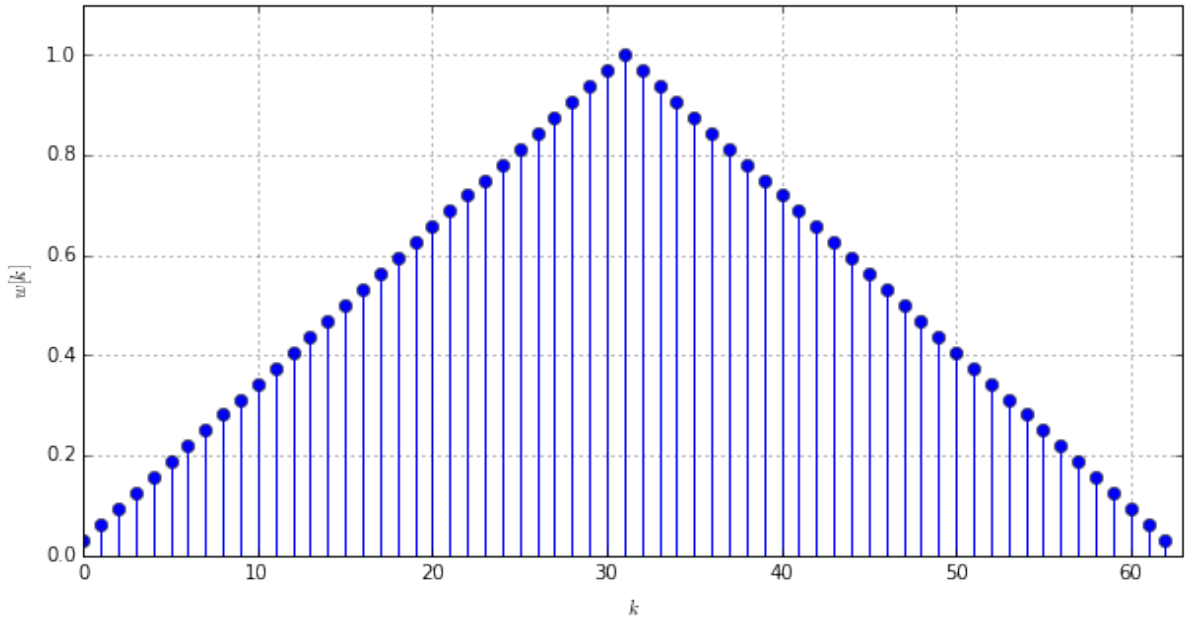


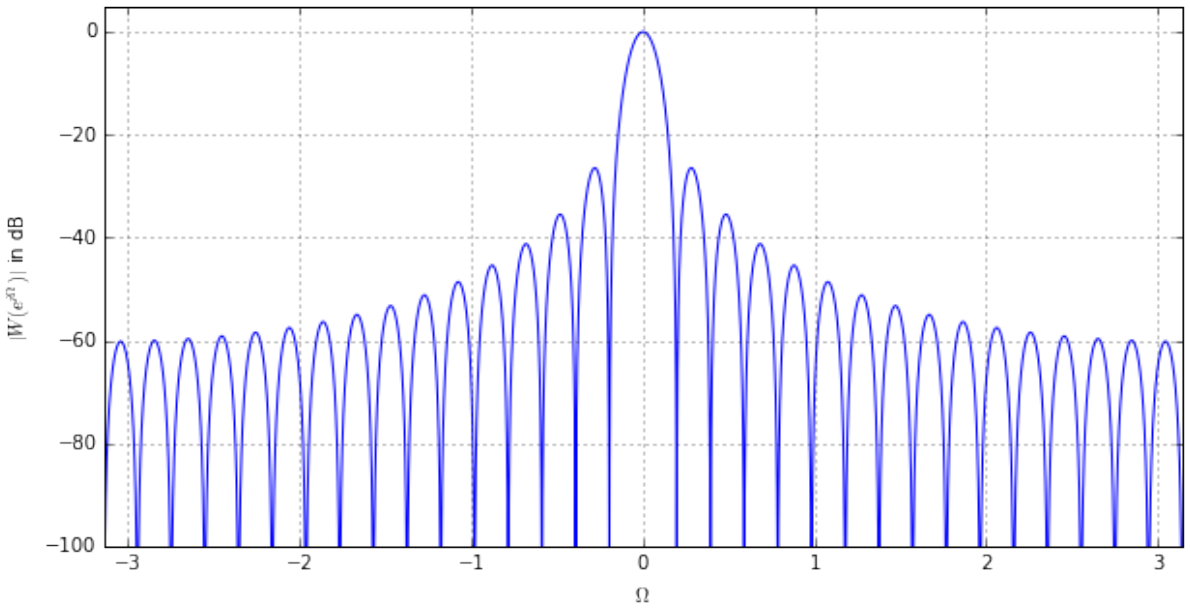


1.2.2 Triangular Window

For an odd window length $2N - 1$, the triangular window can be expressed as the convolution of two rectangular windows $w[k] = \text{rect}_N[k] * \text{rect}_N[k]$. The main lobe is wider as for the rectangular window, but the level of the side lobes decays faster.

```
In [3]: dft_window_function(sig.triang(63))
```

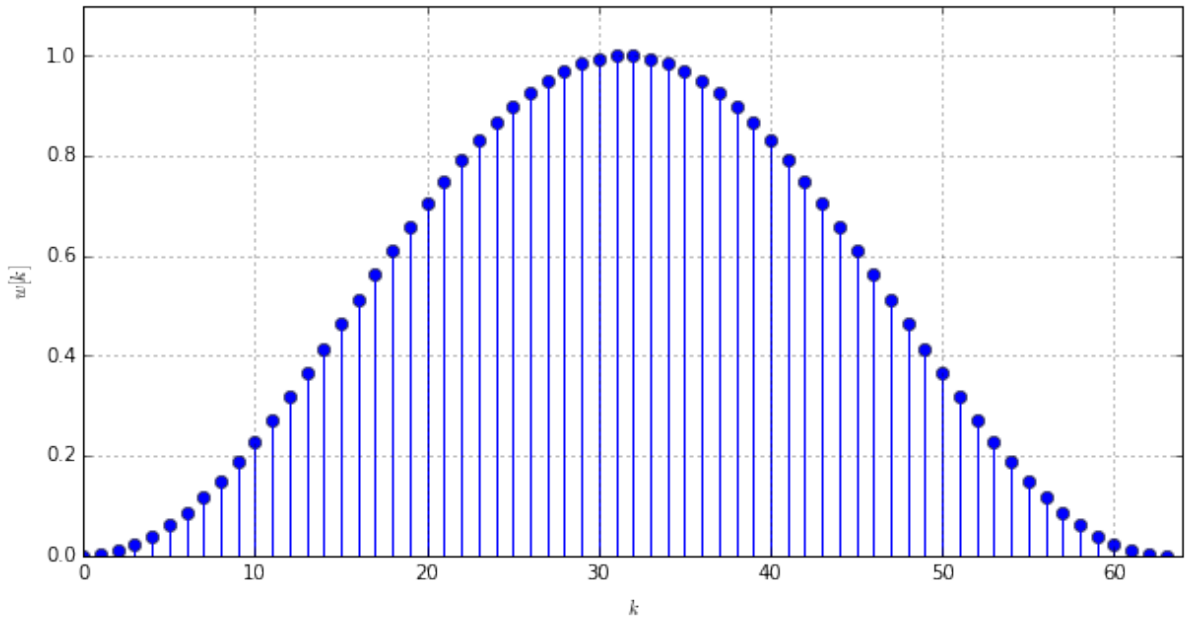


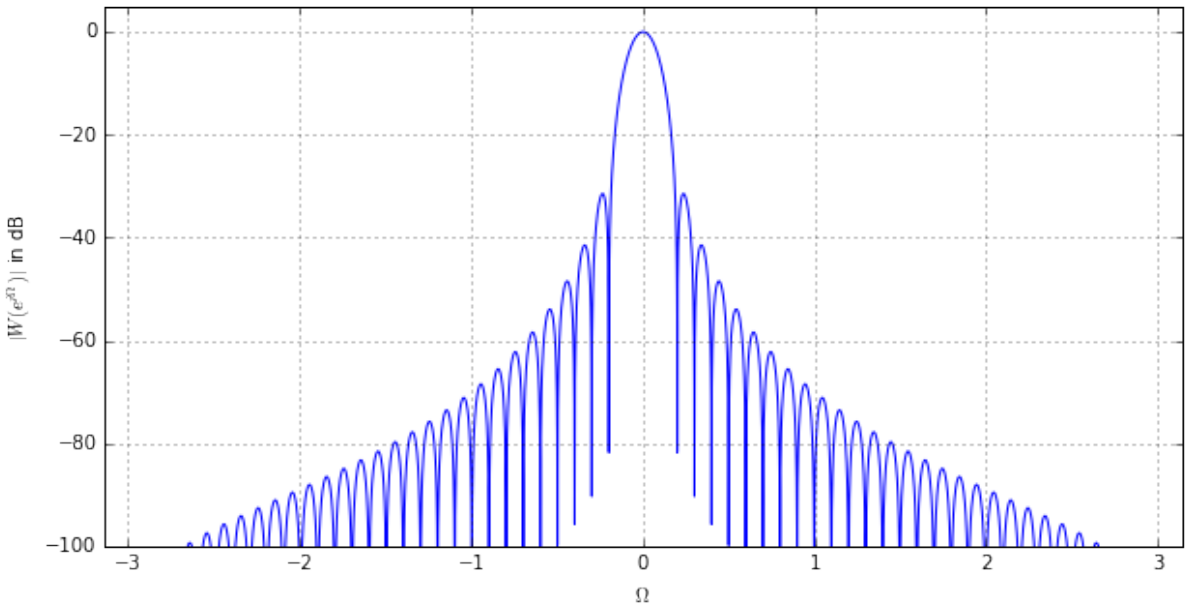


1.2.3 Hanning Window

The Hanning window $w[k] = \frac{1}{2}(1 - \cos[2\pi \frac{k}{N}])$ is a smooth window whose first and last value is zero. It features a fast decay of the side lobes.

```
In [4]: dft_window_function(np.hanning(64))
```

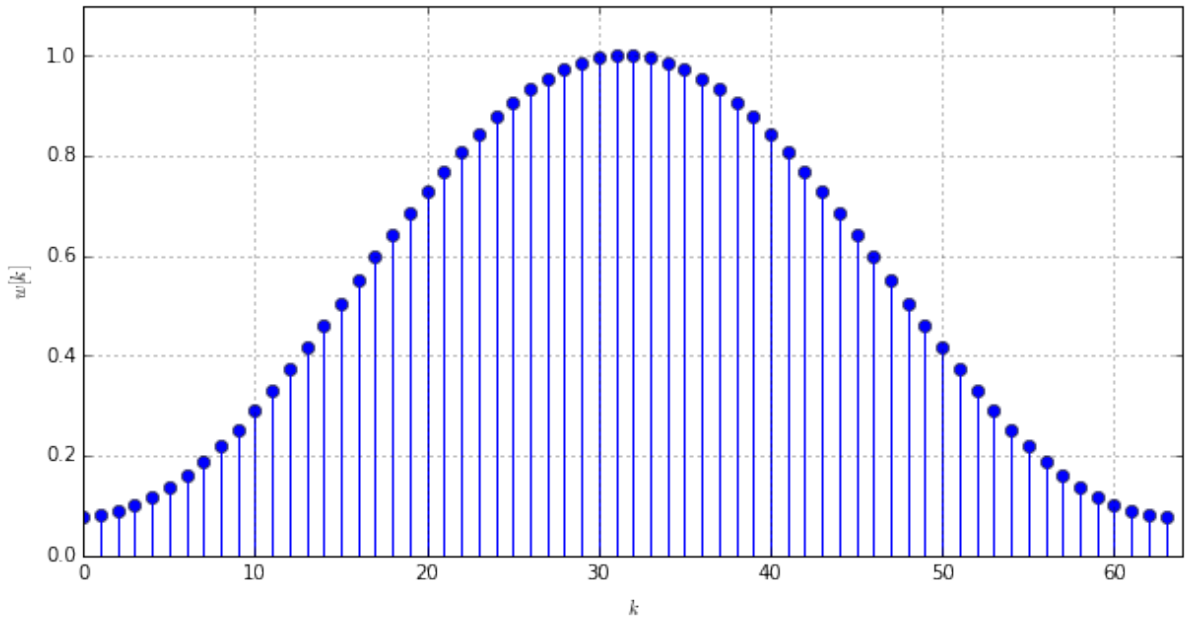


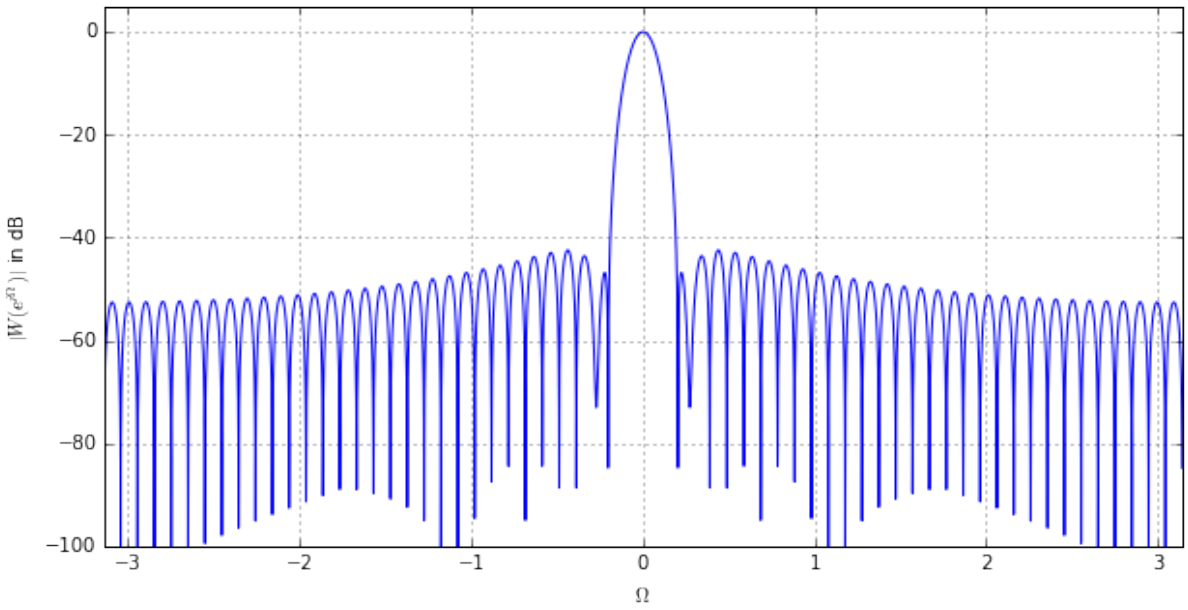


1.2.4 Hamming Window

The Hamming window $w[k] = 0.54 - 0.46\cos[2\pi\frac{k}{N}]$ is a smooth window function whose first and last value is not zero. The level of the side lobes is approximately constant.

```
In [5]: dft_window_function(np.hamming(64))
```

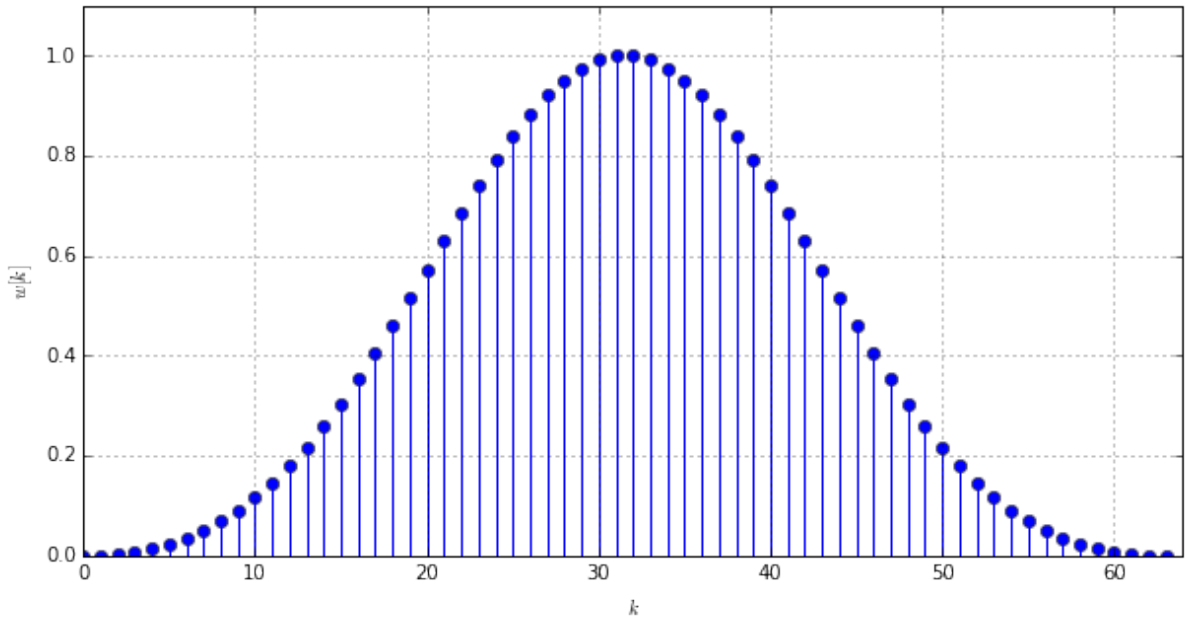


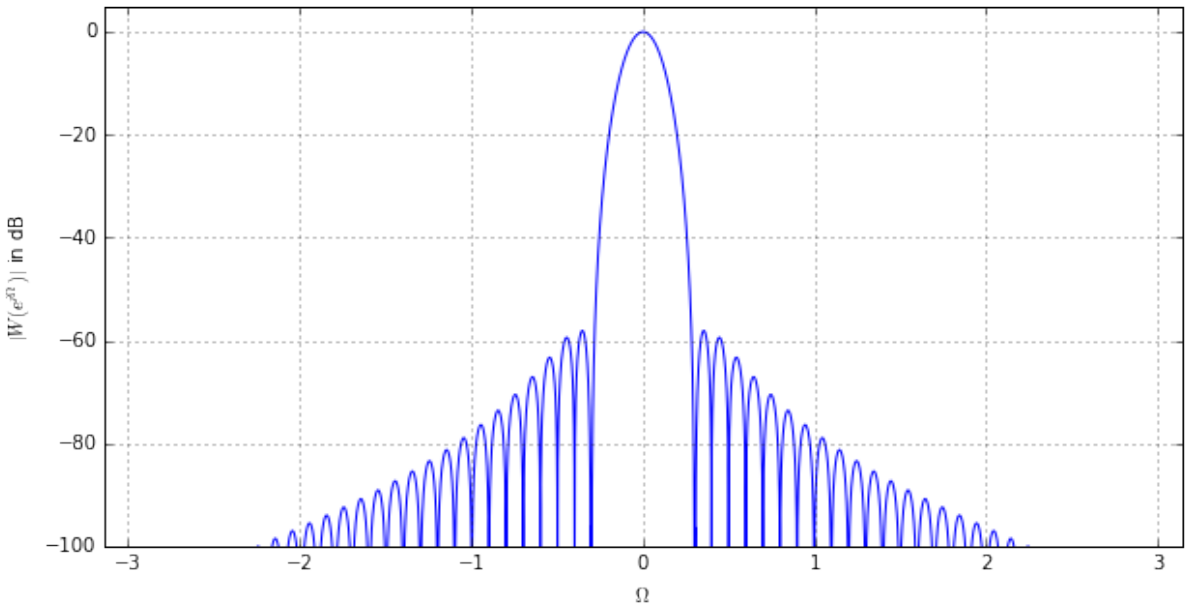


1.2.5 Blackman Window

The Blackman window $w[k] = 0.42 - 0.5\cos[2\pi\frac{k}{N}] + 0.08\cos[4\pi\frac{k}{N}]$ features a rapid decay of side lobes at the cost of a wide main lobe and low frequency selectivity.

```
In [6]: dft_window_function(np.blackman(64))
```





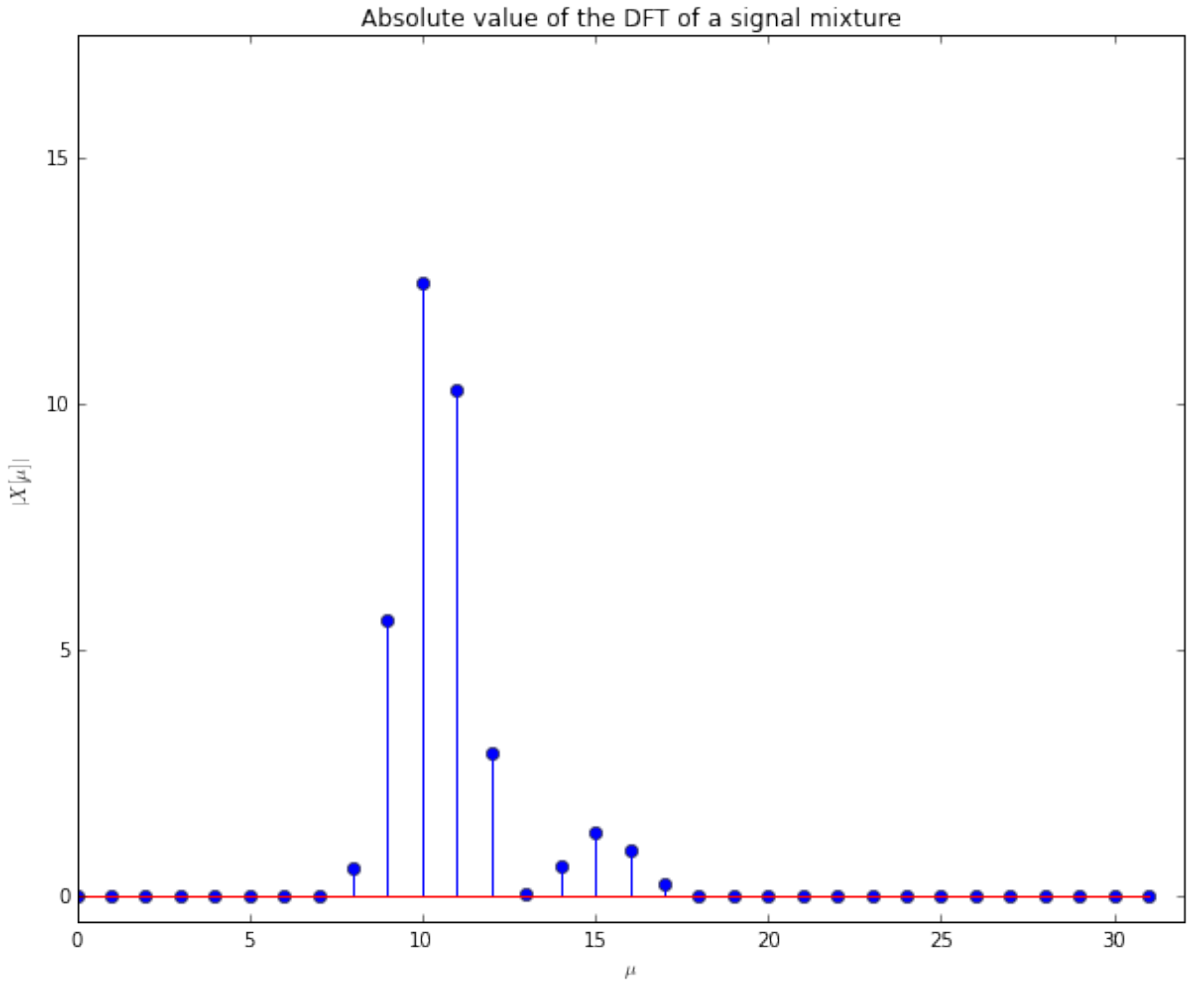
1.2.6 Analysis of Signal Mixtures by the Windowed Discrete Fourier Transformation

The function for the analysis of a *superposition of two exponential signals from the previous section* is extended by windowing

```
In [7]: def dft_signal_mixture_window(N, A1, P1, A2, P2, w):
            # N: length of signal/DFT
            # A1, P1, A2, P2: amplitude and periodicity of 1st/2nd complex exponential
            # window applied to the signal

            # generate the signal mixture
            Om0_1 = P1*(2*np.pi/N)  # frequency of 1st exponential signal
            Om0_2 = P2*(2*np.pi/N)  # frequency of 2nd exponential signal
            k = np.arange(N)
            x = A1 * np.exp(1j*Om0_1*k) + A2 * np.exp(1j*Om0_2*k)
            x = x * w

            # DFT of the signal mixture
            mu = np.arange(N)
            X = np.fft.fft(x)

            # plot spectrum
            plt.figure(figsize = (10, 8))
            plt.stem(mu, abs(X))
            plt.title(r'Absolute value of the DFT of a signal mixture')
            plt.xlabel(r'$\mu$')
            plt.ylabel(r'$|X[\mu]|$')
            plt.axis([0, N, -0.5, abs(X).max()+5]);
```

Now the last example is re-investigated by using a Blackman window which features a high suppression of the sidelobes. The second exponential signal with the lower level now becomes visible in the spectrum.

```
In [8]: dft_signal_mixture_window(32, 1, 10.3, 0.1, 15.2, np.blackman(32))
```



Exercise

- Examine the effect of the other window functions for small/large frequency and level differences. What window function is best suited for what situation?

1.3 Zero-Padding

1.3.1 Concept

Let's assume a signal $x_N[k]$ of finite length N, for instance a windowed signal $x_N[k] = x[k] \cdot \text{rect}_N[k]$. The discrete Fourier transformation (DFT) of $x_N[k]$ reads

$$X_N[\mu] = \sum_{k=0}^{N-1} x_N[k] \; w_N^{\mu k}$$

where $w_N = \mathrm{e}^{-\mathrm{j}\frac{2\pi}{N}}$ denotes the kernel of the DFT. For a sampled time-domain signal, the distance in frequency between two neighboring coefficients is given as $\Delta f = \frac{f_s}{N}$, where $f_s = \frac{1}{T}$ denotes the sampling frequency. Hence, if N is increased the distance between neighboring frequencies is decreased. This leads to the concept of zero-padding in spectral analysis. Here the signal $x_N[k]$ of finite length is filled up with (M-N) zero values to a total length $M \geq N$

$$x_M[k] = [x[0], x[1], \ldots, x[N-1], 0, \ldots, 0]^{\mathrm{T}}$$

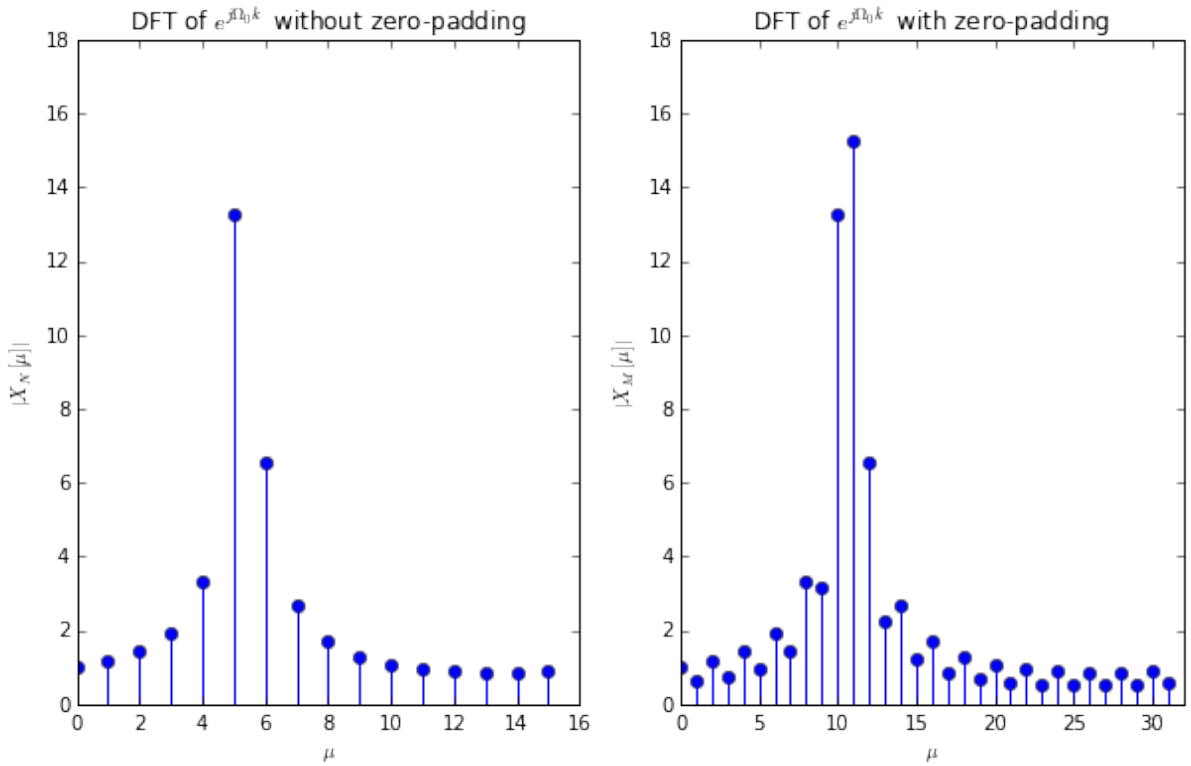
The DFT $X_M[\mu]$ of $x_M[k]$ has now a decreased distance between neighboring frequencies $\Delta f = \frac{f_s}{M}$.

The question arises what influence zero-padding has on the spectrum and if it can enhance spectral analysis. On first sight it seems that the frequency resolution is higher, however do we get more information for the signal? In order to discuss this, a short numerical example is shown followed by a derivation of the relations between the spectrum $X_M[k]$ with zero-padding and $X_N[k]$ without zero-padding.

Example

The following example computes and plots the magnitude spectra of a truncated complex exponential signal $x_N[k] = \mathrm{e}^{\,\mathrm{j}\,\Omega_0\,k} \cdot \mathrm{rect}_N[k]$ and its zero-padded version $x_M[k]$.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        N = 16  # length of the signal
        M = 32  # length of zero-padded signal
        Om0 = 5.33*(2*np.pi/N)  # frequency of exponential signal
        #chosen to be not exactly an eigenfrequency mu*2*pi/N of the DFT


        # DFT of the exponential signal
        xN = np.exp(1j*Om0*np.arange(N))
        XN = np.fft.fft(xN)
        # DFT of the zero-padded exponential signal
        xM = np.concatenate((xN, np.zeros(M-N)))
        XM = np.fft.fft(xM)


        # plot spectra
        plt.figure(figsize = (10, 6))

        plt.subplot(121)
        plt.stem(np.arange(N),np.abs(XN))
        plt.title(r'DFT of $e^{j \Omega_0 k}$ without zero-padding')
        plt.xlabel(r'$\mu$')
        plt.ylabel(r'$|X_N[\mu]|$')
        plt.axis([0, N, 0, 18])

        plt.subplot(122)
        plt.stem(np.arange(M),np.abs(XM))
        plt.title(r'DFT of $e^{j \Omega_0 k}$ with zero-padding')
        plt.xlabel(r'$\mu$')
        plt.ylabel(r'$|X_M[\mu]|$')
        plt.axis([0, M, 0, 18]);
```



Exercise

- Check the two spectra carefully for relations. Are there common coefficients for the case $M = 2N$?
- Increase the length `M` of the zero-padded signal $x_M[k]$. Can you get additional information from the spectrum?

1.3.2 Interpolation of the Discrete Fourier Transformation

Lets step back to the discrete time Fourier transformation (DTFT) of the finite-length signal $x_N[k]$ without zero-padding

$$X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{k=-\infty}^{\infty} x_N[k]\,\mathrm{e}^{\,-\mathrm{j}\,\Omega\,k} = \sum_{k=0}^{N-1} x_N[k]\,\mathrm{e}^{\,-\mathrm{j}\,\Omega\,k}$$

The discrete Fourier transformation (DFT) is derived by sampling $X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ at $\Omega = \mu\frac{2\pi}{N}$

$$X_N[\mu] = X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega})\big|_{\Omega=\mu\frac{2\pi}{N}} = \sum_{k=0}^{N-1} x_N[k]\,\mathrm{e}^{\,-\mathrm{j}\,\mu\frac{2\pi}{N}\,k}$$

Since the DFT coefficients $X_N[\mu]$ are sampled equidistantly (or rather equiangularly on the unit circle) from the DTFT $X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega})$, we can reconstruct the DTFT of $x_N[k]$ from the DFT coefficients by interpolation. Introduce the inverse DFT of $X_N[\mu]$

$$x_N[k] = \frac{1}{N}\sum_{\mu=0}^{N-1} X_N[\mu]\,\mathrm{e}^{\,\mathrm{j}\,\frac{2\pi}{N}\mu\,k}$$

into the discrete time Fourier transformation (DTFT)

$$X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{k=0}^{N-1} x_N[k]\,\mathrm{e}^{\,-\mathrm{j}\,\Omega\,k} = \sum_{\mu=0}^{N-1} X_N[\mu]\cdot\frac{1}{N}\sum_{k=0}^{N-1}\mathrm{e}^{-\mathrm{j}\,k\,(\Omega-\frac{2\pi}{N}\mu)}$$

reveals the relation between $X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and $X_N[\mu]$. The last sum can be solved analytically yielding the so called periodic sinc function (aliased sinc function, Dirichlet kernel) $\mathrm{psinc}_N(\Omega)$ and a phase shift. This results in

$$X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{\mu=0}^{N-1} X_N[\mu] \cdot \mathrm{e}^{-\,\mathrm{j}\,\frac{(\Omega - \frac{2\pi}{N}\mu)(N-1)}{2}} \cdot \mathrm{psinc}_N(\Omega - \frac{2\pi}{N}\mu)$$

where

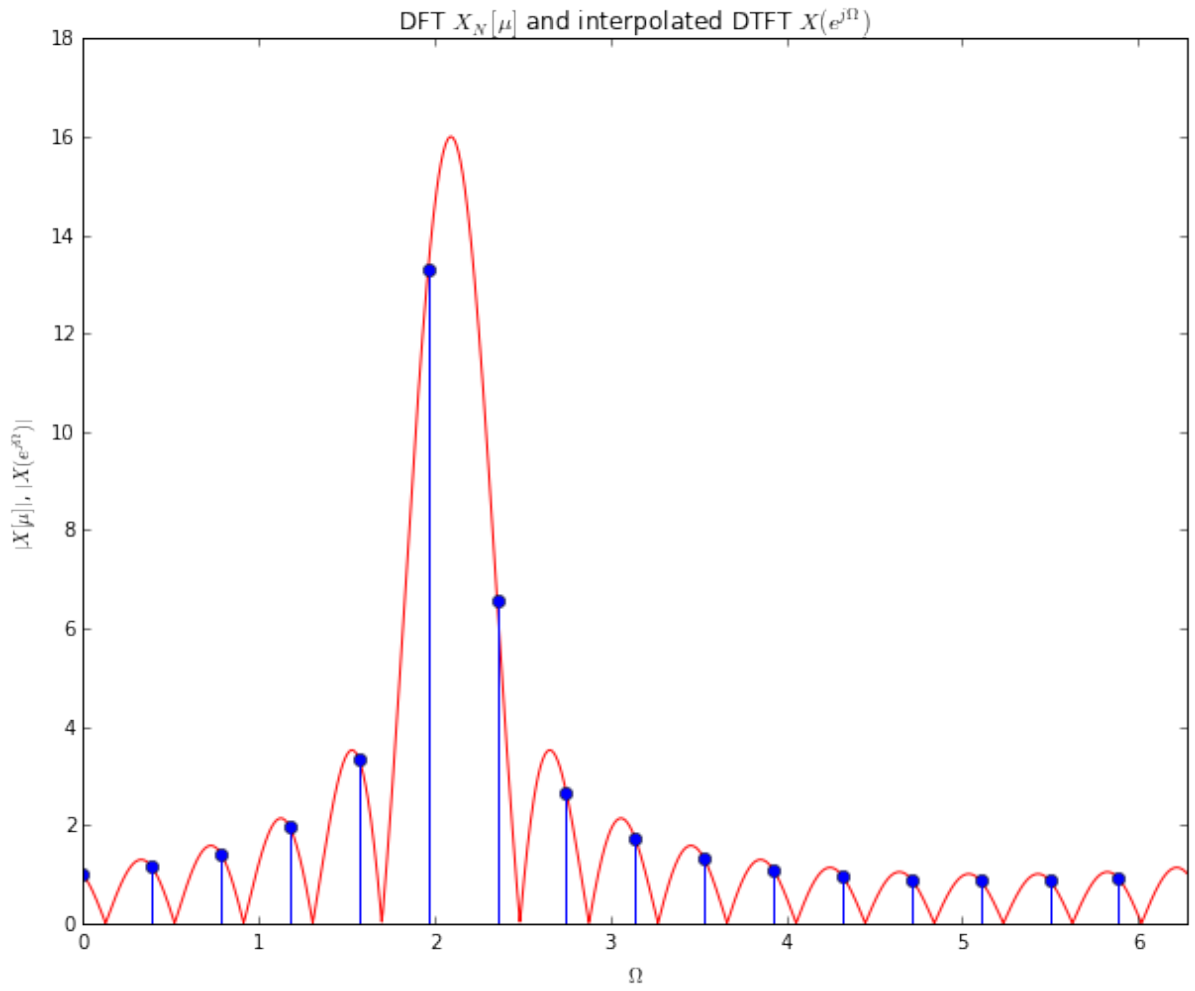
$$\mathrm{psinc}_N(\Omega) = \frac{1}{N} \frac{\sin(\frac{N}{2}\Omega)}{\sin(\frac{1}{2}\Omega)}$$

denotes the N-th order periodic sinc function.

Example

This example illustrates the interpolation of $X_N[\mu]$ using the relation derived above. Using above definition, the periodic sinc function is not defined at $\Omega = 2\pi\, n$ for $n \in \mathbb{Z}$. This is resolved by taking its limit value.

```
In [2]: N = 16  # length of the signal
        M = 1024  # number of frequency points for DTFT
        Om0 = 5.33*(2*np.pi/N)  # frequency of exponential signal


        # periodic sinc function
        def psinc(x, N):
            x = np.asanyarray(x)
            y = np.where(x == 0, 1.0e-20, x)
            return 1/N * np.sin(N/2*y)/np.sin(1/2*y)

        # DFT of the exponential signal
        xN = np.exp(1j*Om0*np.arange(N))
        XN = np.fft.fft(xN)

        # interpolation of DFT coefficients
        Xi = np.asarray(np.zeros(M), dtype=complex)
        for mu in np.arange(M):
            Omd = 2*np.pi/M*mu-2*np.pi*np.arange(N)/N
            interpolator = psinc(Omd, N) * np.exp(-1j*Omd*(N-1)/2)
            Xi[mu] = np.sum(XN * interpolator)

        # plot spectra
        plt.figure(figsize = (10, 8))
        plt.hold(True)

        plt.plot(np.arange(M)*2*np.pi/M, abs(Xi), 'r')
        plt.stem(np.arange(N)*2*np.pi/N, abs(XN))
        plt.title(r'DFT $X_N[\mu]$ and interpolated DTFT $X(e^{j \Omega})$')
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$|X[\mu]|$, $|X(e^{j \Omega})|$')
        plt.axis([0, 2*np.pi, 0, 18]);
```



1.3.3 Relation between Discrete Fourier Transformations with and without Zero-Padding

It was already outlined above that the DFT is related to the DTFT by sampling. Since the zero-paded signal $x_M[k]$ differs to $x_N[k]$ only with respect to the additional zeros, its DFT $X_M[\mu]$ is given by resampling the DTFT interpolation of $X_N[\eta]$, i.e. $X_N(\mathrm{e}^{\mathrm{j}\,\Omega})$ at $\Omega = \frac{2\pi}{M}\mu$

$$X_M[\mu] = \sum_{\eta=0}^{N-1} X_N[\eta] \cdot \mathrm{e}^{-\mathrm{j}\,\frac{(\frac{2\pi}{M}\mu - \frac{2\pi}{N}\eta)(N-1)}{2}} \cdot \mathrm{psinc}_N\left(\frac{2\pi}{M}\mu - \frac{2\pi}{N}\eta\right)$$

for $\mu = 0, 1, \ldots, M-1$.

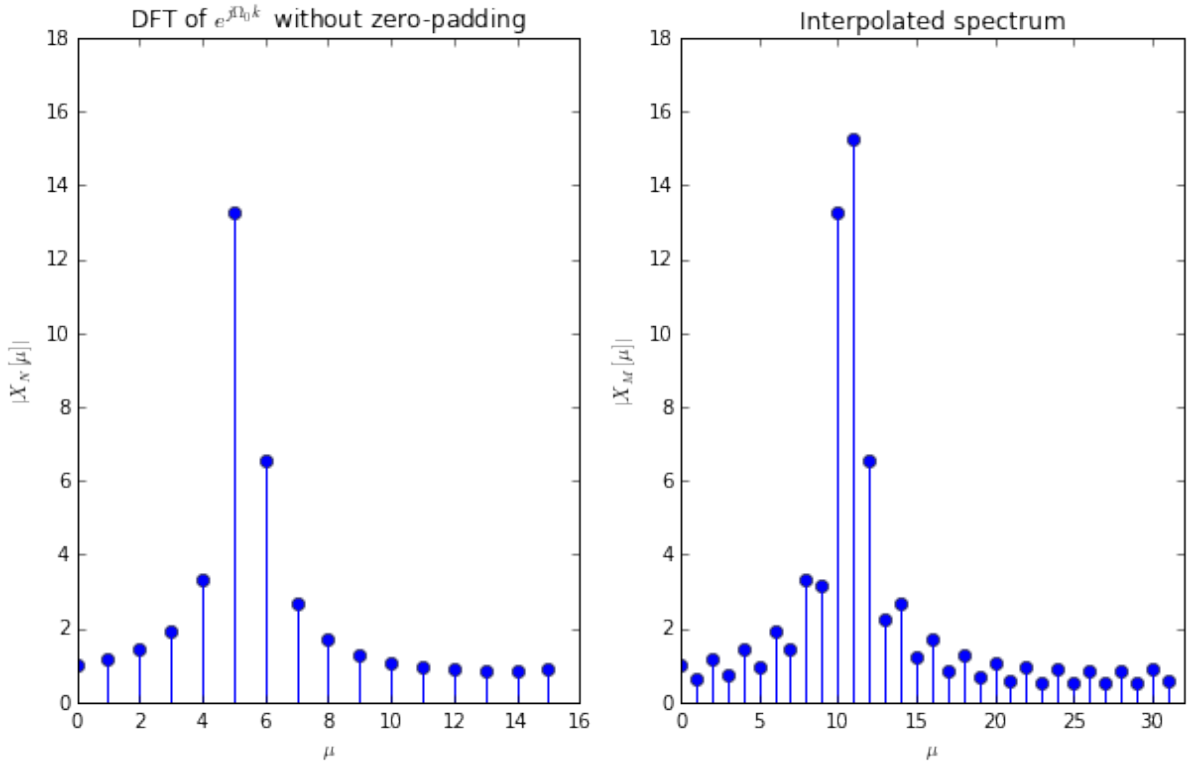
Above equation relates the spectrum $X_N[\mu]$ of the original signal $x_N[k]$ to the spectrum $X_M[\mu]$ of the zero-padded signal $x_M[k]$. It essentially constitutes a bandlimited interpolation of the coefficients $X_N[\mu]$.

All spectral information of a signal of finite length N is already contained in its spectrum derived from a DFT of length N. By applying zero-padding and a longer DFT, the frequency resolution is only virtually increased. The additional coefficients are related by bandlimited interpolation to the original ones. In general, zero-padding does not bring a benefit in spectral analysis. It may bring a benefit in special applications, for instance when estimating the frequency of an isolated harmonic signal from its spectrum.

Example

The following example shows that the coefficients $X_M[\mu]$ of the spectrum after zero-padding can be derived from the spectrum $X_N[\eta]$ by interpolation.

In [3]:
```
N = 16  # length of the signal
M = 32  # number of points for interpolated DFT
Om0 = 5.33*(2*np.pi/N)  # frequency of exponential signal


# periodic sinc function
def psinc(x, N):
    x = np.asanyarray(x)
    y = np.where(x == 0, 1.0e-20, x)
    return 1/N * np.sin(N/2*y)/np.sin(1/2*y)

# DFT of the exponential signal
xN = np.exp(1j*Om0*np.arange(N))
XN = np.fft.fft(xN)

# interpolation of DFT coefficients
XM = np.asarray(np.zeros(M), dtype=complex)
for mu in np.arange(M):
    Omd = 2*np.pi/M*mu-2*np.pi*np.arange(N)/N
    interpolator = psinc(Omd, N) * np.exp(-1j*Omd*(N-1)/2)
    XM[mu] = np.sum(XN * interpolator)

# plot spectra
plt.figure(figsize = (10, 6))

plt.subplot(121)
plt.stem(np.arange(N),np.abs(XN))
plt.title(r'DFT of $e^{j \Omega_0 k}$ without zero-padding')
plt.xlabel(r'$\mu$')
plt.ylabel(r'$|X_N[\mu]|$')
plt.axis([0, N, 0, 18])

plt.subplot(122)
plt.stem(np.arange(M),np.abs(XM))
plt.title(r'Interpolated spectrum')
plt.xlabel(r'$\mu$')
plt.ylabel(r'$|X_M[\mu]|$')
plt.axis([0, M, 0, 18]);
```



Exercise

- Compare the interpolated spectrum to the spectrum with zero padding from the first example.
- Estimate the frequency Ω_0 of the exponential signal from the interpolated spectrum. How could you increase the accuracy of your estimate?

1.4 Short-Time Fourier Transformation

The DFT is not very well suited for the analysis of instationary signals when applied to the entire signal. Practical signals, for instance an antenna signal, cannot be analyzed in an on-line manner by the DFT. This motivates to split a long signal into segments and compute the DFT on these segments. This transformation is known as the short-time Fourier transformation (STFT).

The STFT $X[\mu, n]$ of a signal $x[k]$ is defined as

$$X[\mu, n] = \sum_{k=n}^{n+N-1} x[k]\, w[k-n]\, w_N^{k\mu}$$

where $w_N = \mathrm{e}^{-\mathrm{j}\,\frac{2\pi}{N}}$ denotes the kernel of the DFT and $w[k]$ a window function of length N which is normalized by $\sum_{k=0}^{N-1} w[k] = 1$. Starting from $k = n$, the signal $x[k]$ is windowed by $w[k]$ to a segment of length N. This windowed segment is then transformed by a DFT of length N.

The STFT has many applications in digital signal processing. For instance in the spectral analysis of signals or the processing of instationary signals. The resulting spectrum $X[\mu, n]$ depends on the frequency index μ and the time index n. It is therefore also termed as time-frequency domain and techniques using the STFT as time-frequency processing.

The properties of the STFT depend on

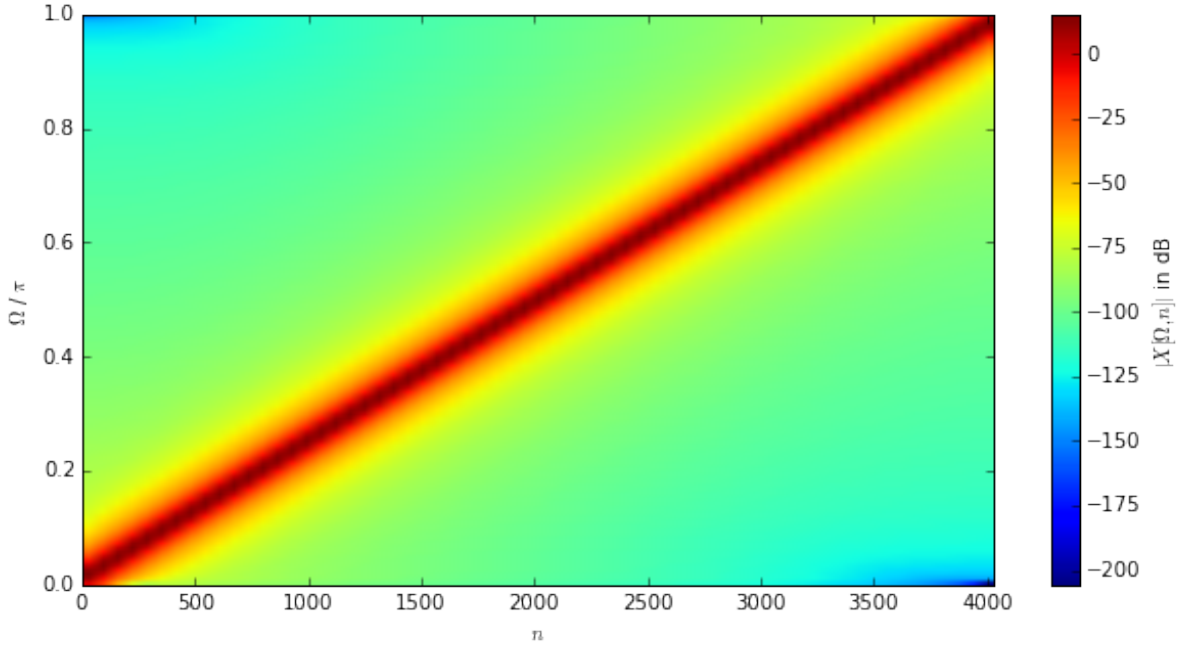
- the length N of the segments,
- the overlap between the segments, and
- the window function $w[k]$.

The size N of the segments and the window function influence the spectral and temporal resolution of the STFT. The time index n of the STFT can be increased by an arbitrary step size. The step size determines the overlap between two consecutive STFTs. For instance the spectra $X[\mu, n]$ and $X[\mu, n+1]$ have $N-1$ overlapping samples. The overlap is sometimes given as percentage of the segment length N.

1.5 The Spectrogram

The magnitude $|X[\mu, n]|$ is known as the spectrogram of a signal. It is frequently used to analyze signals in the time-frequency domain. For instance by a spectrum analyzer. The following example computes the spectrogram of an unknown signal

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig


        N = 8192  # length of the signal
        L = 256  # length of one segment
        overlap = 128  # overlap between segments


        # generate signal
        k = np.arange(N)
        x = sig.chirp(k, 0, N, .5)

        # compute and plot spectrogram
        plt.figure(figsize = (10, 5))
        plt.specgram(x, NFFT=L, noverlap=overlap, sides='onesided')
        plt.xlabel(r'$n$')
        plt.ylabel(r'$\Omega$ / $\pi$')
        cb = plt.colorbar()
        cb.set_label(r'$|X[\Omega,n]|$ in dB')
        plt.autoscale(tight=True)
```



Exercise

- Can you characterize the signal from its spectrogram? How would it sound like?
- Change the segment length `L` and the overlap `overlap` between segments. Rerun the cell. What changes in the spectrogram?

CHAPTER 2

Random Signals

2.1 Introduction

Random signals are signals whose values are not (or only to a limited extend) predictable. Frequently used alternative terms are

- stochastic signals
- non-deterministic signals

Random signals play an important role in various fields of signal processing and communications. This is due to the fact that only random signals carry information. A signal which is observed by some receiver has to have unknown contributions in order to represent information.

Random signals are often classified as useful/desired and disturbing/interfering signals. For instance

- useful signals: data, speech, music, ...
- disturbing signals: thermal noise at a resistor, amplifier noise, quantization noise, ...

Practical signals are frequently modeled as a combination of a useful signal and an additive noise component.

As the values of a random signal cannot be foreseen, the properties of random signals are described by the their statistical characteristics. For instance by average values.

2.1.1 Statistical Signal Processing

Statistical signal processing treats signals as random processes, in contrary to the assumption of deterministic signals in traditional signal processing. Two prominent application examples involving random signals are

Measurement of physical quantities

The measurement of physical quantities is often subject to additive noise and distortions by e.g. the amplifier. The aim of statistical signal processing is to estimate the physical quantity from the observed sensor data.

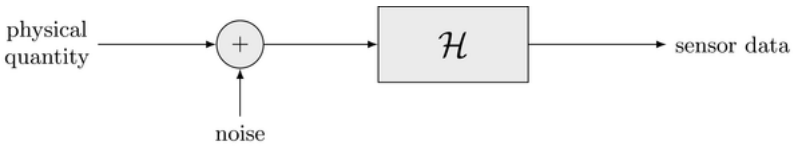


Fig. 2.1: Illustration: Measurement of pyhisical quantities

$\mathcal{H}$ denotes an arbitrary (not necessarily LTI) system.

Communication channel

In communications a message is send over a channel distorting the signal by e.g. multipath propagation. Additive noise is present at the receiver due to background and amplifier noise. The aim of statistical signal processing is to estimate the message sent from the received message.

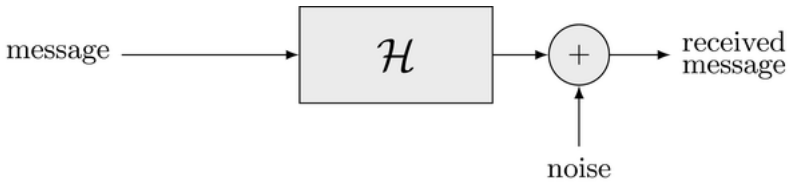


Fig. 2.2: Illustration: Communication channel

2.1.2 Random Processes

A random process is a stochastic process which generates an ensemble of random signals. A random process

- provides a mathematical model for an ensemble of random signals
- generates different sample functions with specific common properties

Its is important to differentiate between

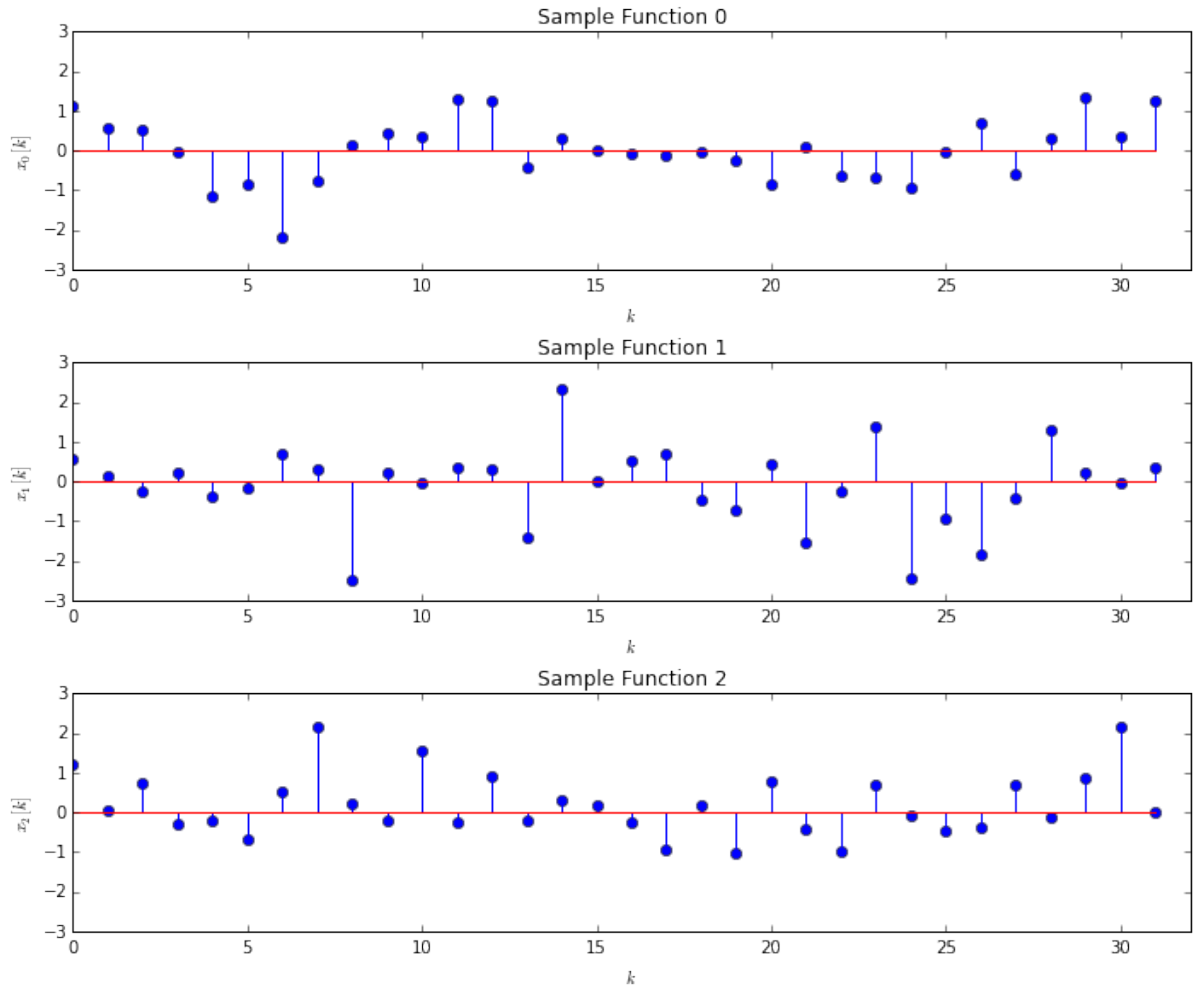
- *ensemble*: collection of all possible signals of a random process
- *sample function*: one specific random signal

An example for a random process is speech produced by humans. Here the ensemble is composed from the speech signals produced by all humans on earth, one particular speech signal produced by one person at a specific time is a sample function.

Sample functions of a random process

The following example shows sample functions of a continuous amplitude real-valued random process. All sample functions have the same characteristics with respect to certain statistical properties.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        # draw 3 sample functions from a random process
        x = np.random.normal(size=(3, 32))

        # plot sample functions
        fig = plt.figure(figsize=(10, 8))
        for n in range(3):
            plt.subplot(3,1,n+1)
            plt.tight_layout()
            plt.stem(x[n,:])
            plt.title('Sample Function %d' %n)
            plt.xlabel(r'$k$')
            plt.ylabel(r'$x_%d[k]$' %n)
            plt.axis([0, 32, -3, 3])
```



Exercise

- What is different, what is common between the sample functions?
- Rerun the cell. What changes now?

2.1.3 Properties of Random Processes and Random Signals

Random signals can be characterized by their

- amplitude distributions and
- ensemble averages/moments.

Both measures will be introduced in the following section.

2.2 Cumulative Distribution Functions

A random process can be characterized by the statistical properties of its amplitude values. Cumulative distribution functions (CDFs) are one possibility.

2.2.1 Univariate Cumulative Distribution Function

The univariate CDF $P_x(\theta, k)$ of a continuous-amplitude real-valued random signal $x[k]$ is defined as

$$P_x(\theta, k) := \Pr\{x[k] \leq \theta\}$$

where $\Pr\{\cdot\}$ denotes the probability that the given condition holds. The univariate CDF quantifies the probability that for a fixed time index k the condition $x[k] \leq \theta$ holds for the entire ensemble. It has the following properties which can be concluded directly from its definition

$$\lim_{\theta \to -\infty} P_x(\theta, k) = 0$$

and

$$\lim_{\theta \to \infty} P_x(\theta, k) = 1$$

The probability that $\theta_1 < x[k] \leq \theta_2$ is given as

$$\Pr\{\theta_1 < x[k] \leq \theta_2\} = P_x(\theta_2, k) - P_x(\theta_1, k) \tag{2.1}$$

Hence, the probability that a continuous-amplitude random signal takes a specific value $x[k] = \theta$ is not defined by the CDF. This motivates the use of the probability density function introduced later.

2.2.2 Bivariate Cumulative Distribution Function

The bivariate or joint CDF $P_{xy}(\theta_x, \theta_y, k_x, k_y)$ of two continuous-amplitude real-valued random signals $x[k]$ and $y[k]$ is defined as

$$P_{xy}(\theta_x, \theta_y, k_x, k_y) := \Pr\{x[k_x] \leq \theta_x \wedge y[k_y] \leq \theta_y\}$$

The joint CDF quantifies the probability that for a fixed k_x the condition $x[k_x] \leq \theta_x$ and for a fixed k_y the condition $y[k_y] \leq \theta_y$ holds for the entire ensemble of sample functions.

2.3 Probability Density Functions

Propability density functions (PDFs) describe the probability for a random signal to take on a given value.

2.3.1 Univariate Probability Density Function

The univariate PDF $p_x(\theta, k)$ of a continuous-amplitude real-valued random signal $x[k]$ is defined as the derivative of the univariate CDF

$$p_x(\theta, k) = \frac{\partial}{\partial \theta} P_x(\theta, k)$$

Due to the properties of the CDF and the definition of the PDF, it has the following properties

$$p_x(\theta, k) \geq 0$$

and

$$\int_{-\infty}^{\infty} p_x(\theta, k) \, \mathrm{d}\theta = P_x(\infty, k) = 1$$

The CDF can be derived from the PDF by integration

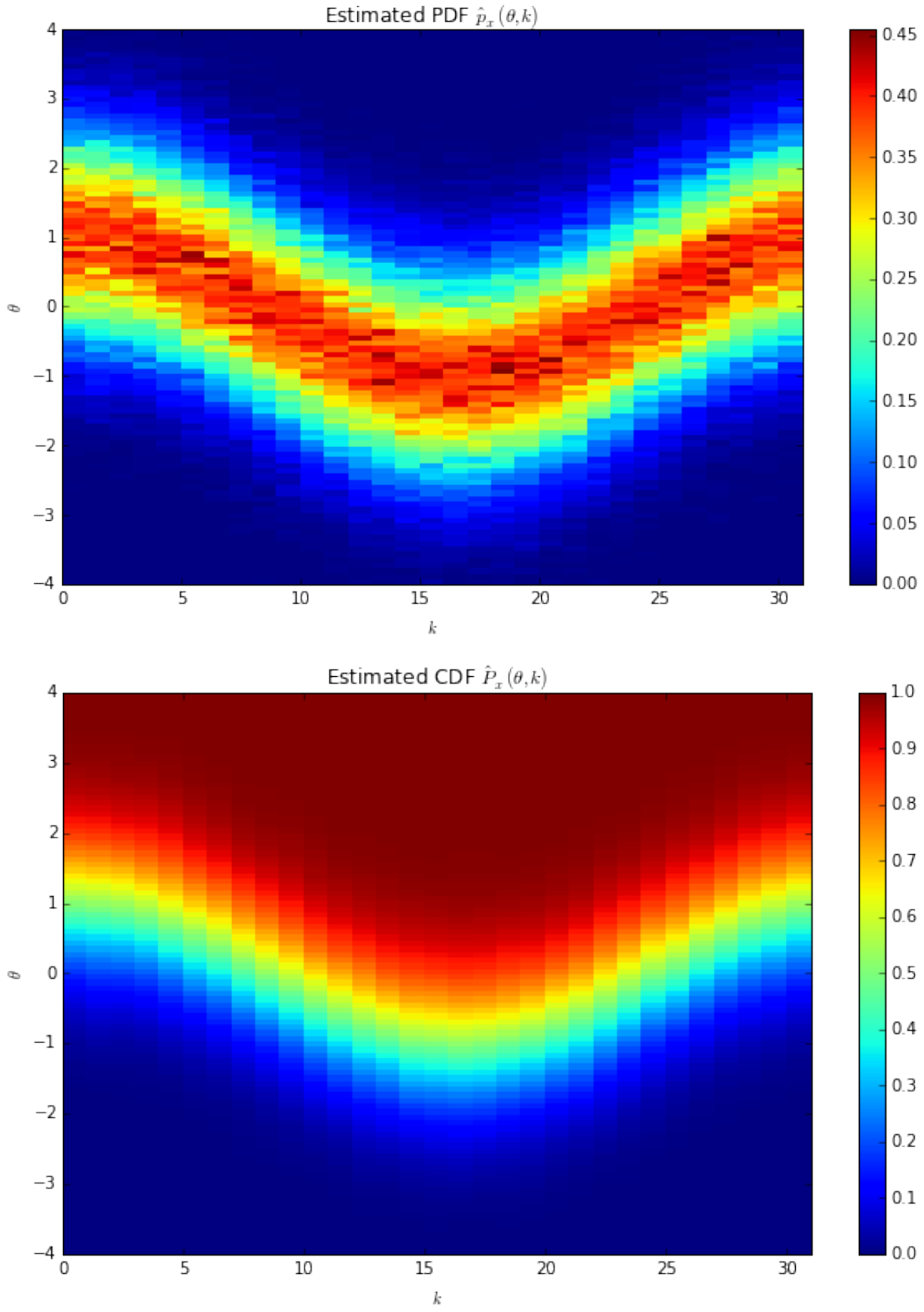
$$P_x(\theta, k) = \int_{-\infty}^{\theta} p_x(\theta, k) \, \mathrm{d}\theta$$

Estimate of an univariate PDF/CDF by the histogram

In the process of calculating a histogram, the entire range of amplitude values of a random signal is split into a series of intervals (bins). Then it is counted how many values of the signal fall into these intervals. This constitutes a numerical approximation of the PDF.

In the following example the histogram of an ensemble is calculated for each time index k. The CDF is approximated by the cumulative sum over the histogram bins.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        K = 32  # number of temporal samples
        N = 10000  # number of sample functions
        bins = 100  # number of bins for the histogram


        # draw sample functions from a random process
        x = np.random.normal(size=(N, K))
        x += np.tile(np.cos(2*np.pi/K*np.arange(K)), [N, 1])

        # compute the histogram
        px = np.zeros((bins, K))
        for n in range(K):
            px[:, n], edges = np.histogram(x[:, n], bins=bins, range=(-4,4), density=Tru

        # compute the CDF
        Px = np.cumsum(px, axis=0) * 8/bins

        # plot the PDF
        plt.figure(figsize=(10,6))
        plt.pcolor(np.arange(K), edges, px)
        plt.title(r'Estimated PDF $\hat{p}_x(\theta, k)$')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$\theta$')
        plt.colorbar()
        plt.autoscale(tight=True)

        # plot the CDF
        plt.figure(figsize=(10,6))
        plt.pcolor(np.arange(K), edges, Px, vmin=0, vmax=1)
        plt.title(r'Estimated CDF $\hat{P}_x(\theta, k)$')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$\theta$')
        plt.colorbar()
        plt.autoscale(tight=True)
```



Exercise

- Change the parameters `N` and `bins` and rerun the cell. What changes? Why?

In numerical simulations of random processes only a finite number of sample functions and temporal samples can

be considered. This holds also for the number of intervals (bins) used for the histogram. As a result, numerical approximations of the CDF/PDF will be subject to statistical uncertainties that typically will become smaller if the number of sample functions is increased.

2.3.2 Bivariate Probability Density Function

The bivariate or joint PDF $p_{xy}(\theta_x, \theta_y, k_x, k_y)$ of two continuous-amplitude real-valued random signals $x[k]$ and $y[k]$ is defined as

$$p_{xy}(\theta_x, \theta_y, k_x, k_y) := \frac{\partial^2}{\partial\theta_x \partial\theta_y} P_{xy}(\theta_x, \theta_y, k_x, k_y)$$

The joint PDF quantifies the joint probability that $x[k]$ takes the value θ_x and that $y[k]$ takes the value θ_y for the entire ensemble of sample functions.

If $x[k] = y[k]$ the bivariate PDF $p_{xx}(\theta_1, \theta_2, k_1, k_2)$ describes the probability that a random signal takes the value θ_1 at time instance k_1 and the value θ_2 at time instance k_2. Hence, $p_{xx}(\theta_1, \theta_2, k_1, k_2)$ provides insights into the temporal dependencies of a random signal $x[k]$.

2.4 Ensemble Averages

Ensemble averages characterize the average properties of a sample function across the population of all possible sample functions of the ensemble. We distinguish between first and higher-order ensemble averages. The former considers the properties of the sample functions for one particular time instant k, while the latter take different signals at different time instants into account.

2.4.1 First Order Ensemble Averages

Definition

The first order ensemble average of a continuous-amplitude real-valued random signal $x[k]$ is defined as

$$E\{f(x[k])\} = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} f(x_n[k])$$

where $E\{\cdot\}$ denotes the expectation operator, $x_n[k]$ the n-th sample function and $f(\cdot)$ an arbitrary mapping function. It is evident from the definition that the ensemble average can only be given exactly for random processes where the internal structure is known. For practical random processes, like e.g. speech, the ensemble average can only be approximated by a finite but sufficient large number N of sample functions.

The ensemble average can also be given in terms of the univariate probability density function (PDF)

$$E\{f(x[k])\} = \int_{-\infty}^{\infty} f(\theta)\, p_x(\theta, k)\, \mathrm{d}\theta$$

Properties

1. The ensemble averages for two different time instants k_1 and k_2 differ in general
$$E\{f(x[k_1])\} \neq E\{f(x[k_2])\}$$
2. For a linear mapping $f(x[k]) = x[k]$, the ensemble average is a linear operation
$$E\{Ax[k] + By[k]\} = A \cdot E\{x[k]\} + B \cdot E\{y[k]\}$$
3. For a deterministic signal $x_n[k] = s[k]\ \forall n$ the ensemble average is
$$E\{f(s[k])\} = f(s[k])$$

Linear mean

The choice of the mapping function $f(\cdot)$ determines the property of the random process which is characterized by the ensemble average. The linear mean, which is given for $f(x[k]) = x[k]$, is the arithmetic mean value across the sample functions for a given time instant k.

Introducing $f(x[k]) = x[k]$ into the definition of the ensemble average yields

$$\mu_x[k] = E\{x[k]\} = \int_{-\infty}^{\infty} \theta \, p_x(\theta, k) \, \mathrm{d}\theta$$

where $\mu_x[k]$ is a common abbreviation of the linear mean. A process with $\mu_x[k] = 0$ is termed *zero-mean*. Note that μ_x should not be confused with the frequency index variable of the DFT.

Quadratic mean

The quadratic mean is given for $f(x[k]) = x^2[k]$

$$E\{x^2[k]\} = \int_{-\infty}^{\infty} \theta^2 \, p_x(\theta, k) \, \mathrm{d}\theta$$

It quantifies the mean instantaneous power of a sample function for a given time index k.

Variance

The variance is the quadratic mean of a zero-mean random process. It is given as

$$\sigma_x^2[k] = E\{(x[k] - \mu_x[k])^2\}$$

where $\sigma_x^2[k]$ is a common abbreviation of the variance, $\sigma_x[k]$ is known as the standard deviation. The variance characterizes how far the amplitude values of a random signal are spread out from its mean value.

The variance can be given in terms of the linear and quadratic mean as

$$\sigma_x^2[k] = E\{x^2[k]\} - \mu_x^2[k]$$

Excercise

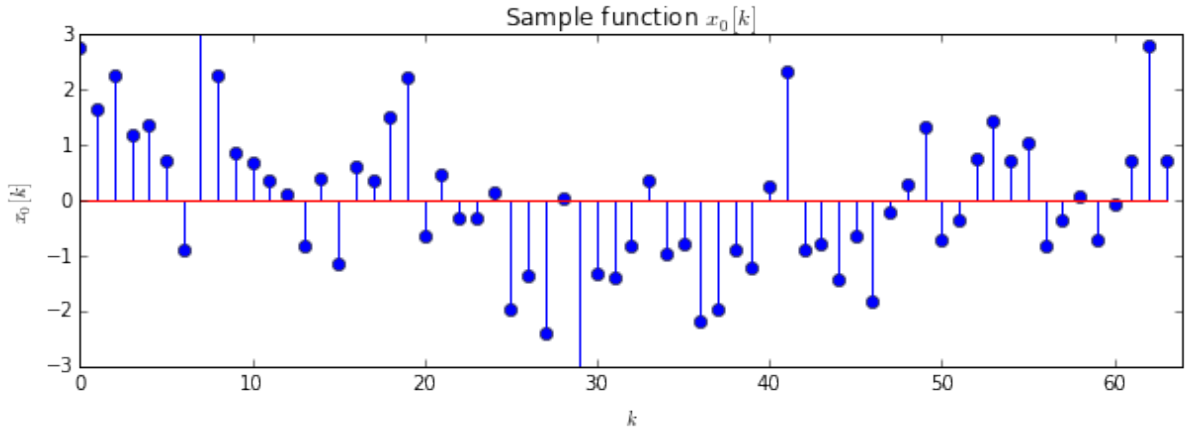
- Derive the relation above from the definitions and properties of the first order ensemble average

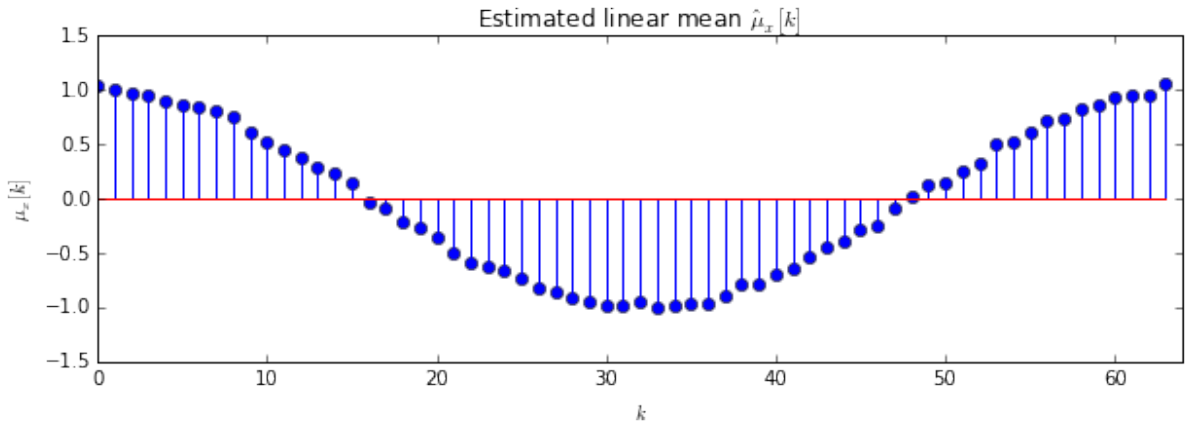
Linear/quadratic mean and variance of a random process

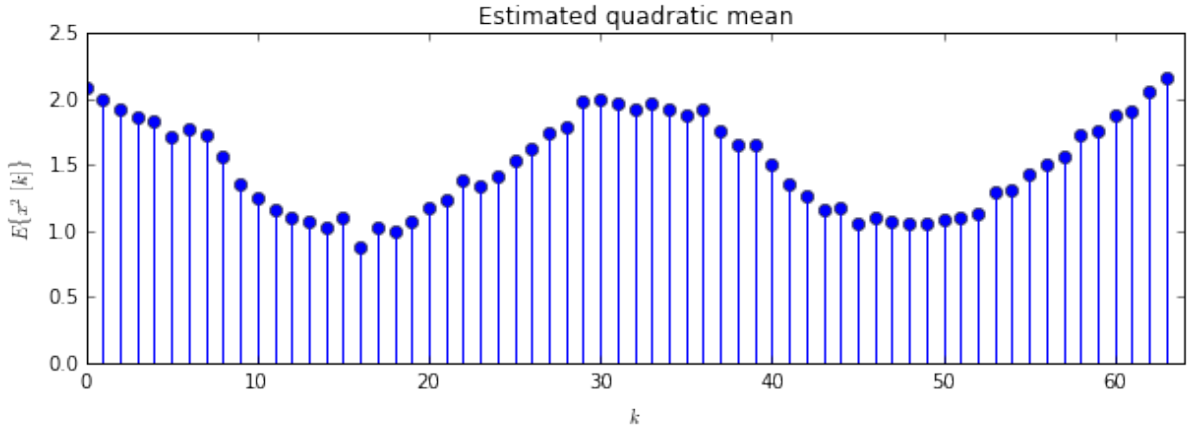
The following example shows the linear and quadratic mean, and variance of a random process. Since in practice only a limited number N of sample functions can be evaluated numerically these properties are only approximated/estimated.

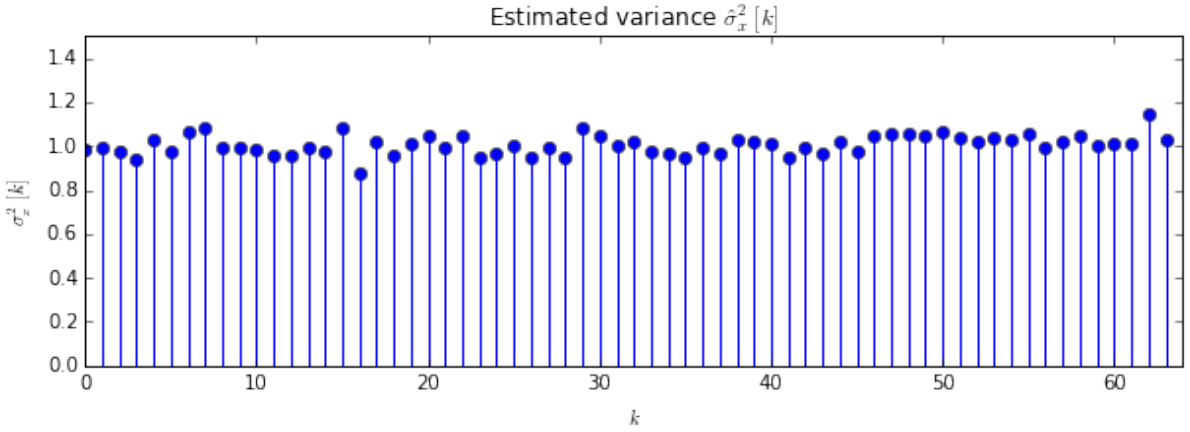
```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        K = 64  # number of random samples
        N = 1000  # number of sample functions


        # generate the sample functions
        x = np.random.normal(size=(N, K))
        x += np.tile(np.cos(2*np.pi/K*np.arange(K)), [N, 1])
```

```
# approximate the linear mean as ensemble average
mu = 1/N * np.sum(x, 0)
# approximate the quadratic mean
qu = 1/N * np.sum(x**2, 0)
# approximate the variance
sigma = 1/N * np.sum((x-mu)**2, 0)


# plot results
plt.rc('figure', figsize=(10, 3))

plt.figure()
plt.stem(x[0, :])
plt.title(r'Sample function $x_0[k]$')
plt.xlabel(r'$k$')
plt.ylabel(r'$x_0[k]$')
plt.axis([0, K, -3, 3])

plt.figure()
plt.stem(mu)
plt.title(r'Estimated linear mean $\hat{\mu}_x[k]$ ')
plt.xlabel(r'$k$')
plt.ylabel(r'$\mu_x[k]$')
plt.axis([0, K, -1.5, 1.5])

plt.figure()
plt.stem(qu)
plt.title(r'Estimated quadratic mean')
plt.xlabel(r'$k$')
plt.ylabel(r'$E\{x^2[k]\}$')
plt.axis([0, K, 0, 2.5])

plt.figure()
plt.stem(sigma)
plt.title(r'Estimated variance $\hat{\sigma}^2_x[k]$')
plt.xlabel(r'$k$')
plt.ylabel(r'$\sigma_x^2[k]$')
plt.axis([0, K, 0, 1.5]);
```









Exercise

- What does the linear and quadratic mean, and the variance tell you about the general behavior of the sample functions?
- Change the number `N` of sample functions and rerun the cell. What influence has a decrease/increase of the sample functions on the estimated ensemble averages?

2.4.2 Second Order Ensemble Averages

Definition

The second order ensemble average of two continuous-amplitude real-valued random signals $x[k]$ and $y[k]$ is defined as

$$E\{f(x[k_x], y[k_y])\} := \lim_{N \to \infty} \frac{1}{N} \sum_{n=0}^{N-1} f(x_n[k_x], y_n[k_y])$$

It can be given in terms of the bivariate PDF as

$$E\{f(x[k_x], y[k_y])\} = \int_{-\infty}^{\infty}\int f(\theta_x, \theta_y)\, p_{xy}(\theta_x, \theta_y, k_x, k_y)\, \mathrm{d}\theta_x\, \mathrm{d}\theta_y$$

Cross-correlation function

The cross-correlation function (CCF) of two random signals $x[k]$ and $y[k]$ is defined as the second order ensemble average for a linear mapping $f(x[k_x], y[k_y]) = x[k_x] \cdot y[k_y]$

$$\varphi_{xy}[k_x, k_y] = E\{x[k_x] \cdot y[k_y]\}$$

It characterizes the statistical dependencies of two random signals at two different time instants.

Auto-correlation function

The auto-correlation function (ACF) of a random signal $x[k]$ is defined as the second order ensemble average for a linear mapping $f(x[k_x], y[k_y]) = x[k_1] \cdot x[k_2]$

$$\varphi_{xx}[k_1, k_2] = E\{x[k_1] \cdot x[k_2]\}$$

It characterizes the statistical dependencies between the samples of a random signal at two different time instants.

2.5 Stationary Random Processes

2.5.1 Definition

When the statistical properties of a random process do not depend on the time index k, this process is termed as *stationary random process*. This can be expressed formally as

$$E\{f(x[k_1], x[k_2], \dots)\} = E\{f(x[k_1 + \Delta], x[k_2 + \Delta], \dots)\}$$

where $\Delta \in \mathbb{Z}$ denotes an arbitrary (temporal) shift. From this definition it becomes clear that

- random signals of finite length and
- deterministic signals

cannot be stationary random processes in a strict sense. However, in practice it is often assumed to be sufficient if above condition holds within the finite length of a random signal.

2.5.2 Cumulative Distribution Functions and Probability Density Functions

It follows from above definition of a stationary process, that the univariate cumulative distribution function (CDF) of a stationary random process does not depend on the time index k

$$P_x(\theta, k) = P_x(\theta)$$

the same holds for the univariate probability density function (PDF). The bivariate CDF of two stationary random signals $x[k]$ and $y[k]$ depends only on the difference $\kappa = k_x - k_y$

$$P_{xy}(\theta_x, \theta_y, k_x, k_y) = P_{xy}(\theta_x, \theta_y, \kappa)$$

The same holds for the bivariate PDF.

2.5.3 First Order Ensemble Averages

For a first order ensemble average of a stationary process the following relation must hold

$$E\{f(x[k])\} = E\{f(x[k + \Delta])\}$$

hence it cannot depend on the time index k.

For the linear mean we consequently get

$$\mu_x[k] = \mu_x$$

and for the variance

$$\sigma_x^2[k] = \sigma_x^2$$

2.5.4 Cross- and Auto-Correlation Function

Introducing the PDF's properties of a stationary process into the definition of the cross-correlation function (CCF) and auro-correlation function (ACF) it follows

$$\varphi_{xy}[k_x, k_y] = \varphi_{xy}[\kappa] = E\{x[k] \cdot y[k - \kappa]\} = E\{x[k + \kappa] \cdot y[k]\}$$

and

$$\varphi_{xx}[k_1, k_2] = \varphi_{xx}[\kappa] = E\{x[k] \cdot x[k - \kappa]\} = E\{x[k + \kappa] \cdot x[k]\}$$

2.6 Weakly Stationary Random Process

2.6.1 Definition

The definition of a stationary random process in the previous section must hold for any mapping function $f(\cdot)$. This cannot be checked in a strict sense for practical random processes. For a weakly (wide sense) stationary random process the conditions for stationarity must hold only for linear mappings. This leads to the following two conditions a weakly stationary random process has to fulfill

$$E\{x[k_1] \cdot x[k_2]\} = E\{x[k_1 + \Delta] \cdot x[k_2 + \Delta]\}$$

and

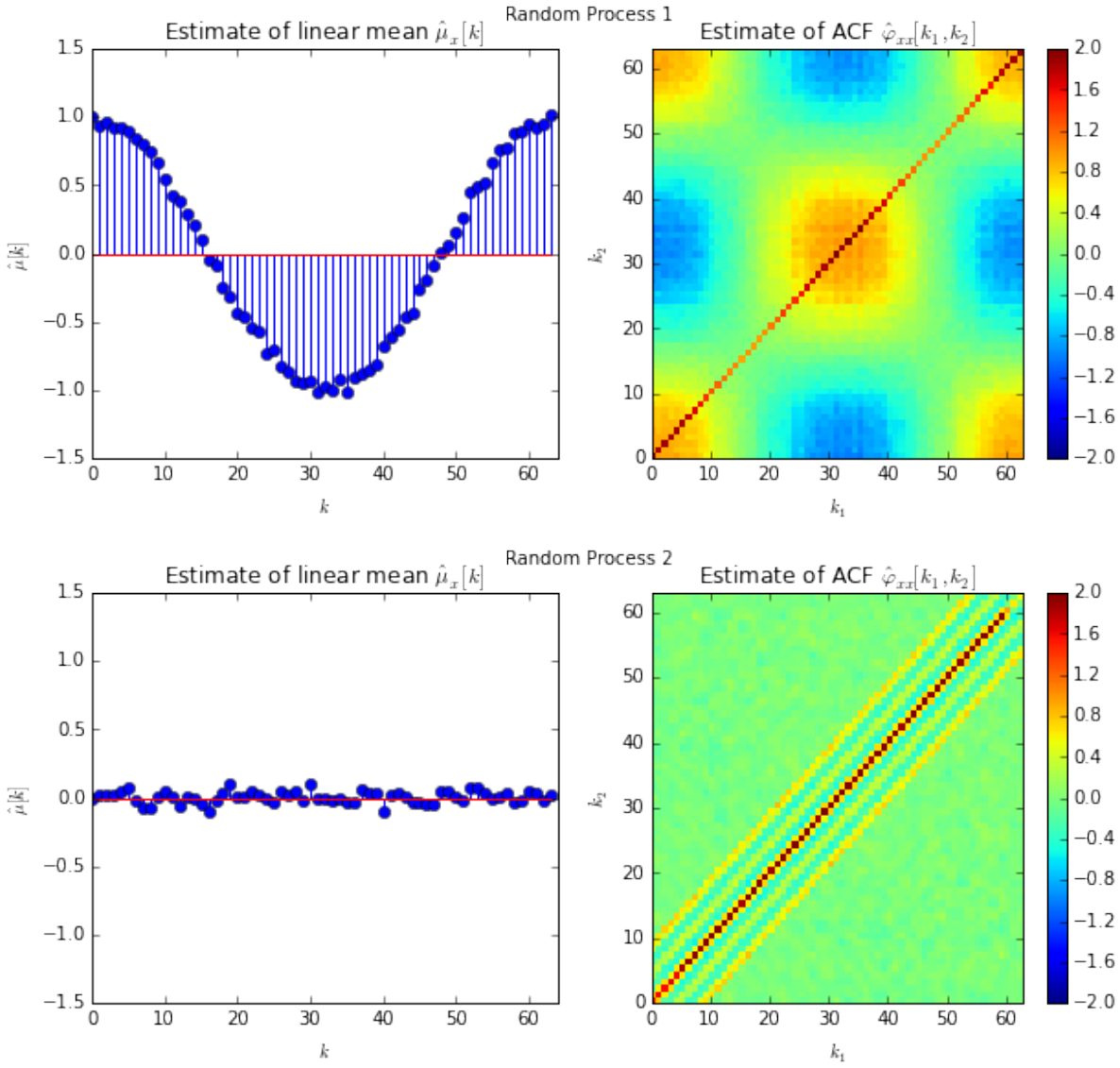
$$E\{x[k]\} = E\{x[k + \Delta]\}$$

A random signal of finite length cannot be weakly stationary in a strict sense.

2.6.2 Example

From above definition of a weakly stationary process it is evident that is sufficient to check the time dependence of the linear mean $x_\mu[k]$ and the auto-correlation function $\varphi_{xx}[k_1, k_2]$ of a random process. Both quantities are calculated and plotted for two different random processes.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        L = 64  # number of random samples
        N = 1000  # number of sample functions

        # generate sample functions
        x = np.random.normal(size=(N, L))
        x1 = x + np.tile(np.cos(2*np.pi/L*np.arange(L)), [N,1])
        h = 2*np.fft.irfft([1,1,1,0,0,0])
        x2 = np.asarray([np.convolve(x[n,:], h, mode='same') for n in range(N)])

        # compute and plot results
        def compute_plot_results(x):

            # estimate linear mean by ensemble average
            mu = 1/N * np.sum(x, 0)
            # estimate the auto-correlation function
            acf = np.zeros((L, L))
            for n in range(L):
                for m in range(L):
                    acf[n, m] = 1/N * np.sum(x[:, n]*x[:, m], 0)

            plt.subplot(121)
            plt.stem(mu)
            plt.title(r'Estimate of linear mean $\hat{\mu}_x[k]$')
            plt.xlabel(r'$k$')
            plt.ylabel(r'$\hat{\mu}[k]$')
            plt.axis([0, L, -1.5, 1.5])

            plt.subplot(122)
            plt.pcolor(np.arange(L), np.arange(L), acf, vmin=-2, vmax=2)
            plt.title(r'Estimate of ACF $\hat{\varphi}_{xx}[k_1, k_2]$')
            plt.xlabel(r'$k_1$')
            plt.ylabel(r'$k_2$')
            plt.colorbar()
            plt.autoscale(tight=True)


        plt.figure(figsize = (10, 4))
        plt.gcf().suptitle('Random Process 1')
        compute_plot_results(x1)

        plt.figure(figsize = (10, 4))
        plt.gcf().suptitle('Random Process 2')
        compute_plot_results(x2)
```



Exercise

- Which process can be assumed to be weakly stationary? Why?
- Increase the number `N` of sample functions. Do the results support your intial assumption?

2.7 Higher Order Temporal Averages

Ensemble averages are defined as the average across all sample functions $x_n[k]$ for particular time indexes. So far we did not consider temporal averaging to characterize a random signal. For a stationary process, the higher order temporal average along the n-th sample function is defined as

$$\overline{f(x_n[k], x_n[k-\kappa_1], x_n[k-\kappa_2], \dots)} = \lim_{K \to \infty} \frac{1}{2K+1} \sum_{k=-K}^{K} f(x_n[k], x_n[k-\kappa_1], x_n[k-\kappa_2], \dots)$$

2.8 Ergodic Random Processes

An ergodic process is a stationary random process whose higher order temporal averages of all sample functions are equal to the ensemble averages

$$\overline{f(x_n[k], x_n[k-\kappa_1], x_n[k-\kappa_2], \dots)} = E\{f(x[k], x[k-\kappa_1], x[k-\kappa_2], \dots)\} \quad \forall n$$

This implies that all higher order temporal averages are equal. Any sample function from the process represents the average statistical properties of the entire process. The ensemble averages for a stationary and ergodic random process are given by the temporal averages of one sample function. This result is very important for the practical computation of statistical properties of random signals.

2.9 Weakly Ergodic Random Processes

2.9.1 Definition

As for a weakly stationary process, the conditions for ergodicity have to hold only for linear mappings $f(\cdot)$. Under the assumption of a weakly stationary process, the following two conditions have to be met by a weakly (wide sense) ergodic random process

$$\overline{x_n[k] \cdot x_n[k-\kappa]} = E\{x[k] \cdot x[k-\kappa]\} \quad \forall n$$

and

$$\overline{x_n[k]} = E\{x[k]\} \quad \forall n$$

2.9.2 Example

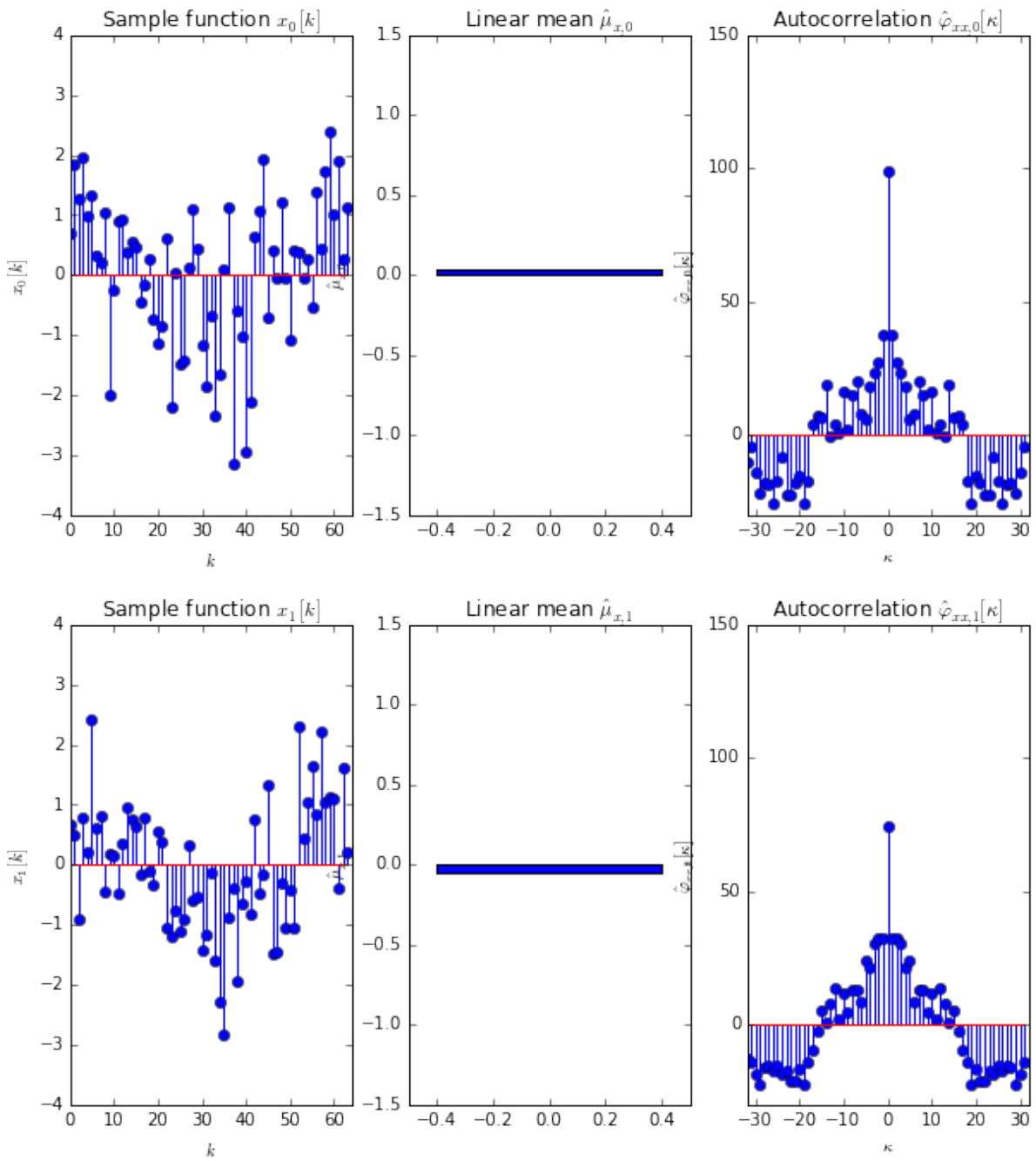
In the following example, the linear mean and autocorrelation function are computed as ensemble and temporal averages for three random processes. The plots show the estimated temporal averages $\hat{\mu}_{x,n}$ and $\hat{\varphi}_{xx,n}[\kappa]$ on the right side of the sample functions $x_n[k]$. Note, the linear mean as temporal average is a scalar value $\hat{\mu}_{x,n}$ which has been plotted by a bar plot. The ensemble averages $\hat{\mu}[k]$ and $\hat{\varphi}[k_1, k_2]$ are shown below the sample functions to indicate the averaging across sample functions.

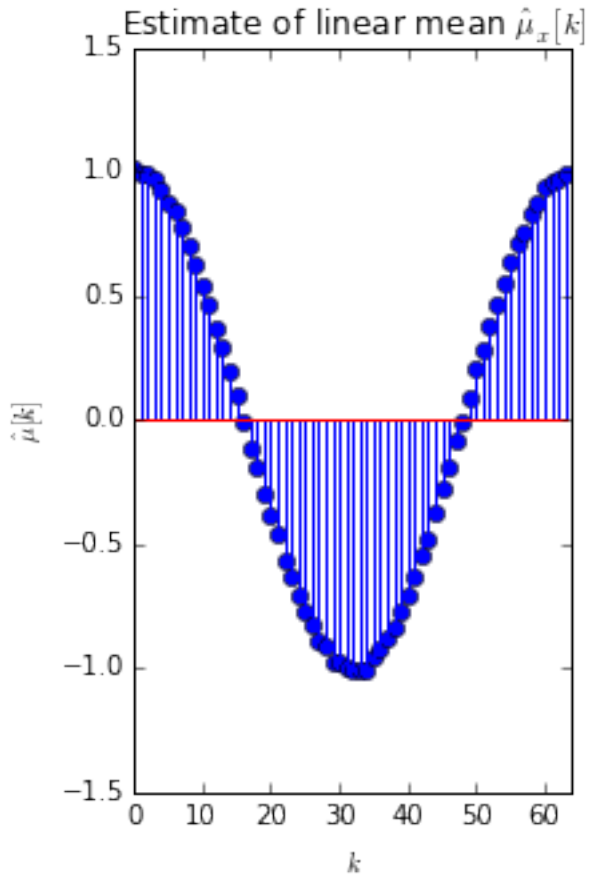
```
In [2]: L = 64  # number of random samples
        N = 10000  # number of sample functions

        # generate sample functions
        x = np.random.normal(size=(N, L))
        k = np.arange(L)
        x1 = x + np.tile(np.cos(2*np.pi/L*k), [N, 1])
        x2 = x + np.tile([np.ones(L), -np.ones(L)], [N//2, 1])
        x3 = x + np.ones([N, L])

        # function to compute and plot results
        def compute_plot_results(x):

            # estimate linear mean by ensemble average
            mu = 1/N * np.sum(x, 0)
            # estimate the auto-correlation function by ensemble average
            acf = np.zeros((L, L))
            for n in range(L):
                for m in range(L):
                    acf[n, m] = 1/N * np.sum(x[:, n]*x[:, m], 0)
            # estimate linear mean as temporal average
            mut = 1/L * np.sum(x, 1)
```

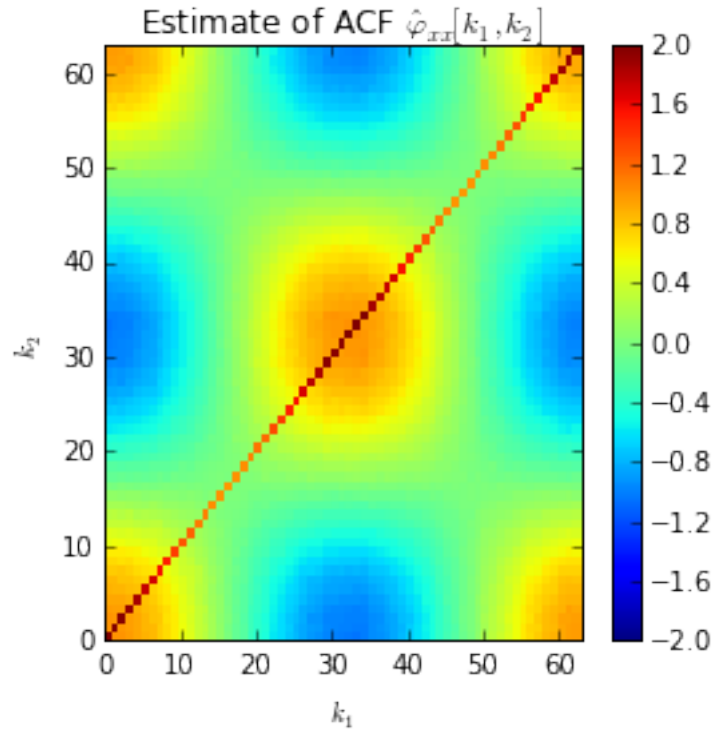
```
    # estimate the auto-correlation function as temporal average
    acft = np.zeros((N, L))
    for n in range(N):
        acft[n, :] = np.correlate(x[n, :], x[n, :], mode='same')
    kappa = np.arange(L) - L//2

    for n in range(2):
        plt.figure(figsize = (10, 5))
        plt.subplot(131)
        plt.stem(x[n, :])
        plt.title(r'Sample function $x_%d[k]$'%n)
        plt.xlabel(r'$k$')
        plt.ylabel(r'$x_%d[k]$'%n)
        plt.axis([0, L, -4, 4])

        plt.subplot(132)
        plt.bar(-0.4, mut[n])
        plt.title(r'Linear mean $\hat{\mu}_{x,%d}$'%n)
        plt.ylabel(r'$\hat{\mu}_{x,%d}$'%n)
        plt.axis([-.5, .5, -1.5, 1.5])

        plt.subplot(133)
        plt.stem(kappa, acft[n, :])
        plt.title(r'Autocorrelation $\hat{\varphi}_{xx,%d}[\kappa]$'%n)
        plt.xlabel(r'$\kappa$')
        plt.ylabel(r'$\hat{\varphi}_{xx,%d}[\kappa]$'%n)
        plt.axis([-L//2, L//2, -30, 150])

    plt.figure(figsize = (10, 5))
    plt.subplot(131)
    plt.stem(mu)
    plt.title(r'Estimate of linear mean $\hat{\mu}_x[k]$')
    plt.xlabel(r'$k$')
    plt.ylabel(r'$\hat{\mu}[k]$')
    plt.axis([0, L, -1.5, 1.5])

    plt.figure(figsize = (4, 4))
    plt.pcolor(k, k, acf, vmin=-2, vmax=2)
    plt.title(r'Estimate of ACF $\hat{\varphi}_{xx}[k_1, k_2]$')
    plt.xlabel(r'$k_1$')
    plt.ylabel(r'$k_2$')
    plt.colorbar()
    plt.autoscale(tight=True)
```

Random Process 1

```
In [3]: compute_plot_results(x1)
```

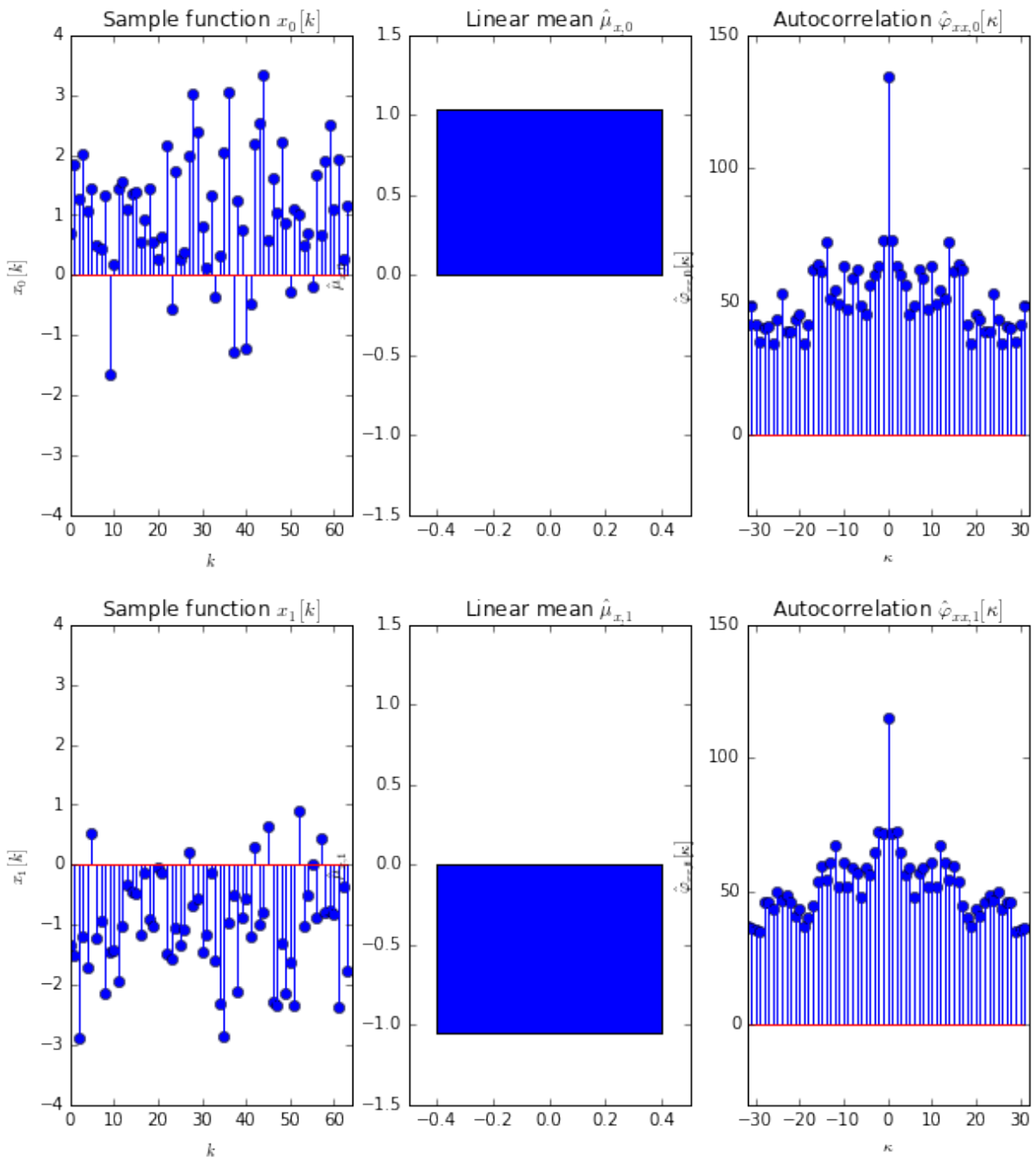

Sample function $x_0[k]$
Linear mean $\hat{\mu}_{x,0}$
Autocorrelation $\hat{\varphi}_{xx,0}[\kappa]$
Sample function $x_1[k]$
Linear mean $\hat{\mu}_{x,1}$
Autocorrelation $\hat{\varphi}_{xx,1}[\kappa]$
k
κ

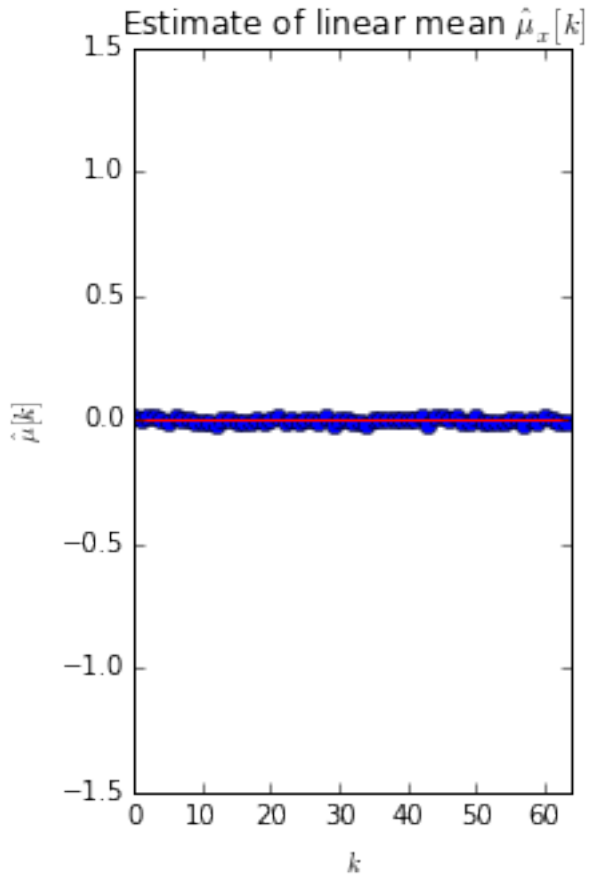


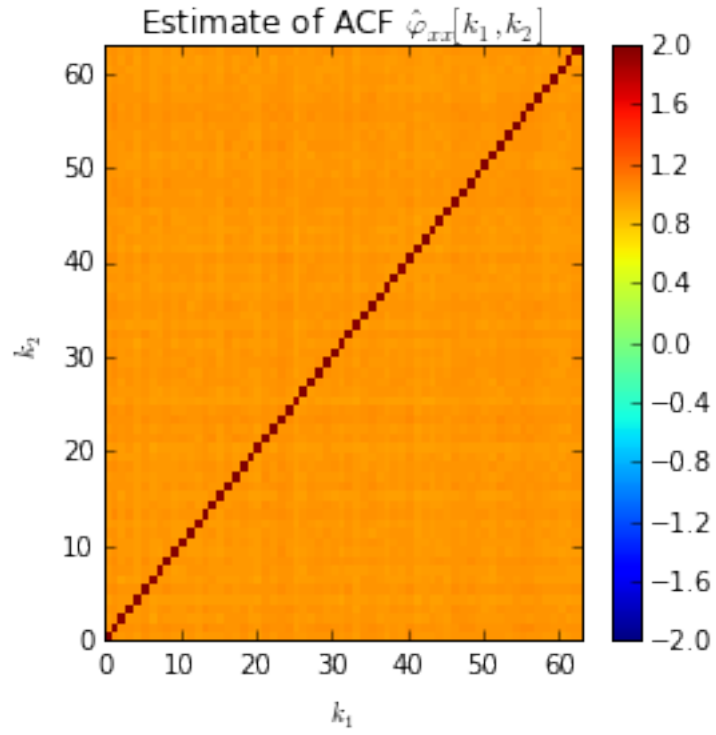


Random Process 2

```
In [4]: compute_plot_results(x2)
```

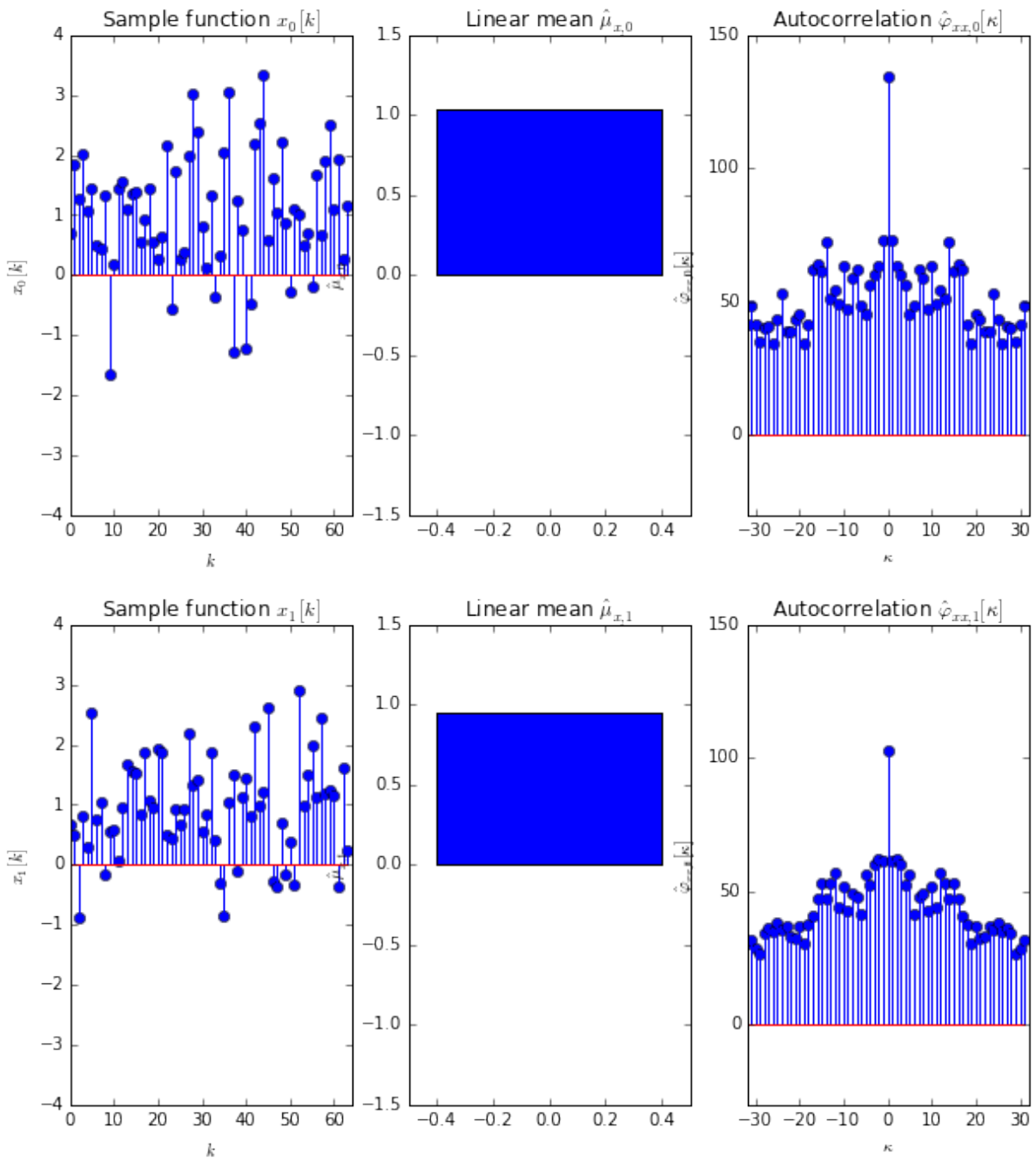

Sample function $x_0[k]$
Linear mean $\hat{\mu}_{x,0}$
Autocorrelation $\hat{\varphi}_{xx,0}[\kappa]$
$x_0[k]$
k
κ
Sample function $x_1[k]$
Linear mean $\hat{\mu}_{x,1}$
Autocorrelation $\hat{\varphi}_{xx,1}[\kappa]$
$x_1[k]$
k
κ

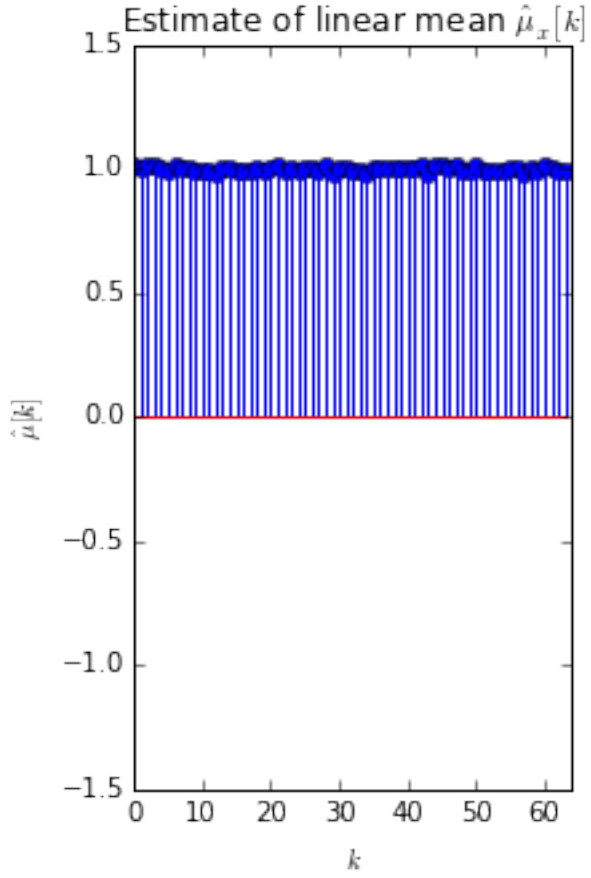


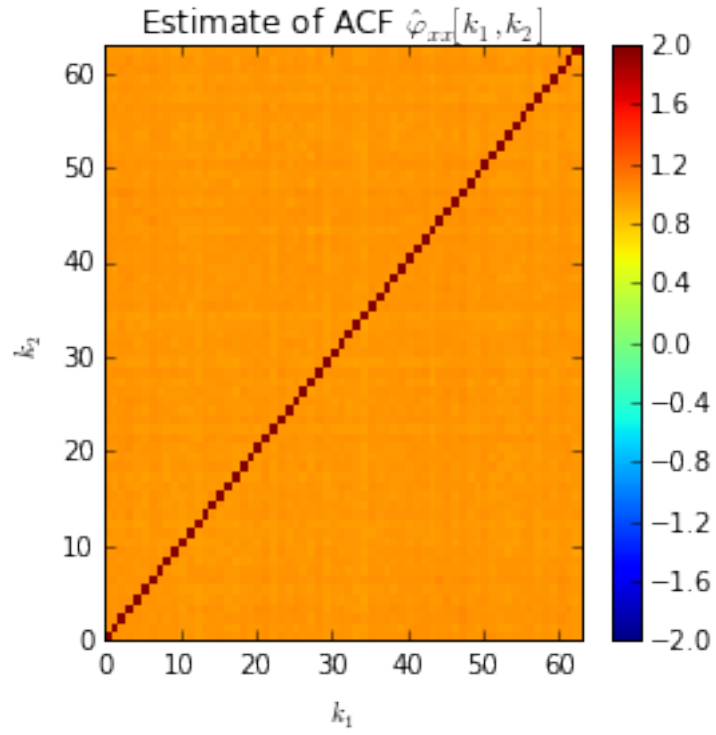


Random Process 3

```
In [5]: compute_plot_results(x3)
```


Sample function $x_0[k]$
Linear mean $\hat{\mu}_{x,0}$
Autocorrelation $\hat{\varphi}_{xx,0}[\kappa]$
$x_0[k]$
k
κ
Sample function $x_1[k]$
Linear mean $\hat{\mu}_{x,1}$
Autocorrelation $\hat{\varphi}_{xx,1}[\kappa]$
$x_1[k]$
k
κ





Exercise

- Which process can be assumed to be stationary and/or ergodic? Why?

2.10 Auto-Correlation Function

The auto-correlation function (ACF) characterizes the temporal dependencies of a random signal $x[k]$. It is an important measure for the analysis of signals in communications engineering, coding and system identification.

2.10.1 Definition

The ACF of a continuous-amplitude real-valued weakly stationary process $x[k]$ is given as

$$\varphi_{xx}[\kappa] = E\{x[k] \cdot x[k-\kappa]\}$$

where κ is commonly chosen as sample index instead of k in order to indicate that it denotes a shift/lag. The ACF quantifies the similarity of a signal with a shifted version of itself. It has high values for high similarity and low values for low similarity.

If the process is additionally weakly ergodic, the ACF can be computed by averaging along one sample function

$$\varphi_{xx}[\kappa] = \lim_{K \to \infty} \frac{1}{2K+1} \sum_{k=-K}^{K} x[k] \cdot x[k-\kappa]$$

Note that the normalization on the left side of the sum is discarded in some definitions of the ACF. Above summation resembles strongly the definition of the discrete convolution. For a random signal $x_N[k] = \text{rect}_N[k] \cdot x[k]$ of finite length N and by exploiting the properties of a weakly ergodic random process one yields

$$\varphi_{xx}[\kappa] = \frac{1}{N} \sum_{k=0}^{N-1} x_N[k] \cdot x_N[k-\kappa] = \frac{1}{N} \; x_N[k] * x_N[-k]$$

where the ACF $\varphi_{xx}[\kappa] = 0$ for $|\kappa| > N-1$. Hence, the ACF can be computed by (fast) convolution of the random signal with a time reversed version of itself.

Note in practical implementations (e.g. Python), the computed ACF is stored in a vector of length $2N-1$. The positive indexes $0, 1, \ldots, 2N-1$ of this vector cannot be directly interpreted as κ. The indexes of the vector have to be shifted by $N-1$.

2.10.2 Properties

The following properties of the ACF can be deduced from its definition

1. The ACF $\varphi_{xx}[\kappa]$ has a maximum for $\kappa = 0$. It is given as

   $$\varphi_{xx}[0] = E\{x^2[k]\} = \sigma_x^2 + \mu_x^2$$

   This is due to the fact that the signal is equal to itself for $\kappa = 0$. Please note that for periodic random signals more than one maximum will be present.

2. The ACF is a function with even symmetry

   $$\varphi_{xx}[\kappa] = \varphi_{xx}[-\kappa]$$

3. For typical random signals, the ACF approaches the limiting value

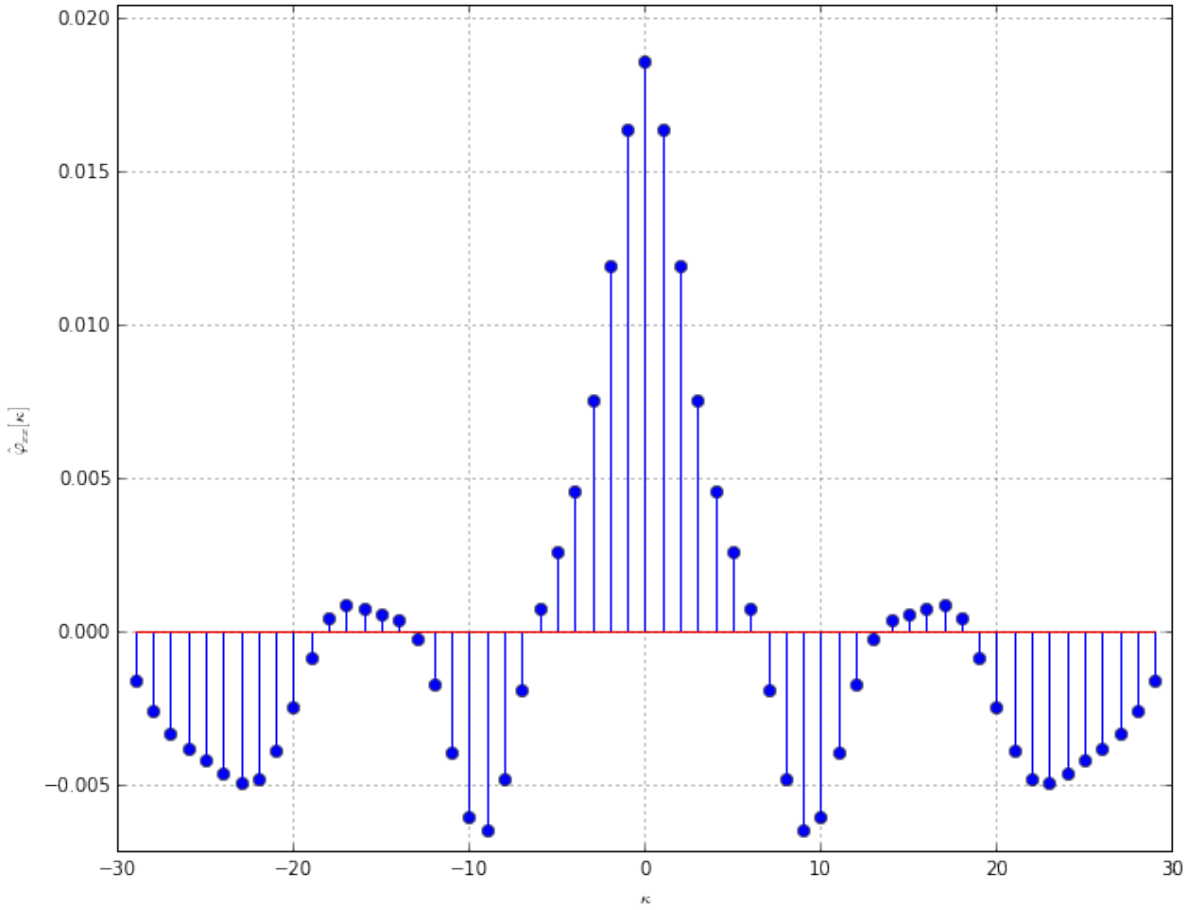
   $$\lim_{|\kappa| \to \infty} \varphi_{xx}[\kappa] = \mu_x^2$$

   The similarity of a typical random signal is often low for large lags κ.

2.10.3 Example

The following example computes and plots the ACF for a speech signal.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        from scipy.io import wavfile

        K = 30  # limit for lags in ACF

        # read sound file
        fs, x = wavfile.read('../data/speech_8k.wav')
        x = np.asarray(x, dtype=float)/2**15

        # compute and truncate ACF
        acf = 1/len(x) * np.correlate(x, x, mode='full')
        acf = acf[(len(x)-1)-(K-1):(len(x)-1)+K]
        kappa = np.arange(-(K-1), K)

        # plot ACF
        fig = plt.figure(figsize = (10, 8))

        plt.stem(kappa, acf)
        plt.xlabel(r'$\kappa$')
        plt.ylabel(r'$\hat{\varphi}_{xx}[\kappa]$')
        plt.axis([-K, K, 1.1*min(acf), 1.1*max(acf)]);
        plt.grid()
```



Exercise

- Does the ACF fulfill the properties stated above?
- The plot shows only a small part of the ACF. Increase the range K for $-K \leq \kappa < K$ and check if the last property is fulfilled for the speech signal.

2.11 Auto-Covariance Function

The auto-covariance function is the ACF for the zero-mean random signal $x[k] - \mu_x$. It is given as

$$\psi_{xx}[\kappa] = \varphi_{xx}[\kappa] - \mu_x^2$$

2.12 Cross-Correlation Function

The cross-correlation function (CCF) is a measure of similarity that two random signals $x[k]$ and $y[k - \kappa]$ have with respect to the temporal shift $\kappa \in \mathbb{Z}$.

2.12.1 Definition

The CCF of two continuous-amplitude real-valued weakly stationary processes $x[k]$ and $y[k]$ is given as

$$\varphi_{xy}[\kappa] = E\{x[k] \cdot y[k - \kappa]\} = E\{x[k + \kappa] \cdot y[k]\}$$

If $x[k]$ and $y[k]$ are weakly ergodic processes, the CCF can be computed by averaging along one sample function

$$\varphi_{xy}[\kappa] = \lim_{K \to \infty} \frac{1}{2K + 1} \sum_{k=-K}^{K} x[k] \cdot y[k - \kappa]$$

For random signals $x_N[k] = \text{rect}_N[k] \cdot x[k]$ and $y_M[k] = \text{rect}_M[k] \cdot Y[k]$ of finite lengths N and M one yields

$$\varphi_{xy}[\kappa] = \frac{1}{N} \sum_{k=0}^{N-1} x[k] \cdot y[k - \kappa] = \frac{1}{N}\, x[k] * y[-k]$$

where the CCF $\varphi_{xy}[\kappa] = 0$ for $\kappa < -(M-1)$ and $\kappa > N-1$. The CCF can be computed by (fast) convolution of one random signal with a time reversed version of the other random signal. Note in practical implementations (e.g. Python), the computed CCF is stored in a vector of length $N + M - 1$. The positive indexes $0, 1, \ldots, N + M - 1$ of this vector cannot be directly interpreted as κ. The indexes of the vector have to be shifted by $M - 1$.

Above definitions hold also for the CCF $\varphi_{yx}[\kappa]$ when exchanging $x[k]$ with $y[k]$ and N with M.

2.12.2 Properties

1. For an exchange of the two random signals, the CCF exhibits the following symmetry

   $$\varphi_{xy}[\kappa] = \varphi_{yx}[-\kappa]$$

2. The CCF is constant for *uncorrelated* random signals

   $$\varphi_{xy}[\kappa] = \mu_x \cdot \mu_y$$

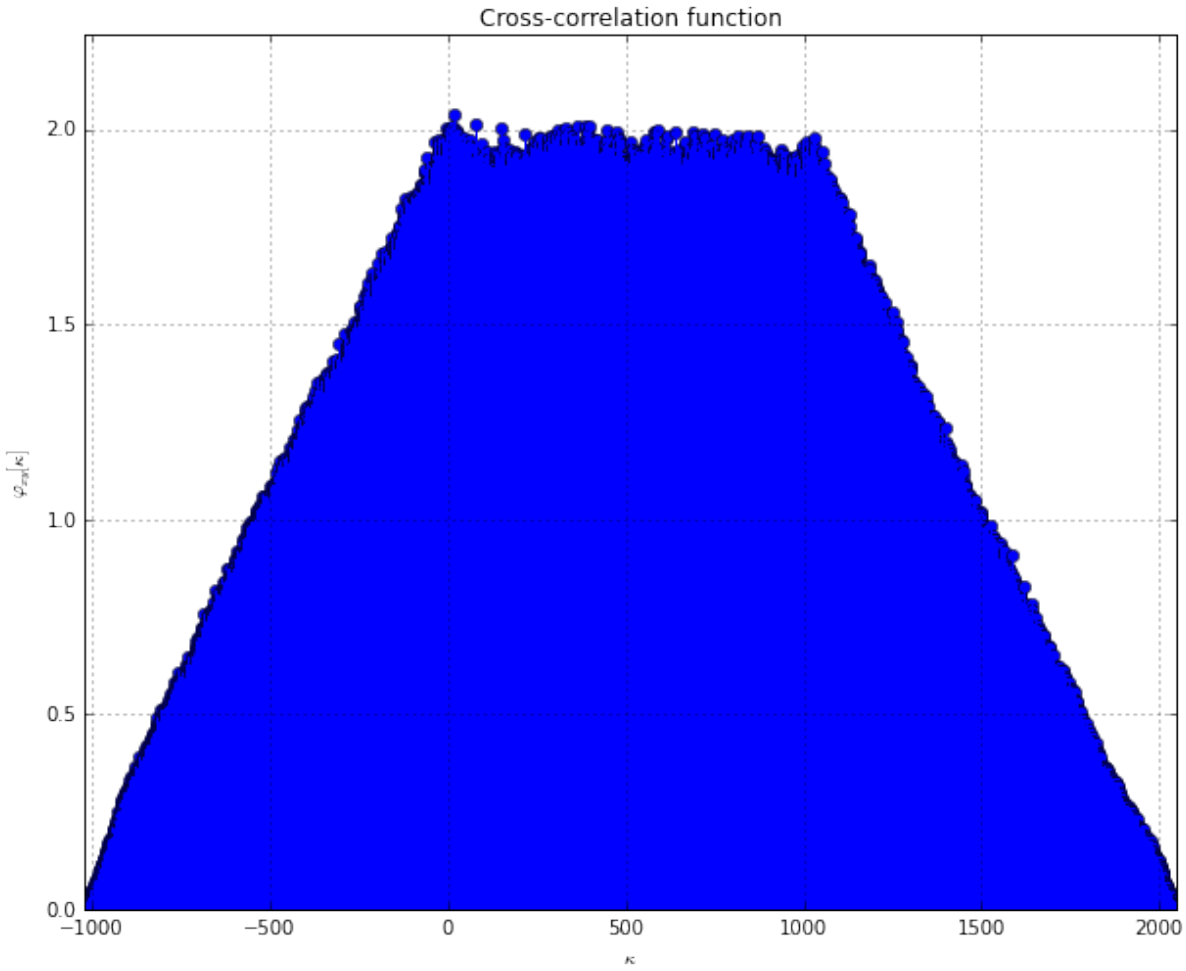
   Typical random processes are uncorrelated for $|\kappa| \to \infty$.

2.12.3 Example

The following example computes the CCF for two uncorrelated random signals

```
In [18]: K = 1024  # length of random signals

         # generate two uncorrelated random signals
         x = 2+np.random.normal(size=K)
         y = 1+np.random.normal(size=2*K)

         # compute CCF
         ccf = 1/len(x) * np.correlate(x, y, mode='full')
         kappa = np.arange(-(K-1), 2*K)

         # print mean values of signals
         print('Mean of signal x[k]: %f' %np.mean(x))
         print('Mean of signal y[k]: %f' %np.mean(y))

         # plot CCF
         plt.figure(figsize = (10, 8))
         plt.stem(kappa, ccf)
         plt.title('Cross-correlation function')
         plt.ylabel(r'$\varphi_{xy}[\kappa]$')
         plt.xlabel(r'$\kappa$')
         plt.axis([kappa[0], kappa[-1], 0, 1.1*max(ccf)]);
         plt.grid()
```

```
Mean of signal x[k]: 2.049598
Mean of signal y[k]: 0.944363
```



Exercise

- Why does the CCF of two finite length signals have this trapezoid like shape?
- What would be its theoretic value for signals of infinte length?

2.13 Cross-Covariance Function

The cross-covariance function is the CCF for the zero-mean random signals $x[k] - \mu_x$ and $y[k] - \mu_y$. It is given as

$$\psi_{xy}[\kappa] = \varphi_{xy}[\kappa] - \mu_x \mu_y$$

Exercise

- How would the plot for $\psi_{xy}[\kappa]$ look like for above example?

2.14 Power Spectral Density

The power spectral density (PSD) is the Fourier transformation of the auto-correlation function (ACF).

2.14.1 Definition

For a continuous-amplitude real-value weakly stationary process the PSD is defined as

$$\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathcal{F}_*\{\varphi_{xx}[\kappa]\}$$

where $\mathcal{F}_*\{\cdot\}$ denotes the discrete-time Fourier transformation (DTFT). The PSD quantifies the power per frequency for a random signal.

2.14.2 Properties

The properties of the PSD can be deduced from the properties of the ACF and the DTFT as

1. From the symmetry of the ACF it follows

$$\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{xx}(\mathrm{e}^{\,-\mathrm{j}\,\Omega})$$

2. The quadratic mean of a random signal is given as

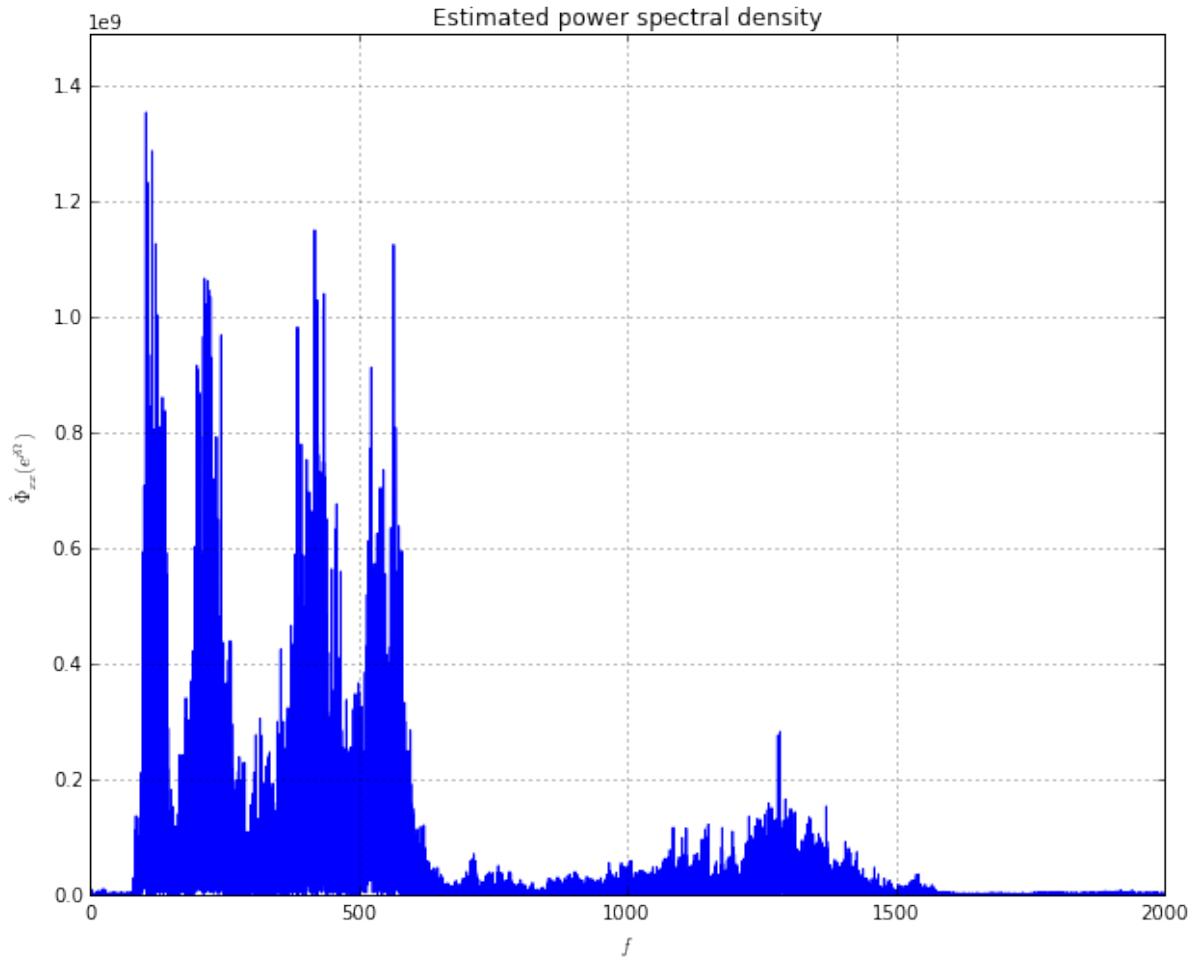
$$E\{x[k]^2\} = \varphi_{xx}[0] = \frac{1}{2\pi} \int_{-\pi}^{\pi} \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\,\mathrm{d}\Omega$$

   The last relation can be found by introducing the definition of the inverse DTFT.

2.14.3 Example

In this example the PSD $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of a speech signal $x[k]$ is computed by applying a discrete Fourier transformation (DFT) to the auto-correlation function. For better interpretation of the PSD, the frequency axis $f = \frac{\Omega}{2\pi}\, f_s$ has been chosen, where f_s denotes the sampling frequency of the signal.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        from scipy.io import wavfile

        # read sound file
        fs, x = wavfile.read('../data/speech_8k.wav')
        x = np.asarray(x, dtype=float)

        # compute and truncate ACF
        acf = 1/len(x) * np.correlate(x, x, mode='full')
        # compute PSD
        psd = np.fft.rfft(acf)
        f = fs/(2*len(x)) * np.arange(len(x))

        # plot PSD
        plt.figure(figsize = (10, 8))
        plt.plot(f, np.abs(psd))
        plt.title('Estimated power spectral density')
        plt.ylabel(r'$\hat{\Phi}_{xx}(e^{j \Omega})$')
        plt.xlabel(r'$f$')
        plt.axis([0, 2000, 0, 1.1*max(np.abs(psd))]);
        plt.grid()
```



Exercise

- What does the PSD tell you about the spectral contents of a speech signal?

2.15 Cross-Power Spectral Density

The cross-power spectral density is the Fourier transformation of the cross-correlation function (CCF). It is defined as follows

$$\Phi_{xy}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathcal{F}_*\{\varphi_{xy}[\kappa]\}$$

The symmetries of $\Phi_{xy}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ can be derived from the symmetries of the CCF and the DTFT as

$$\underbrace{\Phi_{xy}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{xy}^*(\mathrm{e}^{-\,\mathrm{j}\,\Omega})}_{\varphi_{xy}[\kappa]\in\mathbb{R}} = \underbrace{\Phi_{yx}(\mathrm{e}^{-\,\mathrm{j}\,\Omega}) = \Phi_{yx}^*(\mathrm{e}^{\,\mathrm{j}\,\Omega})}_{\varphi_{yx}[-\kappa]\in\mathbb{R}}$$

2.16 Important Distributions

Analytic cumulative distribution functions (CDFs) and probability density functions (PDFs) are frequently used as models for practical random processes. They allow to describe the statistical properties of a random process by a few parameters. These parameters are fitted to an actual random process and are used in algorithms for statistical signal processing. For the following, weakly stationary random processes are assumed.

2.16.1 Uniform Distribution

Definition

The PDF of the uniform distribution is given as

$$p_x(\theta) = \begin{cases} \frac{1}{b-a} & \text{for } a \leq \theta \leq b \\ 0 & \text{otherwise} \end{cases}$$

where a and b denote the lower and upper bound for the amplitude of the random signal $x[k]$. The uniform distribution assumes that all amplitudes between these bounds occur with the same probability. The CDF can be derived from the PDF by integration over θ

$$P_x(\theta) = \begin{cases} 0 & \text{for } \theta < a \\ \frac{\theta - a}{b-a} & \text{for } a \leq \theta < b \\ 1 & \text{for } \theta \geq b \end{cases}$$

The linear mean computes to

$$\mu_x = \frac{a+b}{2}$$

and the variance is

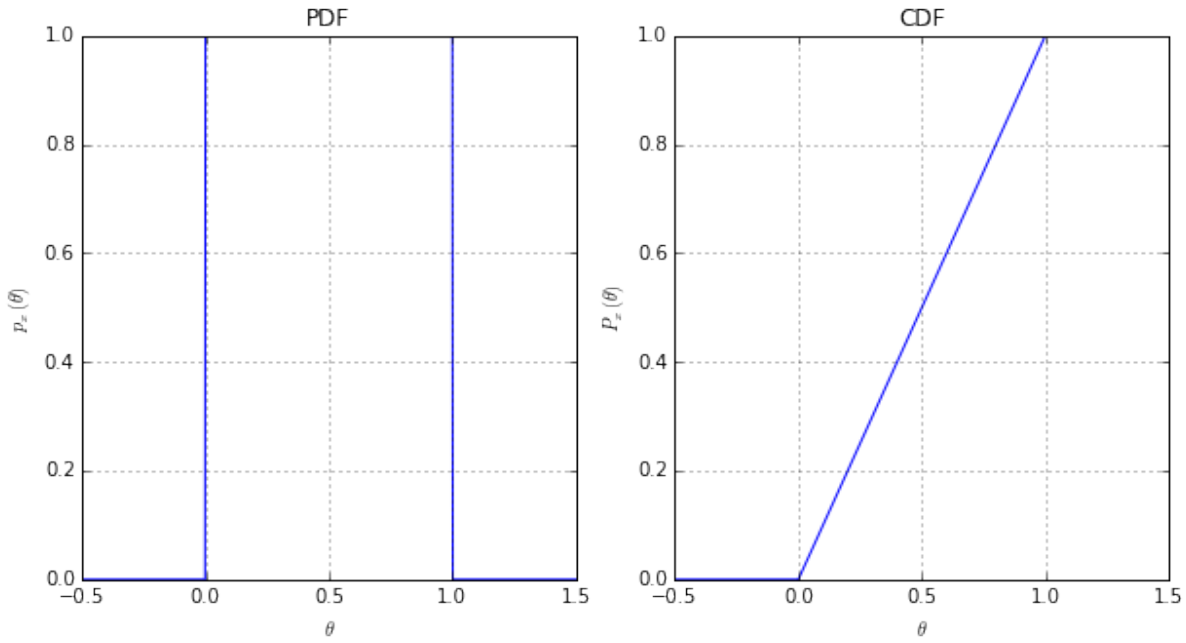
$$\sigma_x^2 = \frac{(b-a)^2}{12}$$

In order to plot the PDF and CDF of the various distributions, a function is defined

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.stats as stats

        def plot_pdf_cdf(x, distr):
            plt.figure(figsize = (10, 5))

            plt.subplot(121)
            plt.plot(x, distr.pdf(x))
            plt.xlabel(r'$\theta$')
            plt.ylabel(r'$p_x(\theta)$')
            plt.title('PDF')
            plt.grid()

            plt.subplot(122)
            plt.plot(x, distr.cdf(x))
            plt.xlabel(r'$\theta$')
            plt.ylabel(r'$P_x(\theta)$')
            plt.title('CDF')
            plt.grid()
```

The PDF/CDF for a uniformly distributed random signal with $a = 0$ and $b = 1$ is plotted

```
In [2]: plot_pdf_cdf(np.linspace(-.5, 1.5, num=1000), stats.uniform)
```



Example

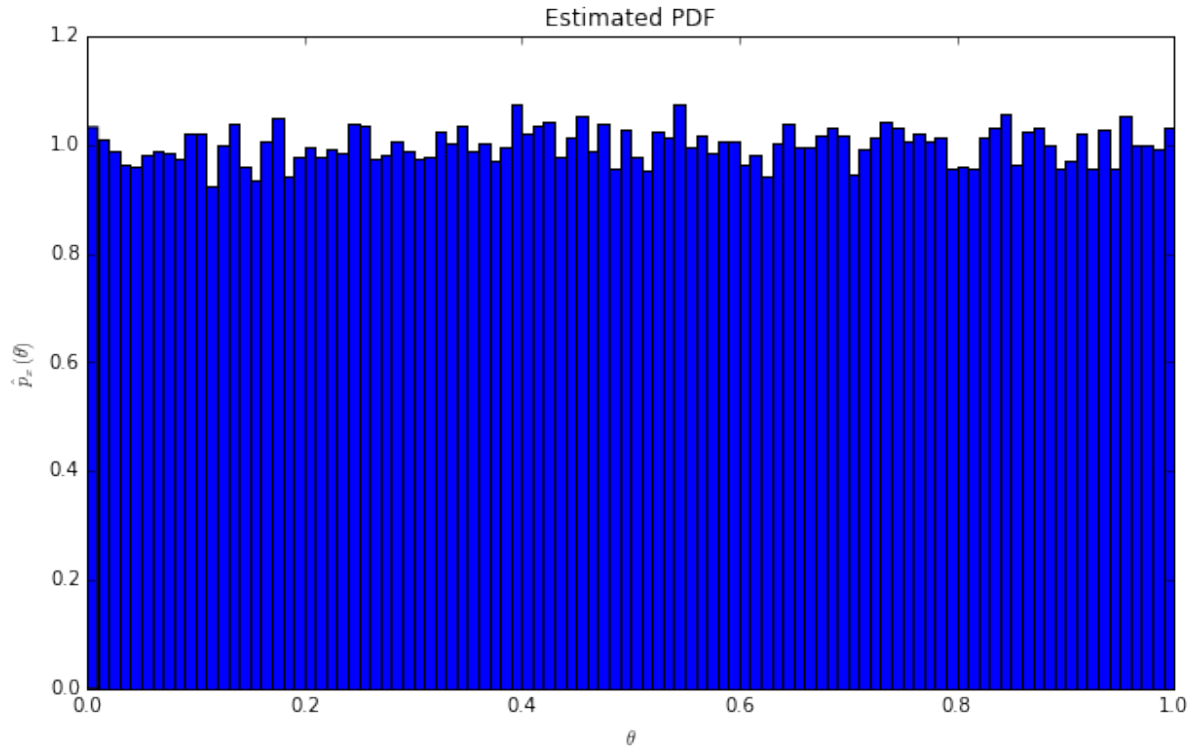
Most software frameworks for numerical mathematics provide functions to generate random samples with a defined PDF. So does `'Numpy <http://docs.scipy.org/doc/numpy/reference/routines.random.html>'__` or `'scipy.stats <http://docs.scipy.org/doc/scipy/reference/stats.html#continuous-distributions>'__`. We again first define a function that computes and plots the PDF and CDF of a given random signal.

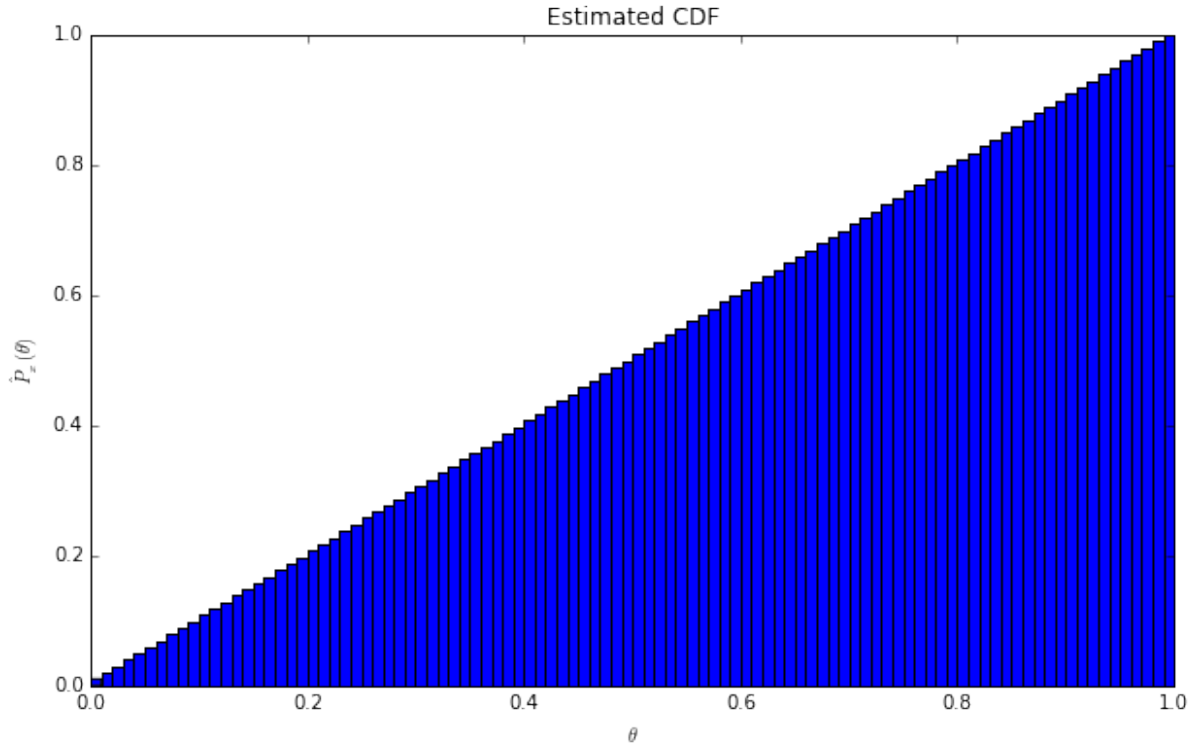
```
In [3]: def compute_plot_pdf_cdf(x, nbins=100):

            plt.figure(figsize = (10, 6))
            plt.hist(x, nbins, normed=True)
            plt.title('Estimated PDF')
            plt.xlabel(r'$\theta$')
            plt.ylabel(r'$\hat{p}_x(\theta)$')

            plt.figure(figsize = (10, 6))
            plt.hist(x, nbins, cumulative=True, normed=True)
            plt.title('Estimated CDF')
            plt.xlabel(r'$\theta$')
            plt.ylabel(r'$\hat{P}_x(\theta)$')

            print('Linear mean: %f' %np.mean(x))
            print('Variance: %f' %np.var(x))
```

For the standard uniform distribution ($a = 0, b = 1$) we get the following results

```
In [4]: compute_plot_pdf_cdf(stats.uniform.rvs(size=100000), nbins=100)
```

```
Linear mean: 0.501043
Variance: 0.083031
```





Exercise

- Why is the estimate of the CDF smoother that the estimate of the PDF?
- How can the upper b and lower bound a be changed in above code?
- What changes if you change the length of the random signal or the number `nbins` of histogram bins?

2.16.2 Normal Distribution

Definition

The PDF of the normal distribution/Gaussian distribution is given as

$$p_x(\theta) = \frac{1}{\sqrt{2\pi}\sigma_x} \mathrm{e}^{-\frac{(\theta-\mu_x)^2}{2\sigma_x^2}}$$

where μ_x and σ_x^2 denote the linear mean and variance, respectively. Normal distributions are often used to represent random variables whose distributions are not known. The central limit theorem states that averages of random variables independently drawn from independent distributions become normally distributed when the number of random variables is sufficiently large. As a result, random signals that are expected to be the sum of many independent processes often have distributions that are nearly normal. The CDF can be derived by integration over θ

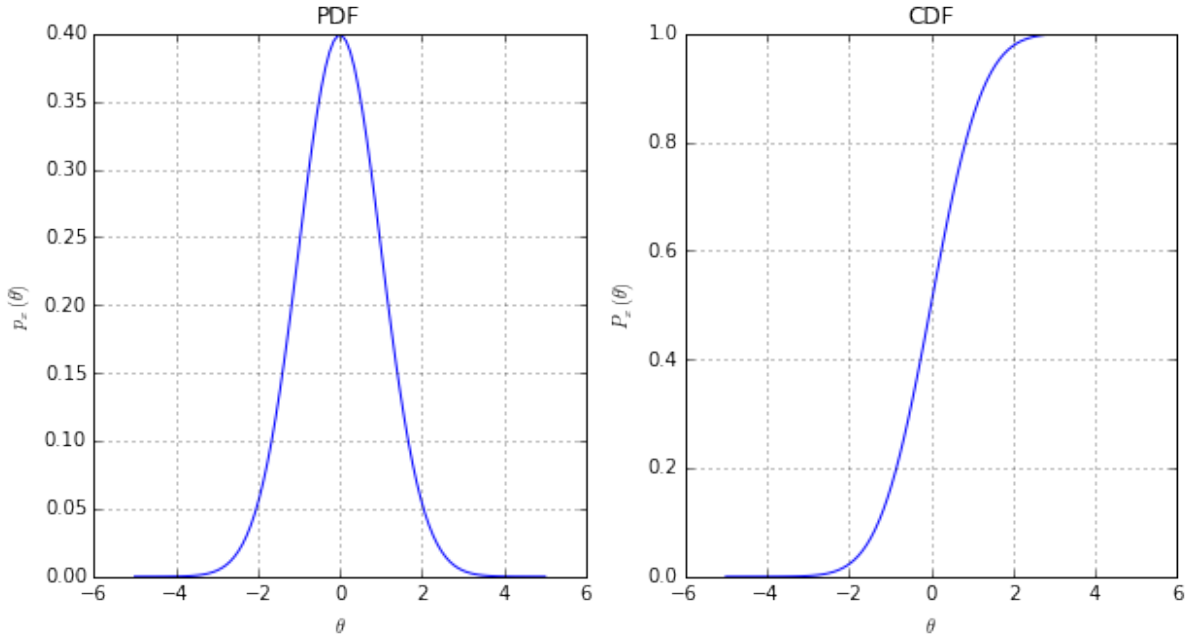
$$P_x(\theta) = \frac{1}{\sqrt{2\pi}\sigma_x} \int_{-\infty}^{\theta} \mathrm{e}^{-\frac{(\zeta-\mu_x)^2}{2\sigma_x^2}} \mathrm{d}\zeta \tag{2.2}$$

$$= \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{\theta-\mu_x}{\sqrt{2}\sigma_x}\right)\right) \tag{2.3}$$

where $\mathrm{erf}(\cdot)$ denotes the error function.

For the standard zero-mean normal distribution ($\mu_x = 0$, $\sigma_x^2 = 1$) the PDF/CDF are illustrated in the following

```
In [5]: plot_pdf_cdf(np.linspace(-5, 5, num=100), stats.norm)
```

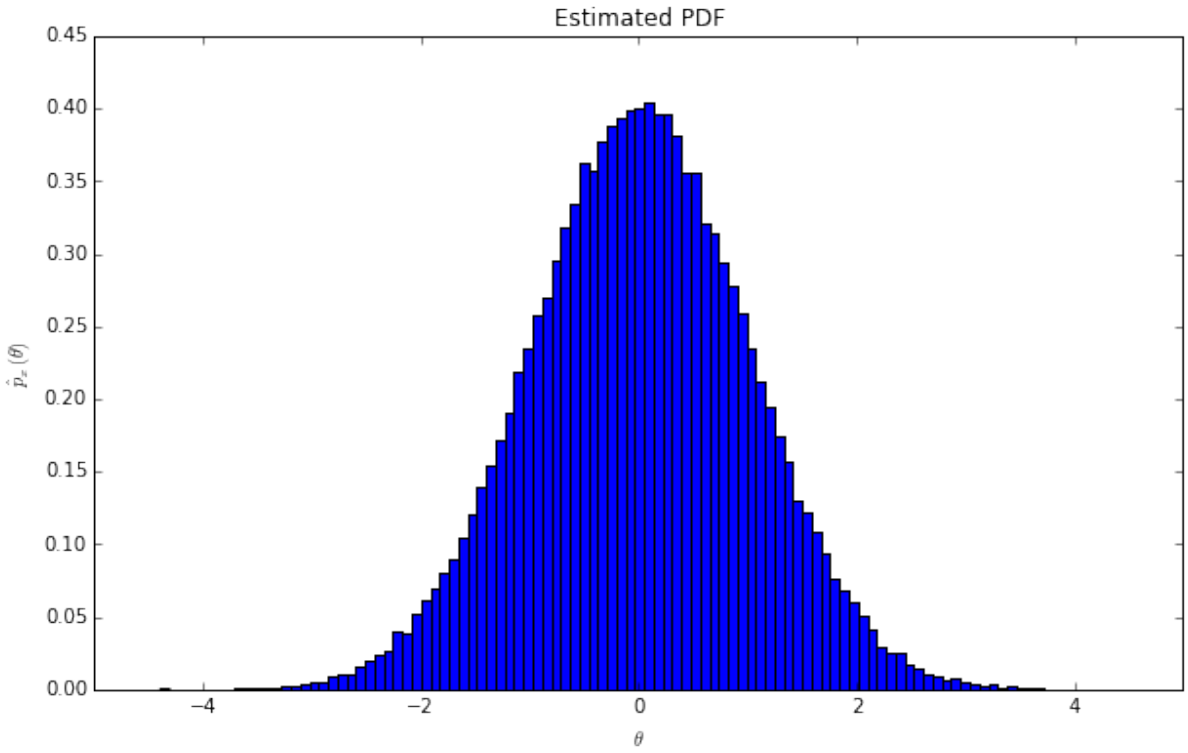


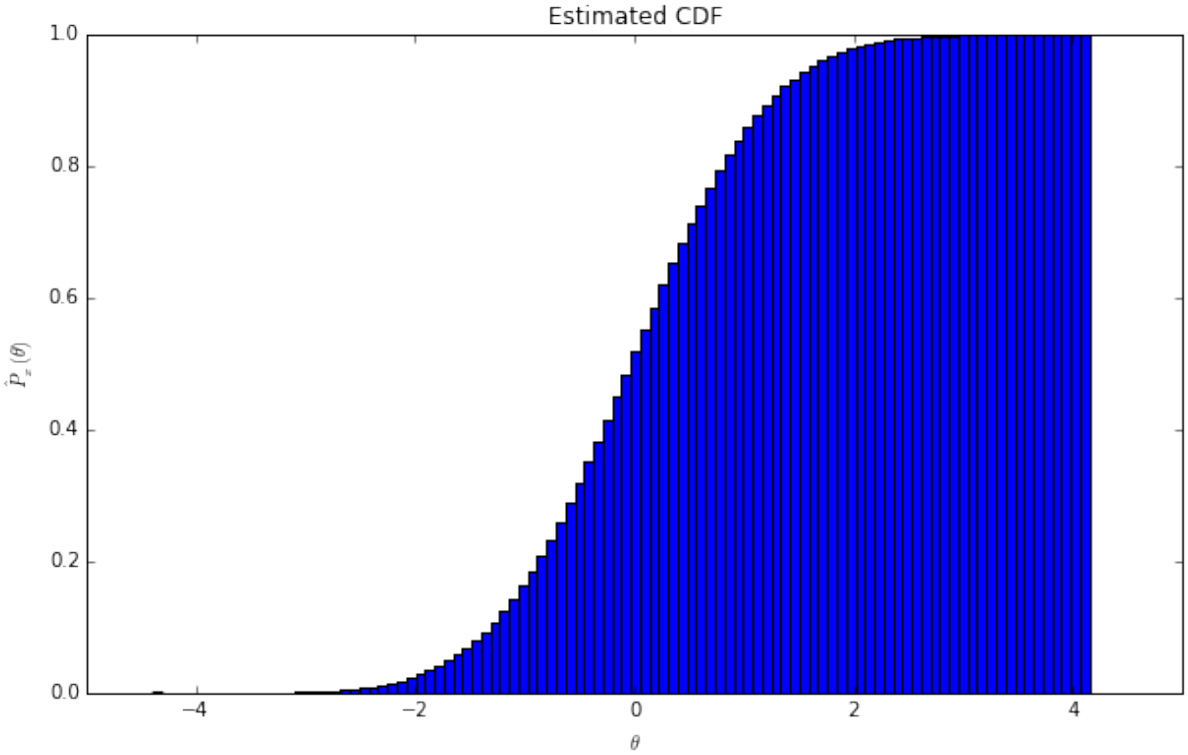
Example

For the standard zero-mean uniform distribution we get the following numerical results when drawing a large number of random samples

```
In [6]: compute_plot_pdf_cdf(stats.norm.rvs(size=100000), nbins=100)
```

```
Linear mean: 0.004293
Variance: 0.998803
```





Exercise

- How can the linear mean μ_x and the variance σ_x^2 be changed?
- Assume you want to model zero-mean measurement noise with a given power P. How do you have to chose the parameters of the normal distribution?

2.16.3 Laplace Distribution

Definition

The PDF of the Laplace distribution is given as

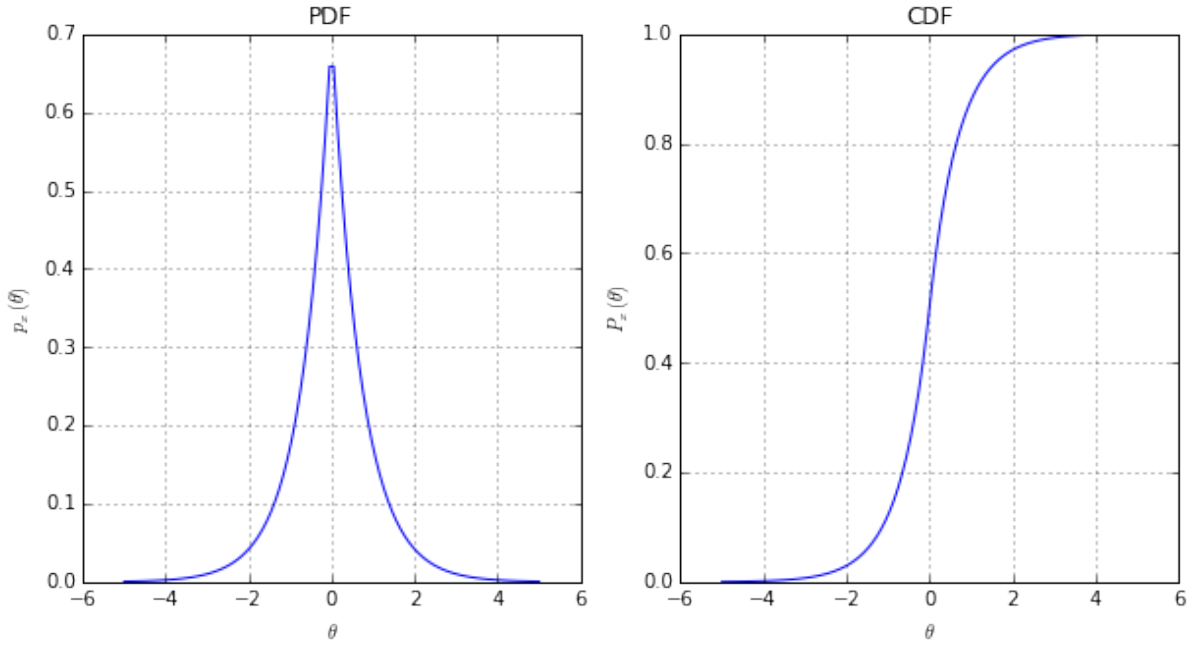
$$p_x(\theta) = \frac{1}{\sqrt{2}\sigma_x} \mathrm{e}^{-\sqrt{2}\frac{|\theta-\mu_x|}{\sigma_x}}$$

where μ_x and σ_x^2 denote the linear mean and variance, respectively. Laplace distributions are often used to model the PDF of a speech or music signal. The CDF can be derived by integration over θ

$$P_x(\theta) = \begin{cases} \frac{1}{2}\mathrm{e}^{\sqrt{2}\frac{\theta-\mu_x}{\sigma_x}} & \text{for } \theta \leq \mu_x \\ 1 - \frac{1}{2}\mathrm{e}^{-\sqrt{2}\frac{\theta-\mu_x}{\sigma_x}} & \text{for } \theta > \mu_x \end{cases}$$

For the zero-mean, unit-variance Laplace distribution the PDF/CDF is illustrated. Note the scale parameter in 'scipy.stats.laplace <http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.laplace.html#scipy.stats.laplace>'__ is related to the variance by $\lambda = \frac{\sigma_x}{\sqrt{2}}$

```
In [7]: plot_pdf_cdf(np.linspace(-5, 5, num=100), stats.laplace(scale=1/np.sqrt(2)))
```

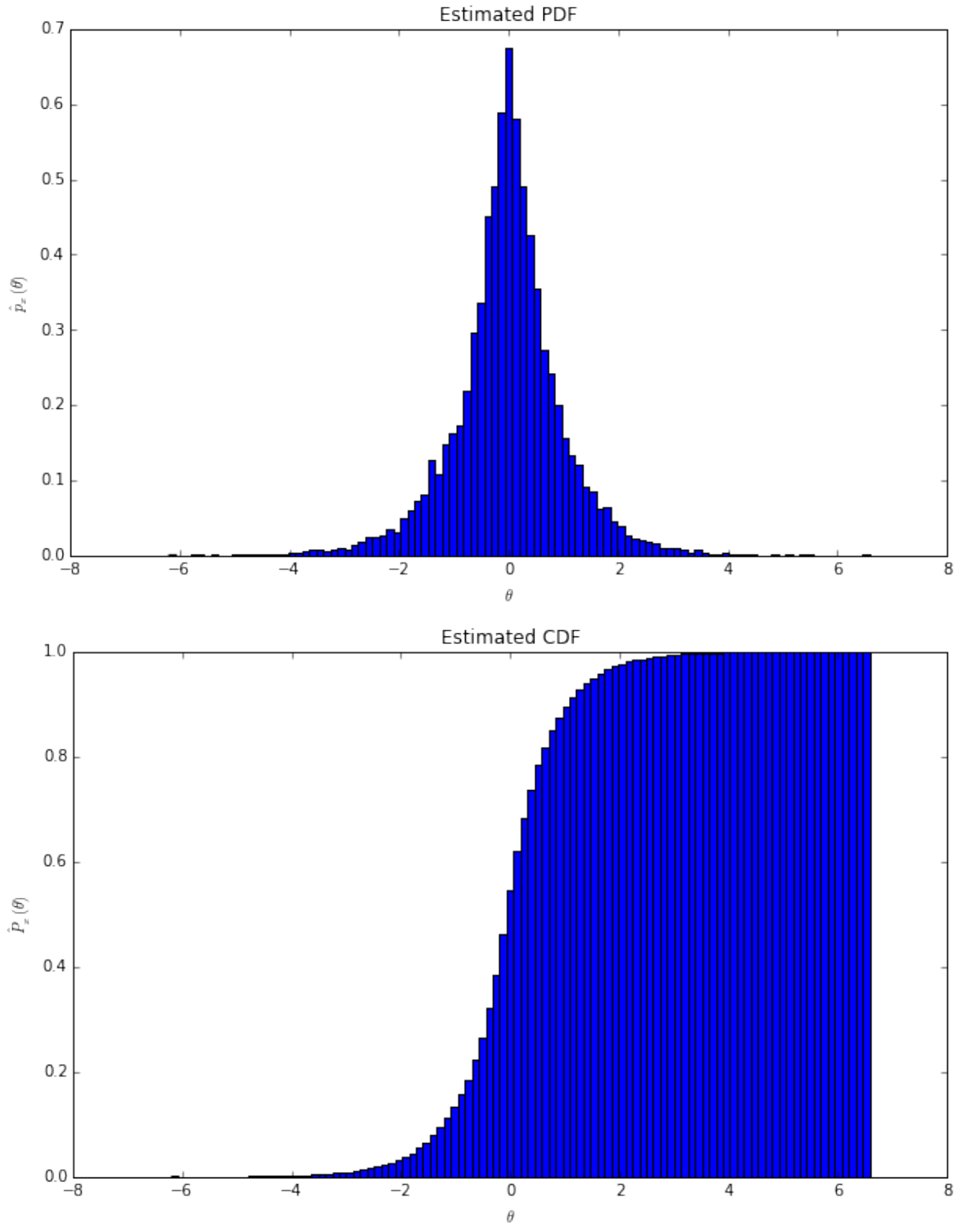


Example

For the standard zero-mean Laplace distribution we get the following numerical results when drawing a large numer of random samples

```
In [8]: compute_plot_pdf_cdf(stats.laplace(scale=1/np.sqrt(2)).rvs(size=10000), nbins=10
```

```
Linear mean: -0.015657
Variance: 1.004444
```



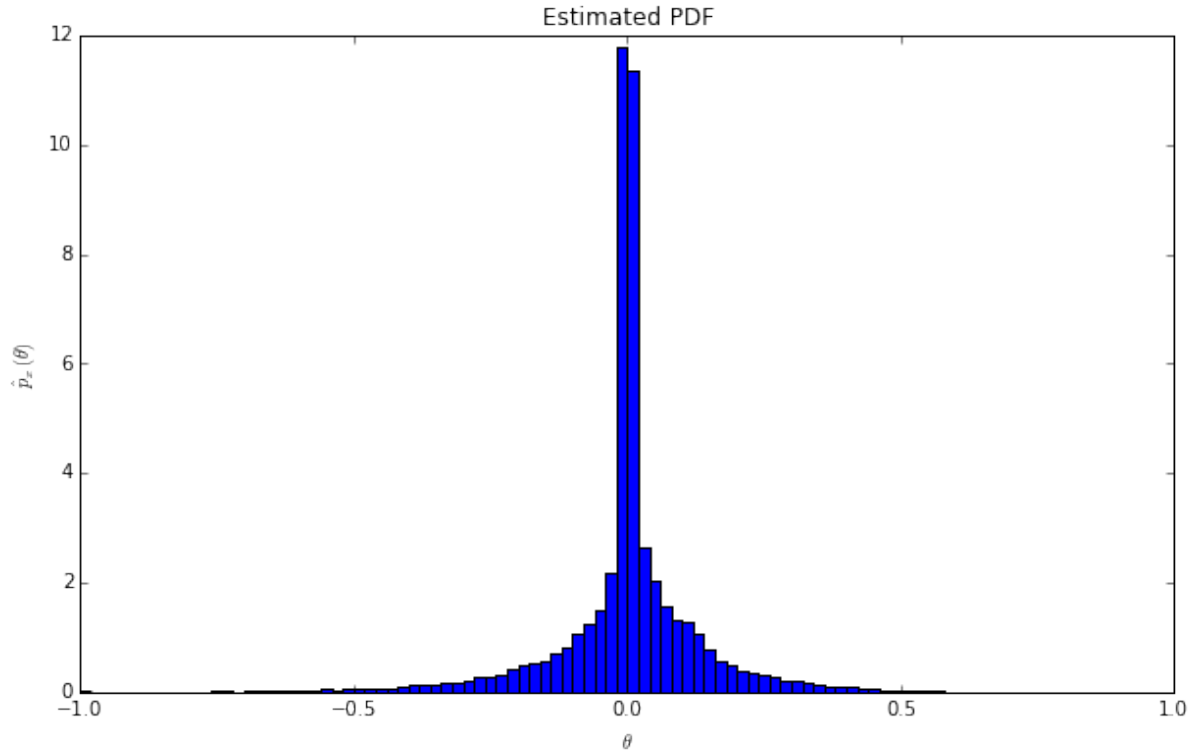
2.16.4 Amplitude Distribution of a Speech Signal

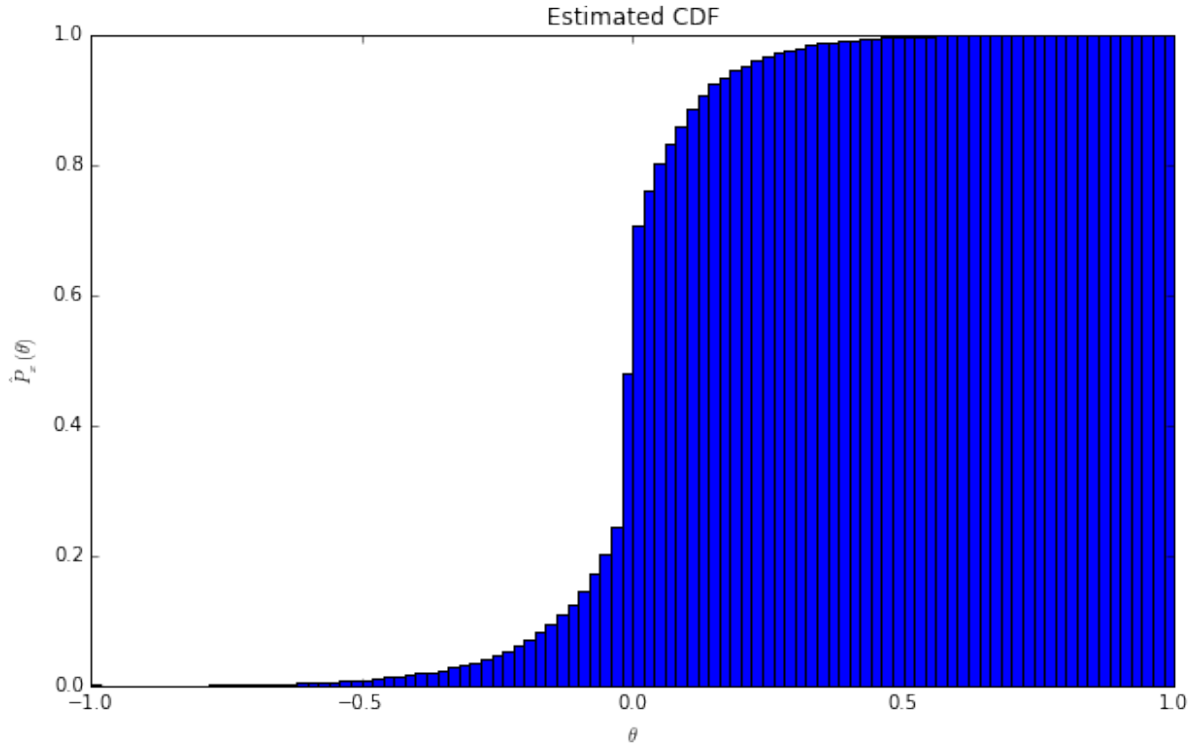
Lets take a look at the PDF/CDF of a speech signal in order to see if we can model it by one of the PDFs introduced above.

```
In [9]: from scipy.io import wavfile

        fs, x = wavfile.read('../data/speech_8k.wav')
```

```
        x = np.asarray(x, dtype=float)/2**15
        compute_plot_pdf_cdf(x, nbins=100)
```

```
Linear mean: -0.000067
Variance: 0.018548
```





Exercise

- Which analytic PDF/CDF can be used to model a speech signal?
- How would you chose the parameters of the distribution to fit the data?

2.17 White Noise

2.17.1 Definition

White noise is a random signal with a constant power spectral density (PSD). White noise draws its name from the analogy to white light. It refers typically to a model of random signals, like e.g. measurement noise. For a zero-mean random signal $x[k]$, its PSD reads

$$\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = N_0$$

where N_0 denotes the power per frequency. The auto-correlation function of white noise can be derived by inverse discrete-time Fourier transformation (DTFT) of the PSD

$$\varphi_{xx}[\kappa] = \mathcal{F}_*\{N_0\} = N_0\ \delta[\kappa]$$

Hence, neighboring samples k and $k+1$ are uncorrelated and have no statistical dependencies. The probability density function (PDF) of white noise is not necessarily normally distributed. Hence, it is necessary to additionally state the amplitude distribution when classifying a signal as white noise.

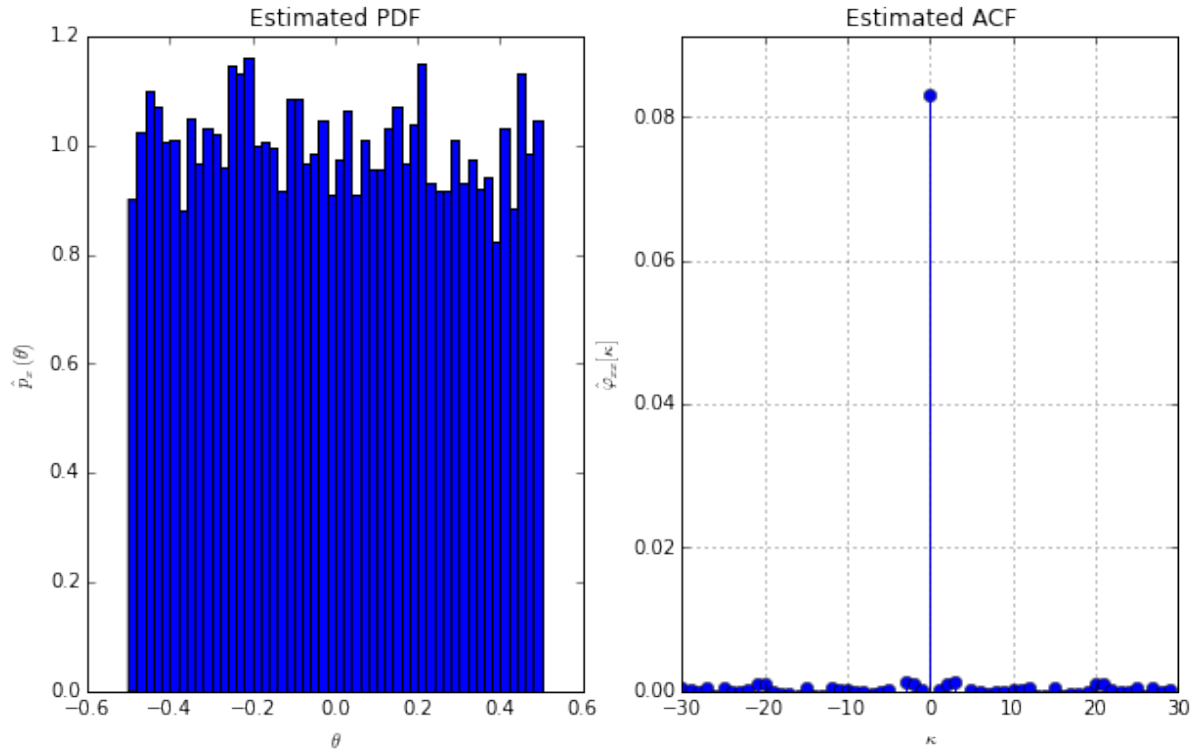
2.17.2 Example

Toolboxes for numerical mathematics like `Numpy` or `scipy.stats` provide functions to draw random uncorrelated samples from various PDFs. In order to evaluate this, a function is defined to compute and plot the PDF and CDF for a given random signal $x[k]$.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        def compute_plot_pdf_acf(x, nbins=50, acf_range=30):

            # compute and truncate ACF
            acf = 1/len(x) * np.correlate(x, x, mode='full')
            acf = acf[len(x)-acf_range-1:len(x)+acf_range-1]
            kappa = np.arange(-acf_range, acf_range)

            # plot PSD
            plt.figure(figsize = (10, 6))
            plt.subplot(121)
            plt.hist(x, nbins, normed=True)
            plt.title('Estimated PDF')
            plt.xlabel(r'$\theta$')
            plt.ylabel(r'$\hat{p}_x(\theta)$')

            # plot ACF
            plt.subplot(122)
            plt.stem(kappa, acf)
            plt.title('Estimated ACF')
            plt.ylabel(r'$\hat{\varphi}_{xx}[\kappa]$')
            plt.xlabel(r'$\kappa$')
            plt.axis([-acf_range, acf_range, 0, 1.1*max(acf)]);
            plt.grid()
```

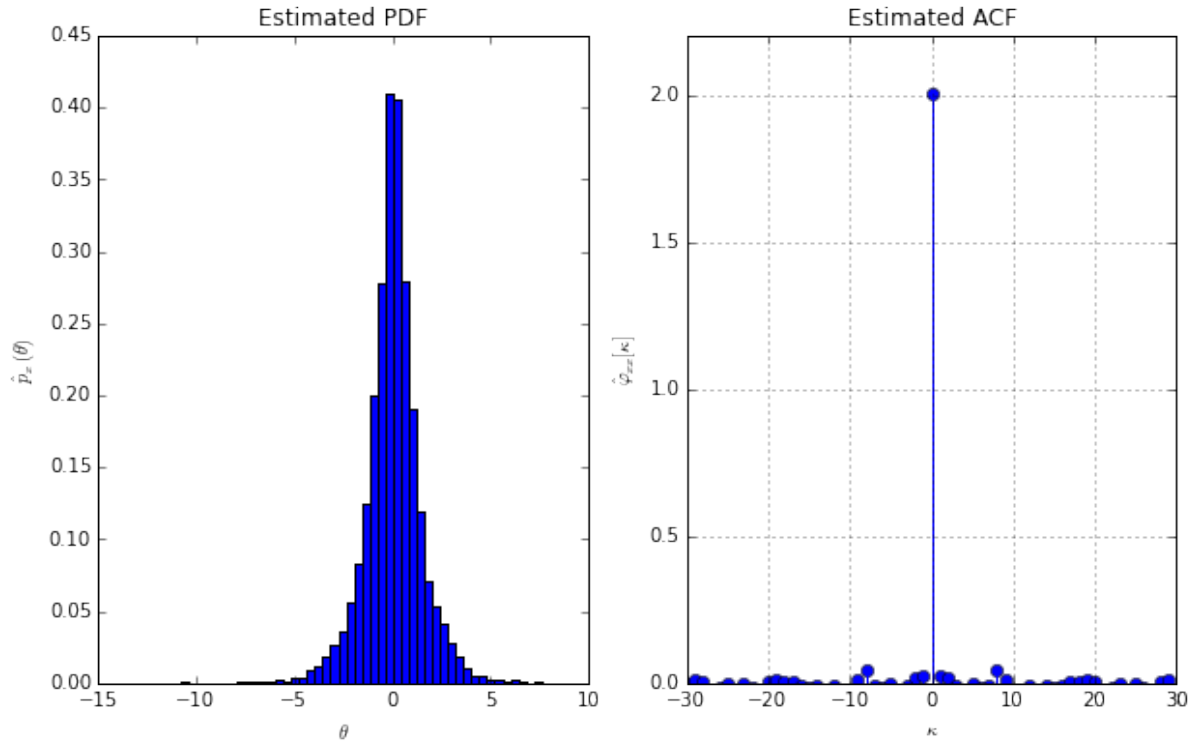
For samples drawn from a zero-mean uniform distribution the PDF and ACF are estimated as

```
In [2]: compute_plot_pdf_acf(np.random.uniform(size=10000)-1/2)
```



For samples drawn from a zero-mean Laplace distribution the PDF and ACF are estimated as

```
In [3]: compute_plot_pdf_acf(np.random.laplace(size=10000))
```



Exercise

- Do both random processes represent white noise?
- How does the ACF change if you lower the length `size` of the random signal. Why?

2.18 Superposition of Random Signals

The superposition of two random signals

$$y[k] = x[k] + n[k]$$

is a frequently applied operation in statistical signal processing. For instance to model the distortions of a measurement procedure or communication channel. We assume that the statistical properties of the real-valued signals $x[k]$ and $n[k]$ are known. We are interested in the statistical properties of $y[k]$, as well as the joint statistical properties between the signals and their superposition $y[k]$. For the following derivations it is assumed that $x[k]$ and $n[k]$ are drawn from weakly stationary real-valued random processes.

2.18.1 Cumulative Distribution and Probability Density Function

The cumulative distribution function (CDF) $P_y(\theta)$ of $y[k]$ is given by rewriting it in terms of the joint probability density function (PDF) $p_{xn}(\theta_x, \theta_n)$

$$P_y(\theta) = \Pr\{y[k] \leq \theta\} = \Pr\{(x[k] + n[k]) \leq \theta\} = \int_{-\infty}^{\infty} \int_{-\infty}^{\theta - \theta_n} p_{xn}(\theta_x, \theta_n) \, \mathrm{d}\theta_x \, \mathrm{d}\theta_n$$

The PDF is derived by introducing above result into *its definition* as

$$p_y(\theta) = \frac{\mathrm{d}P_y(\theta)}{\mathrm{d}\theta} = \int_{-\infty}^{\infty} p_{xn}(\theta - \theta_n, \theta_n) \, \mathrm{d}\theta_n$$

since the inner integral on the right hand side of $P_y(\theta)$ can be interpreted as the inverse operation to the derivation with respect to θ.

An important special case is that $x[k]$ and $n[k]$ are uncorrelated. Under this assumption the joint PDF $p_{xn}(\theta_x, \theta_n)$ can be written as $p_{xn}(\theta_x, \theta_n) = p_x(\theta_x) \cdot p_n(\theta_n)$. For $p_y(\theta)$ follows

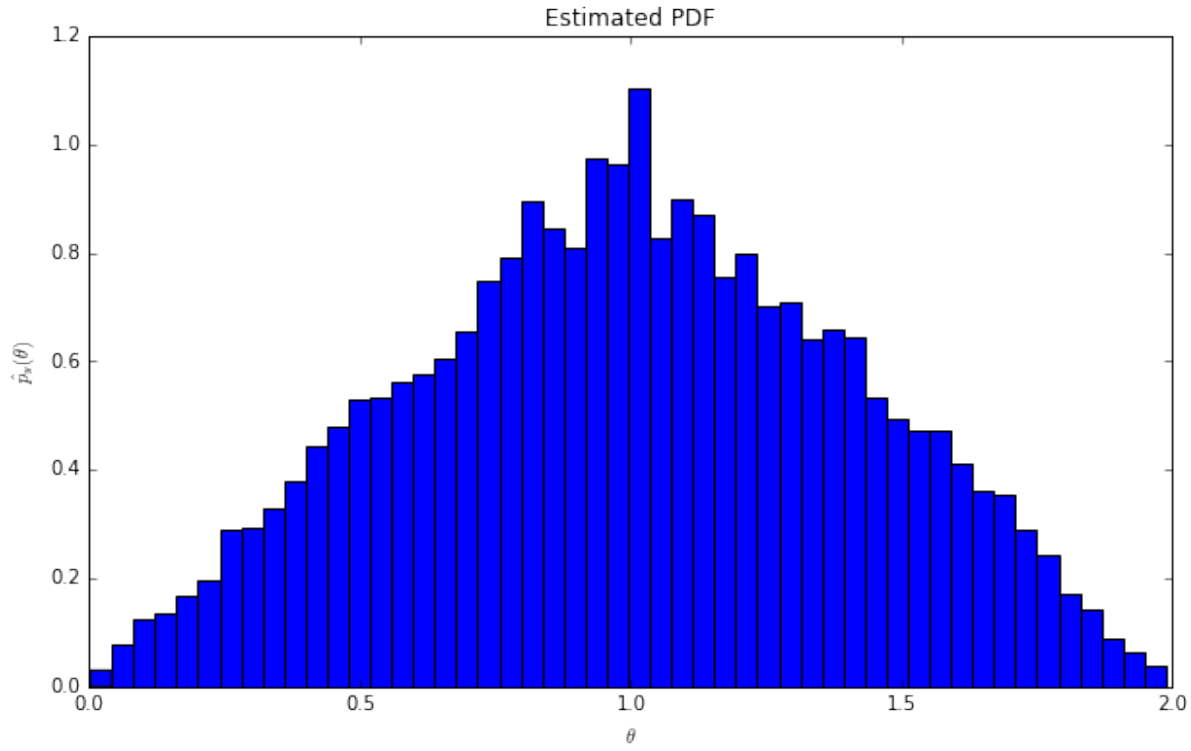
$$p_y(\theta) = p_x(\theta) * p_n(\theta)$$

Hence, the PDF of the superposition is given by the convolution of the PDFs of both signals.

Example

The following example estimates the PDF of a superposition of two uncorrelated signals drawn from *uniformly distributed* white noise sources with $a = 0$ and $b = 1$.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        N = 10000  # length of random signals

        # generate random signals
        x = np.random.uniform(size=N)
        n = np.random.uniform(size=N)
        y = x + n

        # plot estimated pdf
        plt.figure(figsize = (10, 6))
        plt.hist(y, 50, normed=True)
        plt.title('Estimated PDF')
        plt.xlabel(r'$\theta$')
        plt.ylabel(r'$\hat{p}_y(\theta)$');
```



Exercise

- Check the result of the numerical simulation by calculating the theoretical PDF of $y[k]$

2.18.2 Linear Mean

The linear mean μ_y of the superposition is derived by introducing $y[k] = x[k] + n[k]$ into the *definition of the linear mean* and exploiting the *linearity of the expectation operator* as

$$\mu_y[k] = E\{x[k] + n[k]\} = \mu_x[k] + \mu_n[k]$$

The linear mean of the superposition of two random signals is the superposition of its linear means.

2.18.3 Auto-Correlation Function and Power Spectral Density

The ACF is computed in the same manner as above by inserting the superposition into its *definition* and rearranging terms

$$\begin{aligned} \varphi_{yy}[\kappa] &= E\{y[k] \cdot y[k-\kappa]\} && (2.4)\\ &= E\{(x[k] + n[k]) \cdot (x[k-\kappa] + n[k-\kappa])\} && (2.5)\\ &= \varphi_{xx}[\kappa] + \varphi_{xn}[\kappa] + \varphi_{nx}[\kappa] + \varphi_{nn}[\kappa] && (2.6) \end{aligned}$$

The ACF of the superposition of two random signals is given as the superposition of all auto- and cross-correlation functions (CCFs) of the two random signals. The power spectral density (PSD) is derived by discrete-time Fourier transformation (DTFT) of the ACF

$$\Phi_{yy}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) + \Phi_{xn}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) + \Phi_{nx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) + \Phi_{nn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

This can be simplified further by exploiting the symmetry property of the CCFs $\varphi_{xn}[\kappa] = \varphi_{nx}[-\kappa]$ and the DTFT for real-valued signals as

$$\Phi_{yy}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) + 2\,\Re\{\Phi_{xn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\} + \Phi_{nn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

where $\Re\{\cdot\}$ denotes the real part of its argument.

2.18.4 Cross-Correlation Function and Cross Power Spectral Density

The CCF $\varphi_{ny}[\kappa]$ between the random signal $n[k]$ and the superposition $y[k]$ is derived again by introducing the superposition into the *definition of the CCF*

$$\varphi_{ny}[\kappa] = E\{n[k] \cdot (x[k-\kappa] + n[k-\kappa])\} = \varphi_{nx}[\kappa] + \varphi_{nn}[\kappa]$$

It is given as the superposition of the CCF between the two random signals and the ACF of $n[k]$. The cross PSD is derived by applying a DTFT to $\varphi_{ny}[\kappa]$

$$\Phi_{ny}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{nx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) + \Phi_{nn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

The CCF $\varphi_{xy}[\kappa]$ and cross PSD $\Phi_{xy}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ can be derived by exchanging the signals $n[k]$ and $x[k]$

$$\varphi_{xy}[\kappa] = E\{x[k] \cdot (x[k-\kappa] + n[k-\kappa])\} = \varphi_{xx}[\kappa] + \varphi_{xn}[\kappa]$$

and

$$\Phi_{xy}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) + \Phi_{xn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

2.18.5 Additive White Gaussian Noise

In order to model the effect of distortions it is often assumed that a random signal $x[k]$ is distorted by additive normal distributed white noise resulting in the observed signal $y[k] = x[k] + n[k]$. It is furthermore assumed that the noise $n[k]$ is uncorrelated to the signal $x[k]$. This model is known as additive white Gaussian noise (AWGN) model.

For zero-mean random processes it follows $\varphi_{xn}[\kappa] = \varphi_{nx}[\kappa] = 0$ and $\varphi_{nn}[\kappa] = N_0 \cdot \delta[\kappa]$ from the properties of the AWGN model. Introducing this into the findings for additive random signals yields the following relations for the AWGN model

$$\varphi_{yy}[\kappa] = \varphi_{xx}[\kappa] + N_0 \cdot \delta[\kappa] \tag{2.7}$$
$$\varphi_{ny}[\kappa] = N_0 \cdot \delta[\kappa] \tag{2.8}$$
$$\varphi_{xy}[\kappa] = \varphi_{xx}[\kappa] \tag{2.9}$$

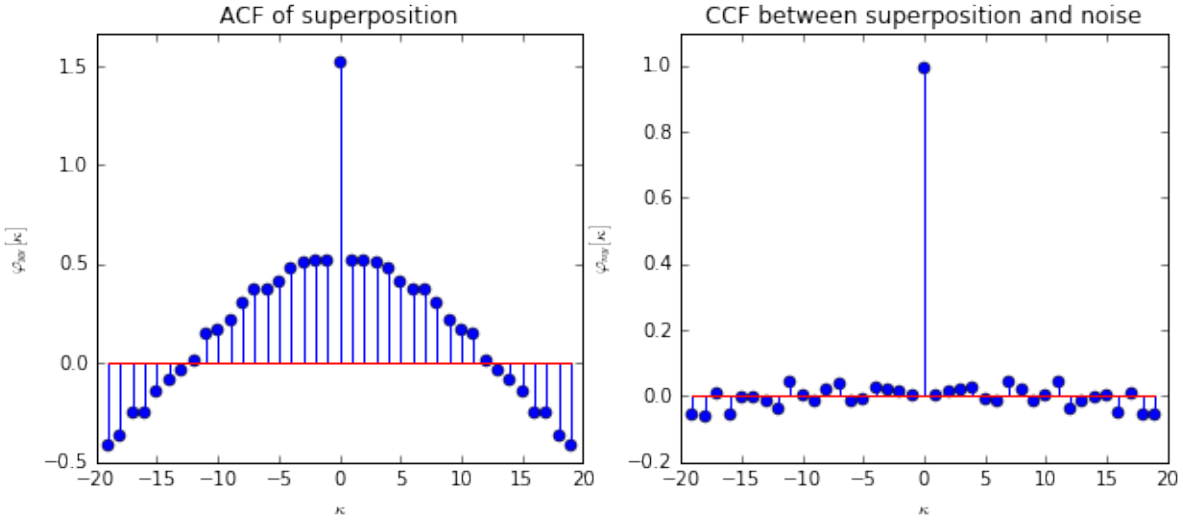
The PSDs are given as the DTFT of these results. The AWGN model is frequently applied in communications as well as measurement of physical quantities to cope for background, sensor and amplifier noise.

Example

For the following numerical example, the disturbance of a harmonic signal $x[k] = \cos[\Omega_0 k]$ by unit variance AWGN is considered.

```
In [2]: N = 1024  # length of signals
        K = 20  # maximum lag for ACF/CCF

        # generate signals
        x = np.cos(20*2*np.pi/N*np.arange(N))
        n = np.random.normal(size=N)
        # superposition of signals
        y = x + n

        # compute and truncate ACF of superposition
        acf = 1/N * np.correlate(y, y, mode='full')
        acf = acf[(len(x)-1)-(K-1):(len(x)-1)+K]
        # compute and truncate CCF of superposition and noise
        ccf = 1/N * np.correlate(n, y, mode='full')
        ccf = ccf[(len(x)-1)-(K-1):(len(x)-1)+K]
```

```
# plot results
kappa = np.arange(-(K-1), K)
plt.figure(figsize=(10, 4))

plt.subplot(121)
plt.stem(kappa, acf)
plt.title('ACF of superposition')
plt.xlabel(r'$\kappa$')
plt.ylabel(r'$\varphi_{yy}[\kappa]$')
plt.axis([-K, K, -.5, 1.1*np.max(acf)])

plt.subplot(122)
plt.stem(kappa, ccf)
plt.title('CCF between superposition and noise')
plt.xlabel(r'$\kappa$')
plt.ylabel(r'$\varphi_{ny}[\kappa]$')
plt.axis([-K, K, -.2, 1.1]);
```



Exercise

- Derive the theoretic result for $\varphi_{xx}[\kappa]$
- Based in this, can you explain the results of the numerical simulation?

CHAPTER 3

Random Signals and LTI Systems

3.1 Introduction

The response of a system $y[k] = \mathcal{H}\{x[k]\}$ to a random input signal $x[k]$ is the foundation of statistical signal processing. In the following we limit ourselves to linear-time invariant (LTI) systems.

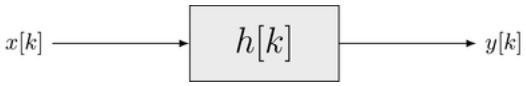


Fig. 3.1: Illustration: LTI-System

Let's assume that the statistical properties of the input signal $x[k]$ are known, for instance its first and second order ensemble averages. Let's further assume that the impulse response $h[k]$ or the transfer function $H(e^{\mathrm{j}\,\Omega})$ of the LTI system is given. We are looking for the statistical properties of the output signal $y[k]$ and the joint properties between the input $x[k]$ and output $y[k]$ signal.

3.2 Stationarity and Ergodicity

The question arises if the output signal $y[k]$ of an LTI system is (weakly) stationary or stationary and ergodic for an input signal $x[k]$ exhibiting the same properties.

Let's assume that the input signal $x[k]$ originates from a stationary random process. According to the *definition of stationarity* the following relation must hold

$$E\{f(x[k_1], x[k_2], \dots)\} = E\{f(x[k_1 + \Delta], x[k_2 + \Delta], \dots)\}$$

where $\Delta \in \mathbb{Z}$ denotes an arbitrary (temporal) shift. The condition for time-invariance of a system reads

$$y[k + \Delta] = \mathcal{H}\{x[k + \Delta]\}$$

By introducing this into the right hand side of the definition of stationarity for the output signal $y[k]$ and recalling that $y[k] = \mathcal{H}\{x[k]\}$ we can show that

$$E\{g(y[k_1], y[k_2], \dots)\} = E\{g(y[k_1 + \Delta], y[k_2 + \Delta], \dots)\}$$

where $g(\cdot)$ denotes an arbitrary mapping function that may differ from $f(\cdot)$. From the equation above, it can be concluded that the output signal of an LTI system for a (weakly) stationary input signal is also (weakly) stationary. The same reasoning can also be applied to a *(weakly) ergodic* input signal.

Summarizing, for an input signal $x[k]$ that is

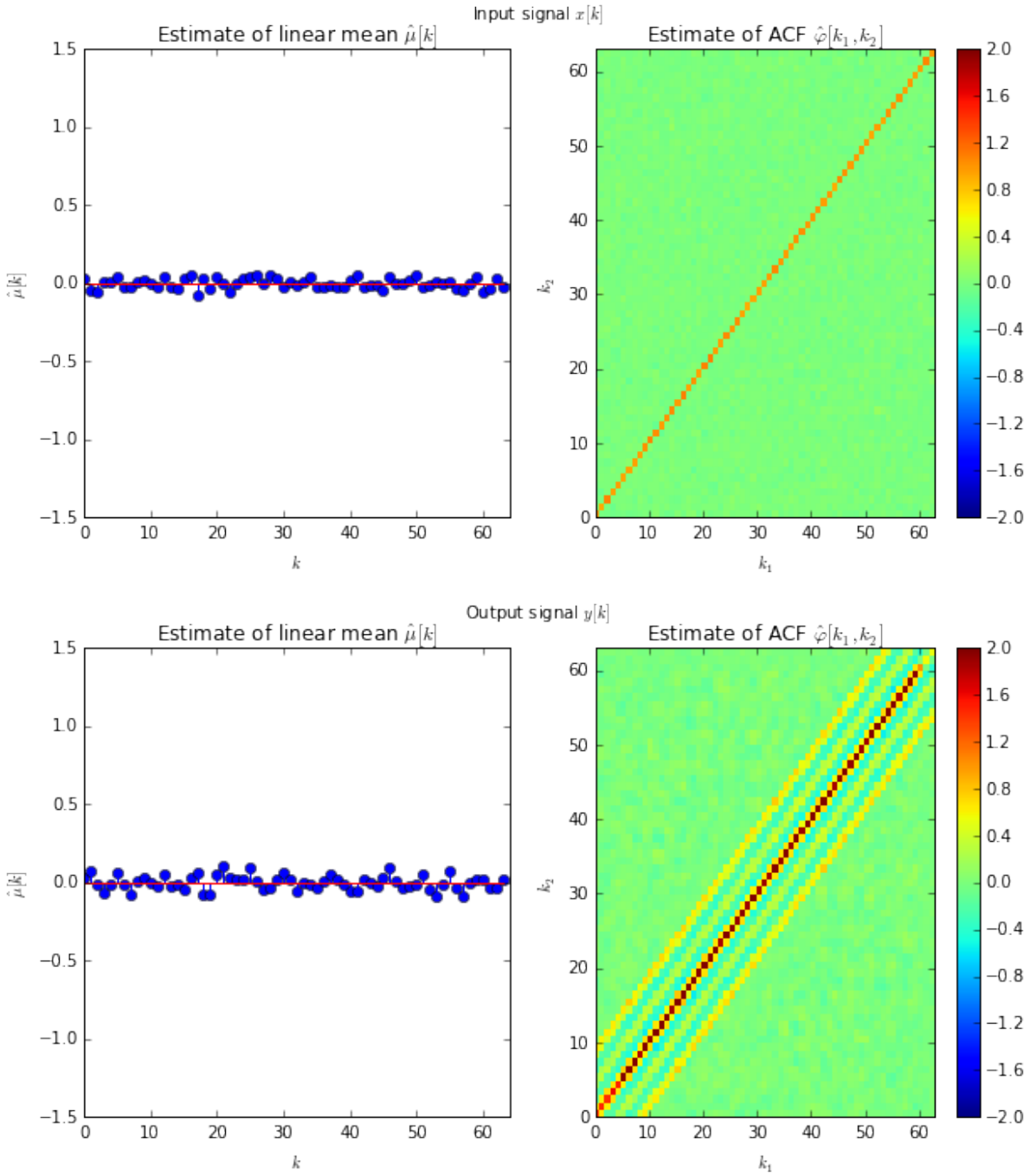
- (weakly) stationary, the output signal $y[k]$ is (weakly) stationary and the in- and output is jointly (weakly) stationary
- (weakly) ergodic, the output signal $y[k]$ is (weakly) ergodic and the in- and output is jointly (weakly) ergodic

This implies for instance, that for a weakly stationary input signal measures like the auto-correlation function (ACF) can also be applied to the output signal.

3.2.1 Example

The following example computes and plots estimates of the linear mean $\mu[k]$ and auto-correlation function (ACF) $\varphi[k_1, k_2]$ for the in- and output of an LTI system. The input $x[k]$ is drawn from a normal distributed white noise process.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        L = 64  # number of random samples
        N = 1000  # number of sample functions

        # generate input signal (white Gaussian noise)
        x = np.random.normal(size=(N, L))
        # generate output signal
        h = 2*np.fft.irfft([1,1,1,0,0,0])
        y = np.asarray([np.convolve(x[n,:], h, mode='same') for n in range(N)])

        # compute and plot results
        def compute_plot_results(x):

            # estimate linear mean by ensemble average
            mu = 1/N * np.sum(x, 0)
            # estimate the auto-correlation function
            acf = np.zeros((L, L))
            for n in range(L):
                for m in range(L):
                    acf[n, m] = 1/N * np.sum(x[:, n]*x[:, m], 0)

            plt.subplot(121)
            plt.stem(mu)
            plt.title(r'Estimate of linear mean $\hat{\mu}[k]$')
            plt.xlabel(r'$k$')
            plt.ylabel(r'$\hat{\mu}[k]$')
            plt.axis([0, L, -1.5, 1.5])

            plt.subplot(122)
            plt.pcolor(np.arange(L), np.arange(L), acf, vmin=-2, vmax=2)
            plt.title(r'Estimate of ACF $\hat{\varphi}[k_1, k_2]$')
            plt.xlabel(r'$k_1$')
            plt.ylabel(r'$k_2$')
            plt.colorbar()
            plt.autoscale(tight=True)


        plt.figure(figsize = (10, 5))
        plt.gcf().suptitle(r'Input signal $x[k]$')
        compute_plot_results(x)

        plt.figure(figsize = (10, 5))
        plt.gcf().suptitle(r'Output signal $y[k]$')
        compute_plot_results(y)
```



Exercise

- Is the in- and output signal weakly stationary?
- Can the output signal $y[k]$ assumed to be white noise?

3.3 Linear Mean

In the following we aim at finding a relation between the linear mean $\mu_x[k]$ of the input signal $x[k]$ and the linear mean $\mu_y[k]$ of the output signal $y[k] = \mathcal{H}\{x[k]\}$ of an LTI system.

3.3.1 Non-Stationary Process

Let's first impose no restrictions in terms of stationarity to the input signal. The *linear mean* of the output signal is then given as

$$\mu_y[k] = E\{y[k]\} = E\{x[k] * h[k]\}$$

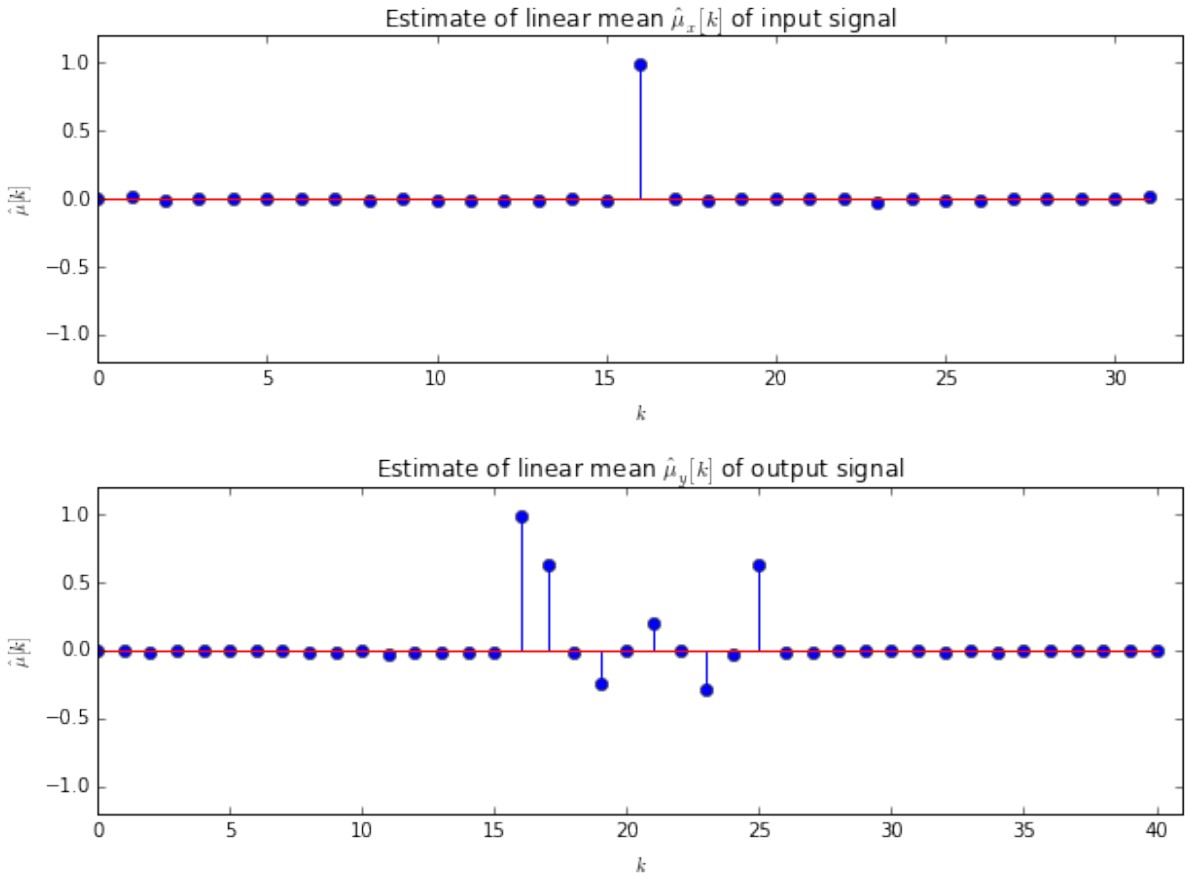
where $h[k]$ denotes the impulse response of the system. Since the convolution and the ensemble average are linear operations, and $h[k]$ is a deterministic signal this can be rewritten as

$$\mu_y[k] = \mu_x[k] * h[k]$$

Hence, the linear mean of the output signal $\mu_y[k]$ is given as the convolution of the linear mean of the input signal $\mu_x[k]$ with the impulse response $h[k]$ of the system.

Example

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        L = 32  # number of random samples
        N = 10000  # number of sample functions

        # generate input signal (white Gaussian noise)
        x = np.random.normal(size=(N, L))
        x[:, L//2] += 1
        # generate output signal
        h = 2*np.fft.irfft([1,1,1,0,0,0])
        y = np.asarray([np.convolve(x[n,:], h, mode='full') for n in range(N)])

        # compute and plot results
        def compute_plot_results(x):

            # estimate linear mean by ensemble average
            mu = 1/N * np.sum(x, 0)
            # plot linear mean
            plt.stem(mu)
            plt.xlabel(r'$k$')
            plt.ylabel(r'$\hat{\mu}[k]$')
            plt.axis([0, x.shape[1], -1.2, 1.2])

        plt.figure(figsize = (10, 3))
        plt.title(r'Estimate of linear mean $\hat{\mu}_x[k]$ of input signal')
        compute_plot_results(x)

        plt.figure(figsize = (10, 3))
        plt.title(r'Estimate of linear mean $\hat{\mu}_y[k]$ of output signal')
        compute_plot_results(y)
```



Exercise

- Can you estimate the impulse response $h[k]$ of the system from above plots of $\hat{\mu}_x[k]$ and $\hat{\mu}_y[k]$?
- You can check your results by plotting the impulse response $h[k]$, for instance with the command `plt.stem(h)`.

3.3.2 Stationary Process

For a (weakly) stationary process, the linear mean of the input signal $\mu_x[k] = \mu_x$ does not depend on the time index k. For a (weakly) stationary input signal, also the output signal of the system is (weakly) stationary. Using the result for the non-stationary case above yields

$$\mu_y = \mu_x * h[k] = \mu_x \cdot H(\mathrm{e}^{\,\mathrm{j}\,\Omega})\big|_{\Omega=0}$$

where $H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathcal{F}_*\{h[k]\}$ denotes the discrete time Fourier transformation (DTFT) of the impulse response. Hence, the linear mean of a (weakly) stationary input signal is weighted by the transmission characteristics for the constant (i.e. DC, $\Omega = 0$) component of the LTI system. This implies that the output signal to a zero-mean $\mu_x = 0$ input signal is also zero-mean $\mu_y = 0$.

3.4 Auto-Correlation Function

The auto-correlation function (ACF) $\varphi_{yy}[\kappa]$ of the output signal of an LTI system $y[k] = \mathcal{H}\{x[k]\}$ is derived. It is assumed that the input signal is a weakly stationary real-valued random process and that the LTI system has a real-valued impulse repsonse $h[k] \in \mathbb{R}$.

Introducing the output relation $y[k] = h[k] * x[k]$ of an LTI system into the definition of the auto-correlation

function and rearranging terms yields

$$\varphi_{yy}[\kappa] = E\{y[k+\kappa] \cdot y[k]\} \tag{3.1}$$

$$= E\left\{ \sum_{\mu=-\infty}^{\infty} h[\mu]\, x[k+\kappa-\mu] \cdot \sum_{\nu=-\infty}^{\infty} h[\nu]\, x[k-\nu] \right\} \tag{3.2}$$

$$= \underbrace{h[\kappa] * h[-\kappa]}_{\varphi_{hh}[\kappa]} * \varphi_{xx}[\kappa] \tag{3.3}$$

where the deterministic function $\varphi_{hh}[\kappa]$ is frequently termed as *filter ACF*. This is related to the *link between ACF and convolution*. The result above is known as the *Wiener-Lee theorem*. It states that the ACF of the output $\varphi_{yy}[\kappa]$ of an LTI system is given by the convolution of the input signal's ACF $\varphi_{xx}[\kappa]$ with the filter ACF $\varphi_{hh}[\kappa]$.

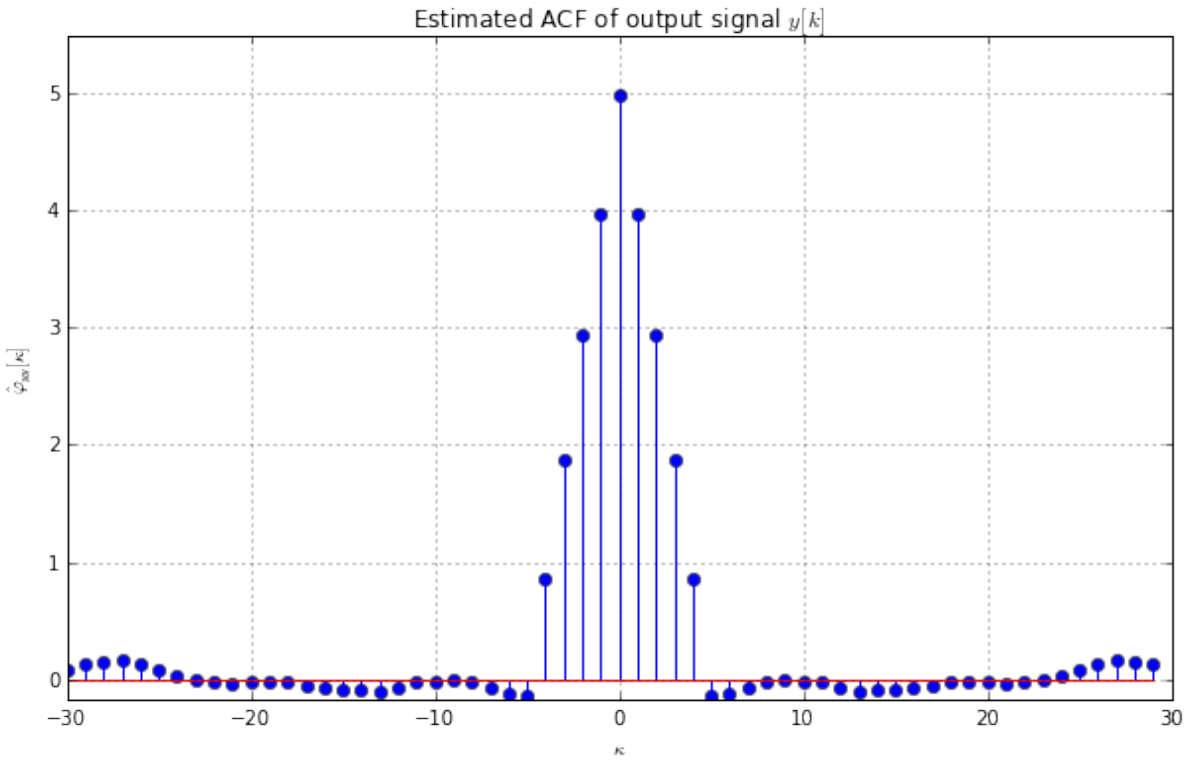
3.4.1 Example

Let's assume that the input signal $x[k]$ of an LTI system with impulse response $h[k] = \text{rect}_N[k]$ is normal distributed white noise. Hence, $\varphi_{xx}[\kappa] = N_0\, \delta[\kappa]$ can be introduced into the Wiener-Lee theorem yielding

$$\varphi_{yy}[\kappa] = N_0 \cdot (\text{rect}_N[\kappa] * \text{rect}_N[-\kappa])$$

The example is evaluated numerically for $N_0 = 1$ and $N = 5$ below.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        L = 10000  # number of samples
        K = 30  # limit for lags in ACF

        # generate input signal (white Gaussian noise)
        x = np.random.normal(size=L)
        # compute system response
        y = np.convolve(x, [1, 1, 1, 1, 1], mode='full')

        # compute and truncate ACF
        acf = 1/len(y) * np.correlate(y, y, mode='full')
        acf = acf[len(y)-K-1:len(y)+K-1]
        kappa = np.arange(-K, K)

        # plot ACF
        plt.figure(figsize = (10, 6))
        plt.stem(kappa, acf)
        plt.title('Estimated ACF of output signal $y[k]$')
        plt.ylabel(r'$\hat{\varphi}_{yy}[\kappa]$')
        plt.xlabel(r'$\kappa$')
        plt.axis([-K, K, 1.2*min(acf), 1.1*max(acf)]);
        plt.grid()
```



Exercise

- Why is the estimated ACF $\hat{\varphi}_{yy}[\kappa]$ of the output signal not exactly equal to its theoretic result $\varphi_{yy}[\kappa]$ given above?
- Change the number of samples `L` and rerun the cell. What changes?

3.5 Cross-Correlation Function

The cross-correlation functions (CCFs) $\varphi_{xy}[\kappa]$ and $\varphi_{yx}[\kappa]$ between the in- and output signal of an LTI system $y[k] = \mathcal{H}\{x[k]\}$ are derived. As for the ACF it is assumed that the input signal originates from a weakly stationary real-valued random process and that the LTI system's impulse response is real-valued, i.e. $h[k] \in \mathbb{R}$.

Introducing the convolution into the definition of the CCF and rearranging the terms yields

$$\begin{aligned}
\varphi_{xy}[\kappa] &= E\{x[k+\kappa] \cdot y[k]\} && (3.4)\\
&= E\left\{ x[k+\kappa] \cdot \sum_{\mu=-\infty}^{\infty} h[\mu]\, x[k-\mu] \right\} && (3.5)\\
&= \sum_{\mu=-\infty}^{\infty} h[\mu] \cdot E\{x[k+\kappa] \cdot x[k-\mu]\} && (3.6)\\
&= h[-\kappa] * \varphi_{xx}[\kappa] && (3.7)
\end{aligned}$$

The CCF $\varphi_{xy}[\kappa]$ between in- and output is given as the time-reversed impulse response of the system convolved with the ACF of the input signal.

The same calculus applied to the CCF between out- and input results in

$$\varphi_{yx}[\kappa] = h[\kappa] * \varphi_{xx}[\kappa]$$

Hence, the CCF $\varphi_{yx}[\kappa]$ between out- and input is given as the impulse response of the system convolved with the ACF of the input signal.

3.6 System Identification by Cross-Correlation

The CCFs of an LTI system play an important role in the measurement of the impulse response $h[k]$ of an unknown system. This is illustrated in the following.

Let's assume that the input signal $x[k]$ of the unknown LTI system is white noise. The ACF of the input signal is then given as $\varphi_{xx}[\kappa] = N_0 \cdot \delta[\kappa]$. According to the relation derived above, the CCF between out- and input for this special choice of input signal gets

$$\varphi_{yx}[\kappa] = N_0 \cdot h[\kappa]$$

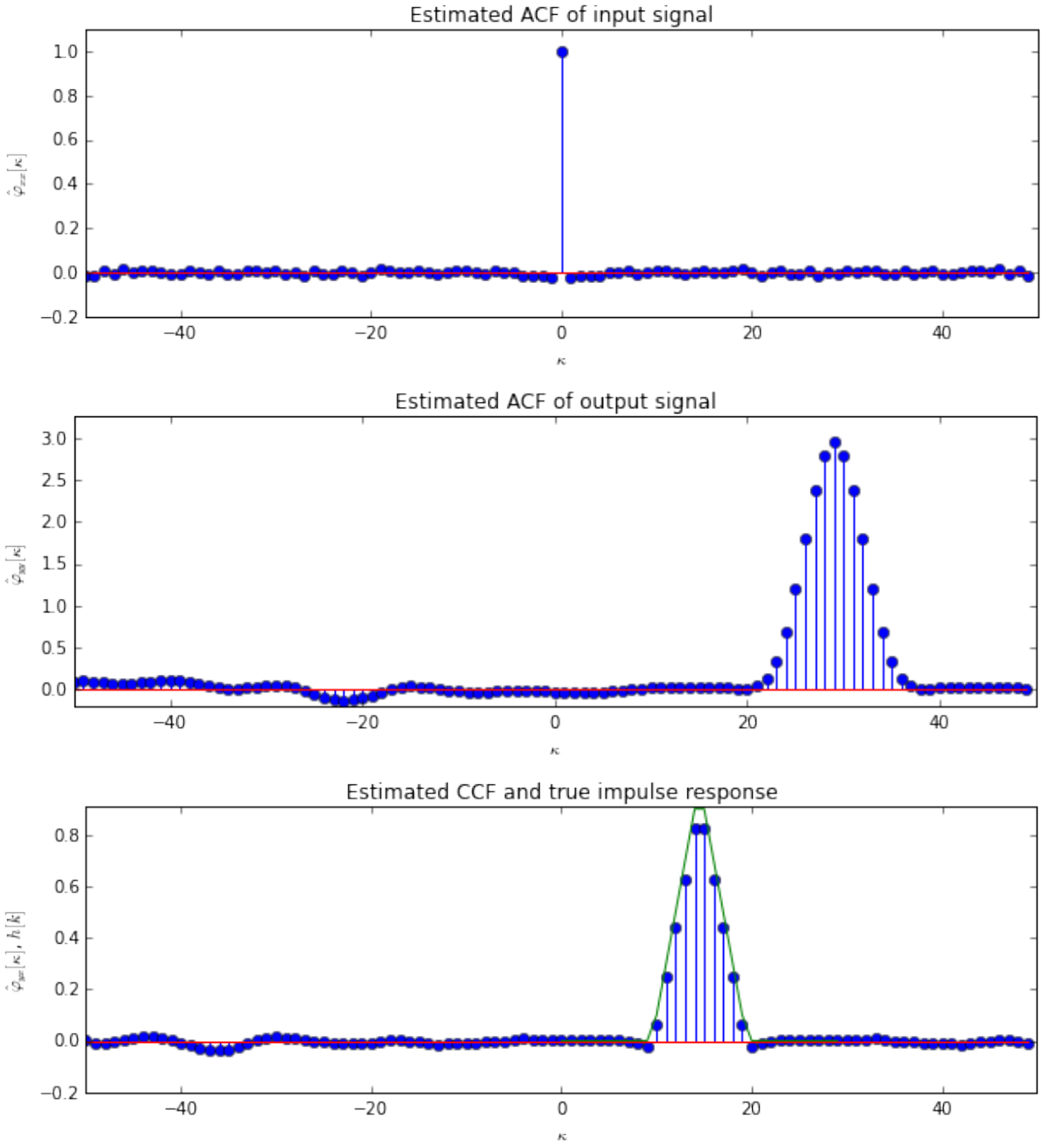
Hence, the impulse response is derived by computing the CCF between out- and input, for white noise as input signal $x[k]$.

3.6.1 Example

The application of the CCF for system identification is illustrated in the following

```
In [2]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig


        N = 10000  # number of samples for input signal
        K = 50  # limit for lags in ACF

        # generate input signal
        x = np.random.normal(size=N)  # normally distributed (zero-mean, unit-variance)
        # impulse response of the system
        h = np.concatenate((np.zeros(10), sig.triang(10), np.zeros(10)))
        # output signal by convolution
        y = np.convolve(h, x, mode='full')

        # compute correlation functions
        acfx = 1/len(x) * np.correlate(x, x, mode='full')
        acfy = 1/len(y) * np.correlate(y, y, mode='full')
        ccfyx = 1/len(y) * np.correlate(y, x, mode='full')

        def plot_correlation_function(cf):
            cf = cf[N-K-1:N+K-1]
            kappa = np.arange(-len(cf)//2,len(cf)//2)
            plt.stem(kappa, cf)
            plt.xlabel(r'$\kappa$')
            plt.axis([-K, K, -0.2, 1.1*max(cf)])

        # plot ACFs and CCF
        plt.rc('figure', figsize=(10, 3))
        plt.figure()
        plot_correlation_function(acfx)
        plt.title('Estimated ACF of input signal')
        plt.ylabel(r'$\hat{\varphi}_{xx}[\kappa]$')


        plt.figure()
        plot_correlation_function(acfy)
        plt.title('Estimated ACF of output signal')
        plt.ylabel(r'$\hat{\varphi}_{yy}[\kappa]$')
```

```
plt.figure()
plt.hold(True)
plot_correlation_function(ccfyx)
plt.plot(np.arange(len(h)), h, 'g-')
plt.title('Estimated CCF and true impulse response')
plt.ylabel(r'$\hat{\varphi}_{yx}[\kappa]$, $h[k]$');
```



Exercise

- Why is the estimated CCF $\hat{\varphi}_{yx}[k]$ not exactly equal to the impulse response $h[k]$ of the system?
- What changes if you change the number of samples `N` of the input signal?

3.7 Measurement of Acoustic Impulse Responses

The propagation of sound from one position (e.g. transmitter) to another (e.g. receiver) conforms reasonable well to the properties of a linear time-invariant (LTI) system. Consequently, the impulse response $h[k]$ characterizes the propagation of sound between theses two positions. Impulse responses have various applications in acoustics. For instance as head-related impulse responses (HRIRs) or room impulse responses (RIRs) for the characterization of room acoustics.

The following example demonstrates how an acoustic impulse response can be measured with *correlation-based system identification techniques* using the soundcard of a computer. The module sounddevice provides access to the soundcard via PortAudio.

```
In [1]: %matplotlib inline

        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig
        import sounddevice as sd
```

3.7.1 Generation of the Measurement Signal

We generate white noise with an uniform distribution between ± 0.5 as the excitation signal $x[k]$

```
In [2]: fs = 44100  # sampling rate
        T = 5  # length of the measurement signal in sec
        Tr = 2  # length of the expected system response in sec

        x = np.random.uniform(-.5, .5, size=T*fs)
```

3.7.2 Playback of Measurement Signal and Recording of Room Response

The measurement signal $x[k]$ has to be played through the output of the soundcard and the response $y[k]$ has to be captured synchronously by the input of the soundcard. Since the length of the played/captured signal has to be equal, the measurement signal $x[k]$ is zero-padded so that the captured signal $y[k]$ includes the system response.

Be sure not to overdrive the speaker and the microphone by keeping the input level well below 0 dB.

```
In [3]: x = np.concatenate((x, np.zeros(Tr*fs)))
        y = sd.playrec(x, fs, channels=1)
        sd.wait()
        y = np.squeeze(y)

        print('Playback level: ', 20*np.log10(max(x)), ' dB')
        print('Input level: ', 20*np.log10(max(y)), ' dB')
```

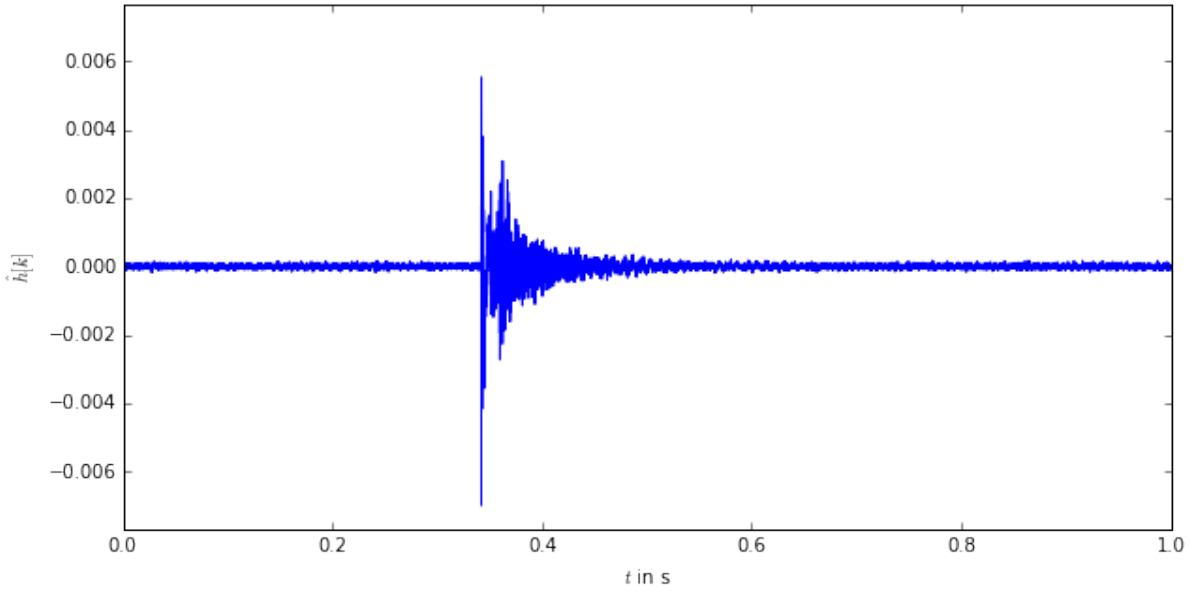
```
Playback level:  -6.02060087394  dB
Input level:  -2.23183822753  dB
```

3.7.3 Estimation of the Acoustic Impulse Response

The acoustic impulse response is estimated by cross-correlation $\varphi_{yx}[\kappa]$ of the output with the input signal. Since the cross-correlation function (CCF) for finite-length signals is given as $\varphi_{yx}[\kappa] = \frac{1}{K} \cdot y[\kappa] * x[-\kappa]$, the computation of the CCF can be speeded up with the fast convolution method.

```
In [4]: h = 1/len(y) * sig.fftconvolve(y, x[::-1], mode='full')
        h = h[fs*(T+Tr):fs*(T+2*Tr)]
```

```
In [5]: plt.figure(figsize=(10, 5))
        t = 1/fs * np.arange(len(h))
        plt.plot(t, h)
        plt.axis([0.0, 1.0, -1.1*np.max(np.abs(h)), 1.1*np.max(np.abs(h))])
        plt.xlabel(r'$t$ in s')
        plt.ylabel(r'$\hat{h}[k]$');
```



3.8 Power Spectral Densitity

For a weakly stationary real-valued random process $x[k]$, the *power spectral density* (PSD) $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is given as the discrete-time Fourier transformation (DTFT) of the auto-correlation function (ACF) $\varphi_{xx}[\kappa]$

$$\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{\kappa=-\infty}^{\infty} \varphi_{xx}[\kappa]\, \mathrm{e}^{\,-\mathrm{j}\,\Omega\,\kappa}$$

Under the assumption of a real-valued LTI system with impulse response $h[k] \in \mathbb{R}$, the power spectral density (PSD) $\Phi_{yy}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the output signal of an LTI system $y[k] = \mathcal{H}\{x[k]\}$ is derived by the DTFT of the *ACF of the output signal* $\varphi_{yy}[\kappa]$

$$\Phi_{yy}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{\kappa=-\infty}^{\infty} \underbrace{h[\kappa] * h[-\kappa]}_{\varphi_{hh}[\kappa]} * \varphi_{xx}[\kappa]\, \mathrm{e}^{\,-\mathrm{j}\,\Omega\,\kappa} \tag{3.8}$$

$$= H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot H(\mathrm{e}^{\,-\mathrm{j}\,\Omega}) \cdot \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = |H(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2 \cdot \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \tag{3.9}$$

The PSD of the output signal $\Phi_{yy}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of an LTI system is given by the PSD of the input signal $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ multiplied with the squared magnitude $|H(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2$ of the transfer function of the system.

3.9 Cross-Power Spectral Densities

The cross-power spectral densities $\Phi_{yx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and $\Phi_{xy}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ between the in- and output of an LTI system are given by the DTFT of the *cross-correlation functions* (CCF) $\varphi_{yx}[\kappa]$ and $\varphi_{xy}[\kappa]$. Hence,

$$\Phi_{yx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{\kappa=-\infty}^{\infty} h[\kappa] * \varphi_{xx}[\kappa]\, \mathrm{e}^{\,-\mathrm{j}\,\Omega\,\kappa} = \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

and

$$\Phi_{xy}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{\kappa=-\infty}^{\infty} h[-\kappa] * \varphi_{xx}[\kappa]\, \mathrm{e}^{\,-\mathrm{j}\,\Omega\,\kappa} = \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot H(\mathrm{e}^{\,-\mathrm{j}\,\Omega})$$

3.10 System Identification by Spectral Division

Using the result above for the cross-power spectral density $\Phi_{yx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ between out- and input, and the relation of the *CCF of finite-length signals to the convolution* one yields

$$H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{\Phi_{yx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})}{\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})} = \frac{\frac{1}{K} Y(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot X(\mathrm{e}^{\,-\mathrm{j}\,\Omega})}{\frac{1}{K} X(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot X(\mathrm{e}^{\,-\mathrm{j}\,\Omega})} = \frac{Y(\mathrm{e}^{\,\mathrm{j}\,\Omega})}{X(\mathrm{e}^{\,\mathrm{j}\,\Omega})}$$

holding for $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \neq 0$ and $X(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \neq 0$. Hence, the transfer function $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of an unknown system can be derived by dividing the spectrum of the output signal $Y(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ through the spectrum of the input signal $X(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. This is equal to the definition of the transfer function. However, care has to be taken that the spectrum of the input signal does not contain zeros.

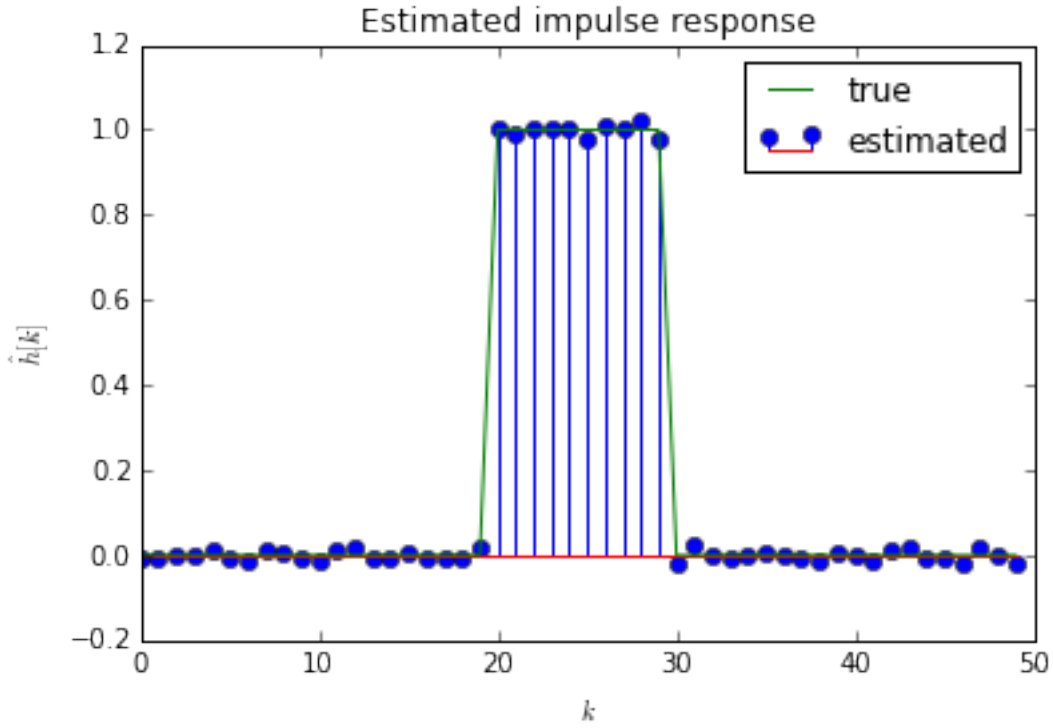
Above relation can be realized by the discrete Fourier transformation (DFT) by taking into account that a multiplication of two spectra $X[\mu] \cdot Y[\mu]$ results in the cyclic/periodic convolution $x[k] \circledast y[k]$. Since we aim at a linear convolution, zero-padding of the in- and output signal has to be applied.

3.10.1 Example

We consider the estimation of the impulse response $h[k] = \mathcal{F}_*^{-1}\{H(\mathrm{e}^{\,\mathrm{j}\,\Omega})\}$ of an unknown system using the spectral division method. Normal distributed white noise with variance $\sigma_n^2 = 1$ is used as input signal $x[k]$. In order to show the effect of sensor noise, normally distributed white noise $n[k]$ with the variance $\sigma_n^2 = 0.01$ is added to the output signal $y[k] = x[k] * h[k] + n[k]$.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt


        N = 1000  # number of samples for input signal

        # generate input signal
        x = np.random.normal(size=N, scale=1)  # normal distributed (zero-mean, unit-var
        # impulse response of the system
        h = np.concatenate((np.zeros(20), np.ones(10), np.zeros(20)))
        # output signal by convolution
        y = np.convolve(h, x, mode='full')
        # add noise to the output signal
        y = y + np.random.normal(size=y.shape, scale=.1)

        # zero-padding of input signal
        x = np.concatenate((x, np.zeros(len(h)-1)))
        # estimate transfer function
        H = np.fft.rfft(y)/np.fft.rfft(x)
        # compute inpulse response
        he = np.fft.irfft(H)
        he = he[0:len(h)]

        # plot impulse response
        plt.figure()
        plt.stem(he, label='estimated')
        plt.plot(h, 'g-', label='true')
```

```
plt.title('Estimated impulse response')
plt.xlabel(r'$k$')
plt.ylabel(r'$\hat{h}[k]$')
plt.legend();
```



Exercise

- Change the length `N` of the input signal. What happens?
- Change the variance σ_n^2 of the additive noise. What happens?

3.11 The Wiener Filter

The Wiener filter, named after *Nobert Wiener*, aims at estimating an unknown random signal by filtering a noisy observation of the signal. It has a wide variety of applications in noise reduction, system identification, deconvolution and signal detection. For instance, the Wiener filter can be used to denoise audio signals, like speech, or to remove noise from a picture.

3.11.1 Signal Model

The following signal model is underlying the following derivation of the Wiener filter

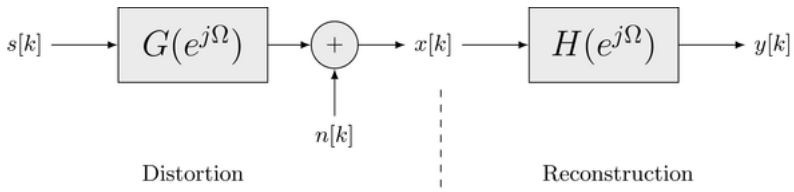


Fig. 3.2: Illustration: Signal model for the Wiener filter

The random signal $s[k]$ is subject to distortion by the linear time-invariant (LTI) system $G(\mathrm{e}^{\mathrm{j}\,\Omega})$ and additive noise $n[k]$, resulting in the observed signal $x[k] = s[k] * g[k] + n[k]$. The additive noise $n[k]$ is assumed to be

uncorrelated from $s[k]$. It is furthermore assumed that all random signals are weakly stationary. This distortion model holds for many practical problems, like e.g. the measurement of a physical quantity by a sensor.

The goal of the Wiener filter is to design the LTI system $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ such that the output signal $y[k]$ matches $s[k]$ as best as possible. In order to quantify this, the error signal

$$e[k] = y[k] - s[k]$$

is introduced. The quadratic mean of the error $e[k]$ is then given as

$$E\{|e[k]|^2\} = \frac{1}{2\pi} \int_{-\pi}^{\pi} \Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\,\mathrm{d}\Omega = \varphi_{ee}[\kappa = 0]$$

which is also known as *mean squared error* (MSE). We aim at the minimization of the MSE between the original signal $s[k]$ and its estimate $y[k]$.

3.11.2 Transfer Function of the Wiener Filter

At first, the Wiener filter shall only have access to the observed signal $x[k]$ and some statistical measures. It is assumed that the cross-power spectral density $\Phi_{xs}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ between the observed signal $x[k]$ and the original signal $s[k]$, and the power spectral density (PSD) of the observed signal $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ are known. This knowledge can either be gained by estimating both from measurements taken at an actual system or by using suitable statistical models.

The optimal filter is found by minimizing the MSE $E\{|e[k]|^2\}$ with respect to the transfer function $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. The solution of this optimization problem goes beyond the scope of this notebook and can be found in the literature, e.g. [*Girod et. al*]. The transfer function of the Wiener filter is given as

$$H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{\Phi_{sx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})}{\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})} = \frac{\Phi_{xs}(\mathrm{e}^{\,-\mathrm{j}\,\Omega})}{\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})}$$

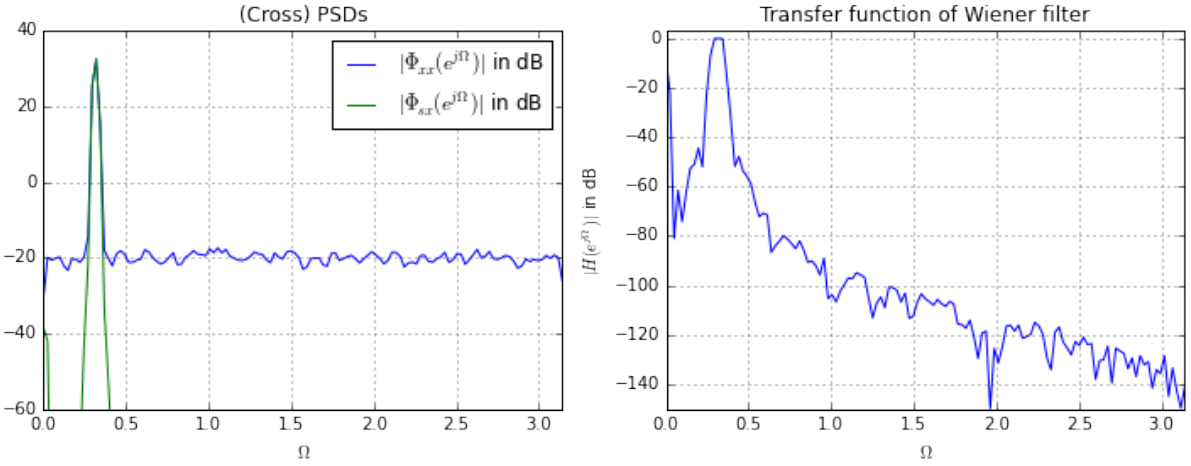
No knowledge on the actual distortion process is required. Only the PSDs $\Phi_{sx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ have to be known in order to estimate $s[k]$ from $x[k]$ in the minimum MSE sense. Care has to be taken that the filter $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is causal and stable in practical applications.

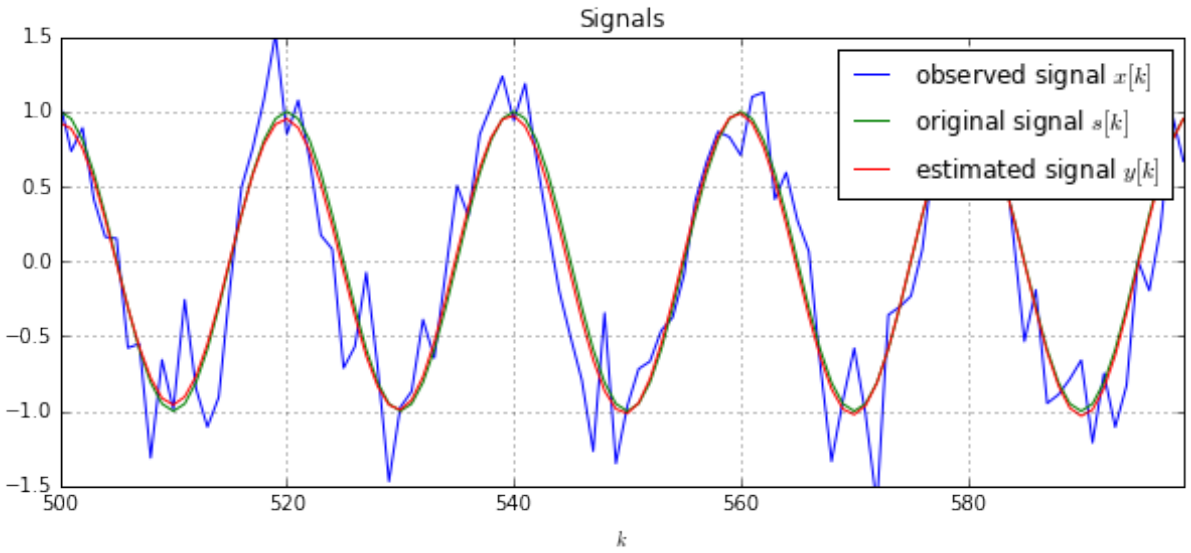
Example

The following example considers the estimation of the original signal from a distorted observation. It is assumed that the original signal is $s[k] = \sin[\Omega_0\, k]$ which is distorted by an LTI system and additive normally distributed zero-mean white noise with $\sigma_n^2 = 0.1$. The PSDs $\Phi_{sx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ are estimated from $s[k]$ and $x[k]$ using the Welch technique. The Wiener filter is applied to the observation $x[k]$ in order to compute the estimate $y[k]$ of $s[k]$.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig

        N = 8129  # number of samples
        M = 256  # length of Wiener filter
        Om0 = 0.1*np.pi  # frequency of original signal
        N0 = 0.1  # PSD of additive white noise

        # generate original signal
        s = np.cos(Om0 * np.arange(N))
        # generate observed signal
        g = 1/20*np.asarray([1, 2, 3, 4, 5, 4, 3, 2, 1])
        n = np.random.normal(size=N, scale=np.sqrt(N0))
        x = np.convolve(s, g, mode='same') + n
        # estimate (cross) PSDs using Welch technique
        f, Pxx = sig.csd(x, x, nperseg=M)
```

```
f, Psx = sig.csd(s, x, nperseg=M)
# compute Wiener filter
H = Psx/Pxx
H = H * np.exp(-1j*2*np.pi/len(H)*np.arange(len(H))*(len(H)//2))  # shift for ca
h = np.fft.irfft(H)
# apply Wiener filter to observation
y = np.convolve(x, h, mode='same')

# plot (cross) PSDs
Om = np.linspace(0, np.pi, num=len(H))

plt.figure(figsize=(10, 4))
plt.subplot(121)
plt.plot(Om, 20*np.log10(np.abs(.5*Pxx)), label=r'$| \Phi_{xx}(e^{j \Omega}) |$
plt.plot(Om, 20*np.log10(np.abs(.5*Psx)), label=r'$| \Phi_{sx}(e^{j \Omega}) |$
plt.title('(Cross) PSDs')
plt.xlabel(r'$\Omega$')
plt.legend()
plt.axis([0, np.pi, -60, 40])
plt.grid()

# plot transfer function of Wiener filter
plt.subplot(122)
plt.plot(Om, 20*np.log10(np.abs(H)))
plt.title('Transfer function of Wiener filter')
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$| H(e^{j \Omega}) |$ in dB')
plt.axis([0, np.pi, -150, 3])
plt.grid()
plt.tight_layout()

# plot signals
idx = np.arange(500, 600)
plt.figure(figsize=(10, 4))
plt.plot(idx, x[idx], label=r'observed signal $x[k]$')
plt.plot(idx, s[idx], label=r'original signal $s[k]$')
plt.plot(idx, y[idx], label=r'estimated signal $y[k]$')
plt.title('Signals')
plt.xlabel(r'$k$')
plt.axis([idx[0], idx[-1], -1.5, 1.5])
plt.legend()
plt.grid()
```





Exercise

- Take a look at the PSDs and the resulting transfer function of the Wiener filter. How does the Wiener filter remove the noise from the observed signal?
- Change the frequency `Om0` of the original signal $s[k]$ and the noise power `N0` of the additive noise. What changes?

3.11.3 Wiener Deconvolution

As discussed above, the general formulation of the Wiener filter is based on the knowledge of the PSDs $\Phi_{sx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ characterizing the distortion process and the observed signal respectively. These PSDs can be derived from the PSDs of the original signal $\Phi_{ss}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and the noise $\Phi_{nn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$, and the transfer function $G(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the distorting system.

Under the assumption that $n[k]$ is uncorrelated from $s[k]$ the PSD $\Phi_{sx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ can be derived as

$$\Phi_{sx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{ss}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot G(\mathrm{e}^{\,-\mathrm{j}\,\Omega})$$

and

$$\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{ss}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot |G(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2 + \Phi_{nn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

Introducing these results into the general formulation of the Wiener filter yields

$$H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{\Phi_{ss}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot G(\mathrm{e}^{\,-\mathrm{j}\,\Omega})}{\Phi_{ss}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot |G(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2 + \Phi_{nn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})}$$

This specialization is also known as *Wiener deconvolution filter*. The filter can be derived from the PSDs of the original signal and the noise, and the transfer function of the distorting system. This form is especially useful when the PSDs can be modeled by analytic probabilty density functions (PSDs). For instance, the additive noise can be modeled by white noise $\Phi_{nn}(e^{j\Omega}) = N_0$ in many cases.

3.11.4 Interpretation

The result above can be rewritten by introducing the frequency dependent signal-to-noise ratio $\mathrm{SNR}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{\Phi_{ss}(\mathrm{e}^{\,\mathrm{j}\,\Omega})}{\Phi_{nn}(\mathrm{e}^{\,\mathrm{j}\,\Omega})}$ between the orignal signal and the noise as

$$H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{1}{G(\mathrm{e}^{\,\mathrm{j}\,\Omega})} \cdot \left(\frac{|G(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2}{|G(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2 + \frac{1}{\mathrm{SNR}(\mathrm{e}^{\,\mathrm{j}\,\Omega})}} \right)$$

This form of the Wiener devonvolution filter can be discussed for two special cases:

1. If there is no additive noise $\Phi_{nn}(\mathrm{e}^{\mathrm{j}\,\Omega}) = 0$, the bracketed expression is equal to 1. Hence, the Wiener filter is simply given as the inverse system to the distorting system

$$H(\mathrm{e}^{\mathrm{j}\,\Omega}) = \frac{1}{G(\mathrm{e}^{\mathrm{j}\,\Omega})}$$

2. If the distorting system is just a pass through $G(\mathrm{e}^{\mathrm{j}\,\Omega}) = 1$, the Wiener filter is given as

$$H(\mathrm{e}^{\mathrm{j}\,\Omega}) = \frac{\mathrm{SNR}(\mathrm{e}^{\mathrm{j}\,\Omega})}{\mathrm{SNR}(\mathrm{e}^{\mathrm{j}\,\Omega}) + 1} = \frac{\Phi_{ss}(\mathrm{e}^{\mathrm{j}\,\Omega})}{\Phi_{ss}(\mathrm{e}^{\mathrm{j}\,\Omega}) + \Phi_{nn}(\mathrm{e}^{\mathrm{j}\,\Omega})}$$

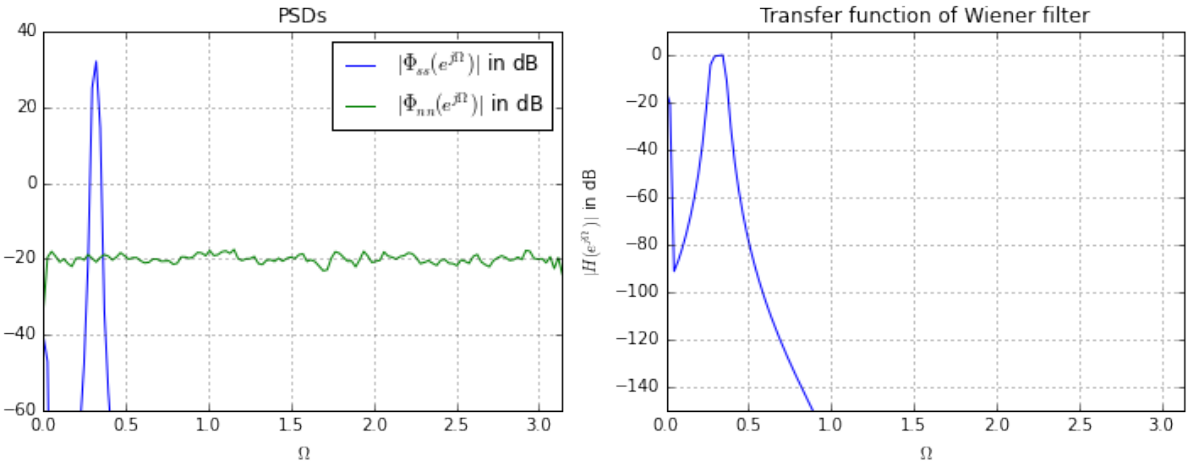
   Hence for a high $\mathrm{SNR}(\mathrm{e}^{\mathrm{j}\,\Omega})$, i.e. $\Phi_{ss}(\mathrm{e}^{\mathrm{j}\,\Omega}) \gg \Phi_{nn}(\mathrm{e}^{\mathrm{j}\,\Omega})$ at a given frequency Ω the transfer function approaches 1; and for a small SNR low values.

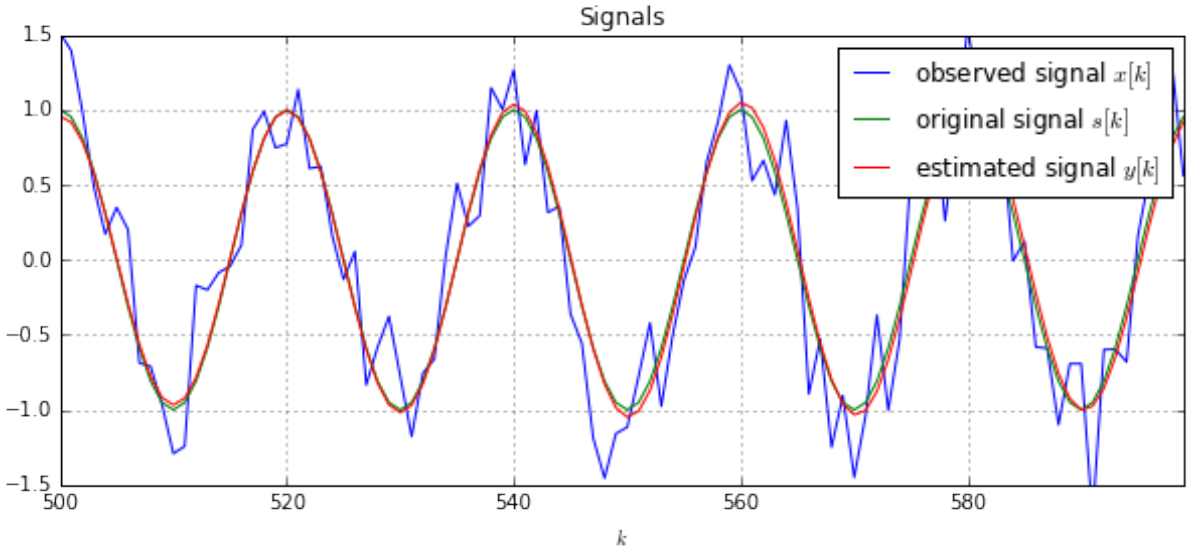
Example

The preceding example of the general Wiener filter will now be reevaluated with the Wiener deconvolution filter.

```
In [2]: N = 8129  # number of samples
        M = 256  # length of Wiener filter
        Om0 = 0.1*np.pi  # frequency of original signal
        N0 = .1  # PSD of additive white noise

        # generate original signal
        s = np.cos(Om0 * np.arange(N))
        # generate observed signal
        g = 1/20*np.asarray([1, 2, 3, 4, 5, 4, 3, 2, 1])
        n = np.random.normal(size=N, scale=np.sqrt(N0))
        x = np.convolve(s, g, mode='same') + n
        # estimate PSD
        f, Pss = sig.csd(s, s, nperseg=M)
        f, Pnn = sig.csd(n, n, nperseg=M)
        # compute Wiener filter
        G = np.fft.rfft(g, M)
        H = 1/G * (np.abs(G)**2 / (np.abs(G)**2 + N0/Pss))
        H = H * np.exp(-1j*2*np.pi/len(H)*np.arange(len(H))*(len(H)//2-8))  # shift for
        h = np.fft.irfft(H)
        # apply Wiener filter to observation
        y = np.convolve(x, h, mode='same')

        # plot (cross) PSDs
        Om = np.linspace(0, np.pi, num=len(H))

        plt.figure(figsize=(10, 4))
        plt.subplot(121)
        plt.plot(Om, 20*np.log10(np.abs(.5*Pss)), label=r'$| \Phi_{ss}(e^{j \Omega}) |$
        plt.plot(Om, 20*np.log10(np.abs(.5*Pnn)), label=r'$| \Phi_{nn}(e^{j \Omega}) |$
        plt.title('PSDs')
        plt.xlabel(r'$\Omega$')
        plt.legend()
        plt.axis([0, np.pi, -60, 40])
        plt.grid()

        # plot transfer function of Wiener filter
        plt.subplot(122)
        plt.plot(Om, 20*np.log10(np.abs(H)))
        plt.title('Transfer function of Wiener filter')
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$| H(e^{j \Omega}) |$ in dB')
        plt.axis([0, np.pi, -150, 10])
```

```
plt.grid()
plt.tight_layout()

# plot signals
idx = np.arange(500, 600)
plt.figure(figsize=(10, 4))
plt.plot(idx, x[idx], label=r'observed signal $x[k]$')
plt.plot(idx, s[idx], label=r'original signal $s[k]$')
plt.plot(idx, y[idx], label=r'estimated signal $y[k]$')
plt.title('Signals')
plt.xlabel(r'$k$')
plt.axis([idx[0], idx[-1], -1.5, 1.5])
plt.legend()
plt.grid()
```





Exercise

- What is different compared to the general Wiener filter? Why?

CHAPTER 4

Spectral Estimation of Random Signals

4.1 Introduction

In the preceding sections various statistical measures have been introduced to characterize random processes and signals. For instance the probability density function (PDF) $p_x(\theta)$, the mean value μ_x, the auto-correlation function (ACF) $\varphi_{xx}[\kappa]$ and its Fourier transformation, the power spectral density (PSD) $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. For many random processes whose internal structure is known these measures can be given in closed-form. However, for practical random signals, measures of interest have to be estimated from a limited number of samples. These estimated quantities can e.g. be used to fit a parametric model of the random process or as parameters in algorithms.

4.1.1 Problem Statement

The estimation of the spectral properties of a random signal is of special interest for spectral analysis. The discrete Fourier transform (DFT) of a random signal is also random. It is not very well suited to get insights into the spectral structure of a random signal. We therefore aim at estimating the PSD $\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of a weakly stationary and ergodic process from a limited number of samples. This is known as *spectral (density) estimation*. Many techniques have been developed for this purpose. They can be classified into

1. non-parametric and
2. parametric

techniques. Non-parametric techniques estimate the PSD of the random signal without assuming any particular structure for the generating random process. In contrary, parametric techniques assume that the generating random process can be modeled by few parameters. Their aim is to estimate these parameters in order to characterize the random signal.

4.1.2 Evaluation

The estimate $\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ can be regarded as a random signal itself. The performance of an estimator is therefore evaluated in a statistical sense. For the PSD, the following metrics are of interest

Bias

The bias of an estimator

$$b_{\hat{\Phi}_{xx}} = E\{\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) - \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\} = E\{\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\} - \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

quantifies the difference between the estimated $\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and the true $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. An estimator is biased if $b_{\hat{\Phi}_{xx}} \neq 0$ and bias-free if $b_{\hat{\Phi}_{xx}} = 0$.

Variance

The variance of an estimator

$$\sigma^2_{\hat{\Phi}_{xx}} = E\left\{ \left(\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) - E\{\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\} \right)^2 \right\}$$

quantifies its quadratic deviation from its mean value $E\{\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\}$.

Consistency

A consistent estimator is an estimator for which the following conditions hold for a large number N of samples:

1. the estimator is unbiased

$$\lim_{N \to \infty} b_{\hat{\Phi}_{xx}} = 0$$

2. its variance converges towards zero

$$\lim_{N \to \infty} \sigma^2_{\hat{\Phi}_{xx}} = 0$$

3. it convergences in probability to the true $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$

$$\lim_{N \to \infty} \Pr\left\{ |\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) - \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})| > \alpha \right\} = 0$$

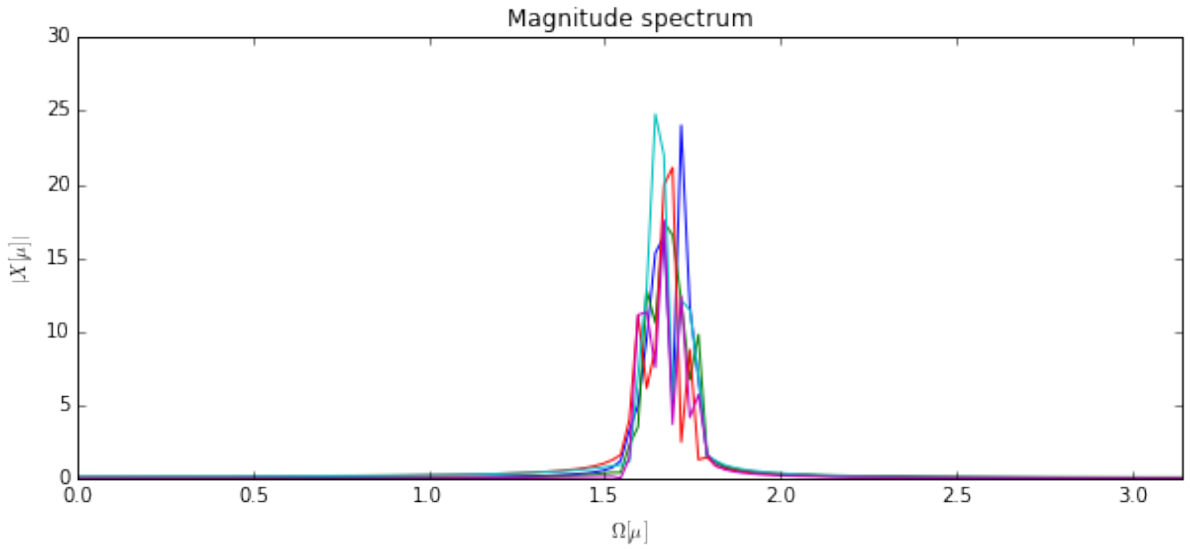
   where $\alpha > 0$ denotes a (small) constant.

Consistency is a desired property of an estimator. It ensures that if the number of samples N increases towards infinity, the resulting estimates converges towards the true PSD.

Example

The following example computes and plots the magnitude spectra $|X_n[\mu]|$ of an ensemble of random signals $x_n[k]$. In the plot, each color denotes one sample function.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig

        N = 256  # number of samples
        M = 5  # number of sample functions

        # generate random signal
        x = np.random.normal(size=(M, N))
        h = sig.firwin2(N, [0, .5, .52, .55, .57, 1], [0, 0, 1, 1, 0, 0])
        x = [np.convolve(xi, h, mode='same') for xi in x]

        # DFT of signal
        X = np.fft.rfft(x, axis=1)
        Om = np.linspace(0, np.pi, X.shape[1])

        # plot signal and its spectrum
        plt.figure(figsize=(10,4))
        plt.plot(Om, np.abs(X.T))
        plt.title('Magnitude spectrum')
        plt.xlabel(r'$\Omega[\mu]$')
        plt.ylabel(r'$|X[\mu]|$')
        plt.axis([0, np.pi, 0, 30]);
```



Exercise

- What can you conclude on the spectral properties of the random process?
- Increase the number `N` of samples. What changes? What does not change with respect to the evaluation criteria introduced above?
- Is the DFT a consistent estimator for the spectral properties of a random process?

4.2 The Periodogram

The periodogram is an estimator for the power spectral density (PSD) $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of a random signal $x[k]$. We assume a weakly ergodic real-valued random process in the following.

4.2.1 Definition

The PSD is given as the *discrete time Fourier transformation (DTFT) of the auto-correlation function (ACF)*

$$\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathcal{F}_*\{\varphi_{xx}[\kappa]\}$$

Hence, the PSD can be computed from an estimate of the ACF. Let's assume that we want to estimate the PSD from N samples of the random signal $x[k]$ by way of the ACF. The truncated signal is given as

$$x_N[k] = x[k] \cdot \mathrm{rect}_N[k]$$

The ACF is estimated by using its definition in a straightforward manner. For a random signal $x_N[k]$ of finite length, the estimated ACF $\hat{\varphi}_{xx}[\kappa]$ can be expressed *in terms of a convolution*

$$\hat{\varphi}_{xx}[\kappa] = \frac{1}{N} \cdot x_N[k] * x_N[-k]$$

Applying the DTFT to both sides and rearranging the terms yields

$$\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{1}{N} X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega}) X_N(\mathrm{e}^{\,-\mathrm{j}\,\Omega}) = \frac{1}{N} |X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2$$

where the latter equality has been derived by applying the symmetry relations of the DTFT. This estimate of the PSD is known as the periodogram. It can be computed directly from the DTFT

$$X_N(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{k=0}^{N-1} x_N[k]\, \mathrm{e}^{\,-\mathrm{j}\,\Omega\, k}$$

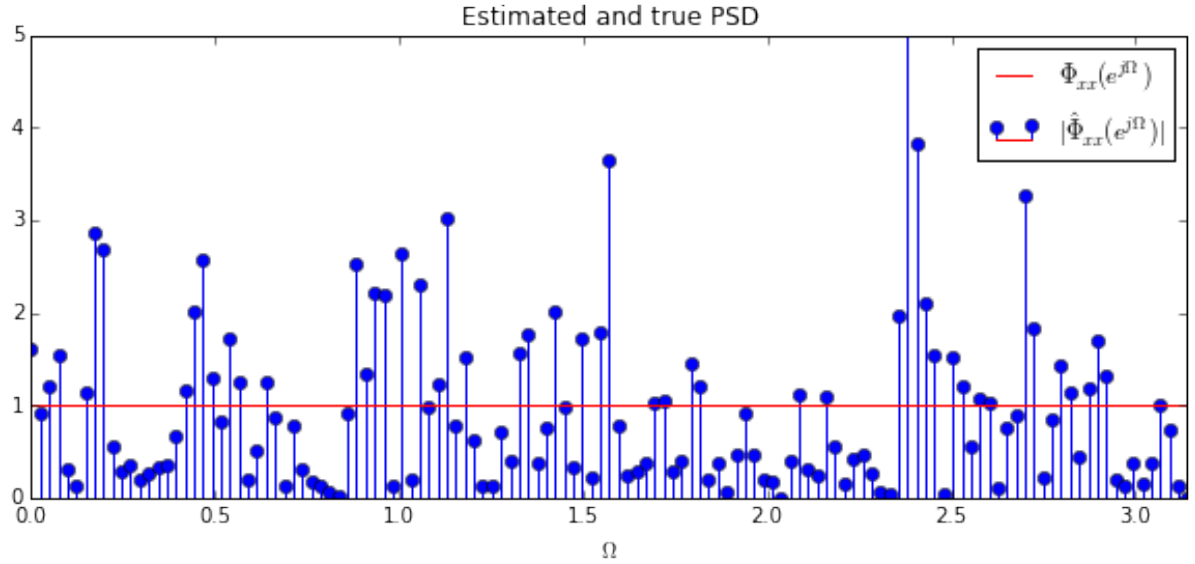
of the truncated random signal (here for $0 \leq k \leq N-1$).

4.2.2 Example

The following example estimates the PSD of a random process which draws samples from normal distributed white noise with zero-mean and unit variance. The true PSD is given as $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = 1$. In order to compute the periodogram by the discrete Fourier transformation (DFT), the signal $x[k]$ has to be zero-padded to ensure that above convolution is not circular.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        N = 128  # number of samples

        # generate random signal
        x = np.random.normal(size=N)

        # compute magnitude of the periodogram
        x = np.concatenate((x, np.zeros_like(x)))
        X = np.fft.rfft(x)
        Om = np.linspace(0, np.pi, len(X))
        Sxx = 1/N * abs(X)**2

        # plot results
        plt.figure(figsize=(10,4))
        plt.stem(Om, Sxx, 'b', label=r'$|\hat{\Phi}_{xx}(e^{j \Omega})|$')
        plt.plot(Om, np.ones_like(Sxx), 'r', label=r'$\Phi_{xx}(e^{j \Omega})$')
        plt.title('Estimated and true PSD')
        plt.xlabel(r'$\Omega$')
        plt.axis([0, np.pi, 0, 5])
        plt.legend()

        # compute mean value of the periodogram
        print('Mean value of the periodogram: %f' %np.mean(np.abs(Sxx)))
```

```
Mean value of the periodogram: 0.960033
```



Exercise

- What do you have to change to evaluate experimentally if the periodogram is a consistent estimator?
- Based on the results, is the periodogram a consistent estimator?

4.2.3 Evaluation

From above numerical example it should have become clear that the periodogram is no consistent estimator for the PSD $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. It can be shown that the estimator is asymptotically bias free when $N \to \infty$, hence

$$\lim_{N \to \infty} E\{\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\} = \Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

This is due to the leakage effect which limits the spectral resolution for signals of finite length.

The variance of the estimator does not converge towards zero

$$\lim_{N \to \infty} \sigma^2_{\hat{\Phi}_{xx}} \neq 0$$

This is due to the fact that with increasing N also the number of independent frequencies $\Omega = \frac{2\pi}{N}\mu$ for $\mu = 0, 1, \ldots, N-1$ increases.

The periodogram is the basis for a variety of advanced estimation techniques for the PSD. These techniques rely on averaging or smoothing of (overlapping) periodograms.

4.3 The Welch Method

In the previous section it has been shown that the periodogram, as a non-parametric estimator of the power spectral density (PSD) $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of a random signal $x[k]$, is not consistent. This is due to the fact that its variance does not converge towards zero even when the length of the random signal is increased towards infinity. In order to overcome this problem, the [Bartlett method](https://en.wikipedia.org/wiki/Bartlett's_method) and [Welch method](https://en.wikipedia.org/wiki/Welch's_method)

1. split the random signal into segments,
2. estimate the PSD for each segment, and
3. average over these local estimates.

The averaging reduces the variance of the estimated PSD. While Barlett's method uses non-overlapping segments, Welch's is a generalization using windowed overlapping segments. As before we assume a weakly ergodic real-valued random process for the discussion of Welch's method.

4.3.1 Derivation

Let's assume that we split the random signal $x[k]$ into L overlapping segments $x_l[k]$ of length N with $0 \leq l \leq L-1$, starting at multiples of the stepsize $M \in 1, 2, \ldots, N$. These segments are then windowed by the window $w[k]$ of length N, resulting in a windowed l-th segment. The discrete time Fourier transformation (DTFT) $X_l(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the windowed l-th segment is thus given as

$$X_l(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{k=0}^{N-1} x[k + l \cdot M]\, w[k]\, \mathrm{e}^{\,-\mathrm{j}\,\Omega\, k}$$

where the window $w[k]$ defined within $0 \leq k \leq N-1$ should be normalized as $\frac{1}{N} \sum_{k=0}^{N-1} |w[k]|^2 = 1$. The stepsize M determines the overlap between the segments. In general $N - M$ number of samples overlap between adjacent segments, for $M = N$ no overlap occurs. The overlap is sometimes given as ratio $\frac{N-M}{N} \cdot 100\%$.

Introducing $X_l(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ into the definition of the periodogram yields the periodogram of the l-th segment

$$\hat{\Phi}_{xx,l}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{1}{N} |X_l(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2$$

The estimated PSD is then given by averaging over the segment's periodograms $\hat{\Phi}_{xx,l}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$

$$\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{1}{L} \sum_{l=0}^{L-1} \hat{\Phi}_{xx,l}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$$

Note, that the total number L of segments has to be chosen such that the last required sample $(L-1) \cdot M + N - 1$ does not exceed the total length of the random signal. Otherwise the last segment $x_{L-1}[k]$ may also be zeropadded towards length N.

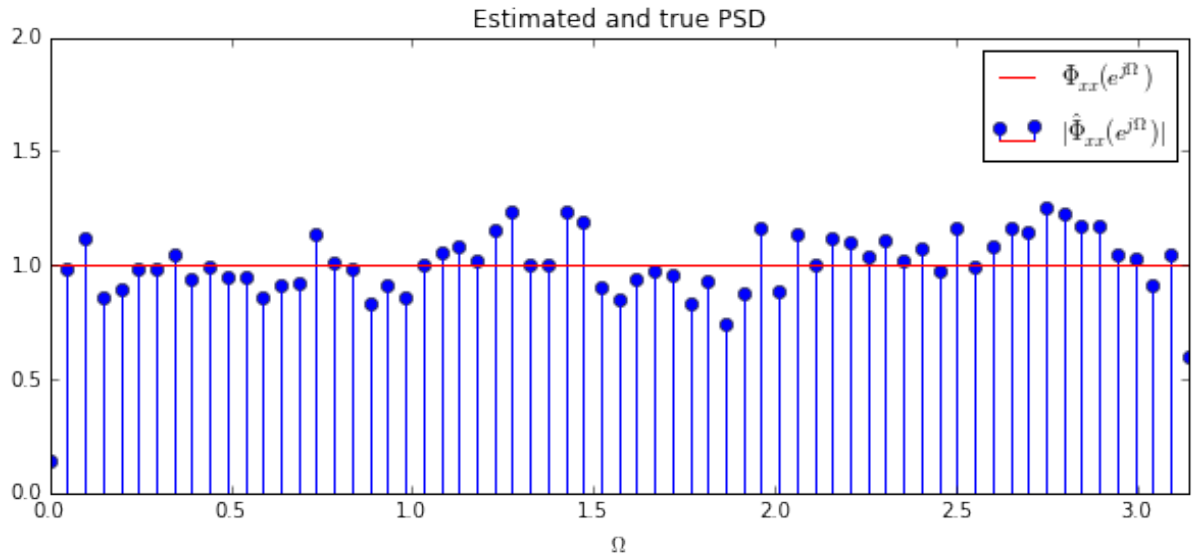
The Bartlett method uses a rectangular window and non-overlapping segments. The Welch method uses overlapping segments and a window that must be chosen according to the intended spectral analysis task.

4.3.2 Example

The following example is equivalent to the *periodogram example*. We aim at estimating the PSD of a random process which draws samples from normal distributed white noise with zero-mean and unit variance. The true PSD is given as $\Phi_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = 1$.

```
In [21]: %matplotlib inline
         import numpy as np
         import matplotlib.pyplot as plt
         import scipy.signal as sig

         N = 128  # length of segments
         M = 64  # stepsize
         L = 100  # total number of segments

         # generate random signal
         x = np.random.normal(size=L*M)

         # compute periodogram by Welch's method
         nf, Pxx = sig.welch(x, window='hamming', nperseg=N, noverlap=(N-M))
         Pxx = .5*Pxx  # due to normalization in scipy.signal
         Om = 2*np.pi*nf

         # plot results
         plt.figure(figsize=(10,4))
         plt.stem(Om, Pxx, 'b', label=r'$|\hat{\Phi}_{xx}(e^{j \Omega})|$')
         plt.plot(Om, np.ones_like(Pxx), 'r', label=r'$\Phi_{xx}(e^{j \Omega})$')
         plt.title('Estimated and true PSD')
         plt.xlabel(r'$\Omega$')
         plt.axis([0, np.pi, 0, 2])
         plt.legend()

         # compute mean value of the periodogram
         print('Mean value of the periodogram: %f' %np.mean(np.abs(Pxx)))
```

```
Mean value of the periodogram: 0.997852
```



Exercise

- Compare the results to the periodogram example. Is the variance of the estimator lower?
- Change the number of segments `L` and check if the variance reduces further
- Change the segment length `N` and stepsize `M`. What changes?

4.3.3 Evaluation

It is shown in [*Stoica et al.*] that Welch's method is asymptotically unbiased. Under the assumption of a weakly stationary random process, the periodograms $\hat{\Phi}_{xx,l}(e^{j\Omega})$ of the segments can be assumed to be approximately uncorrelated. Hence, averaging over these reduces the variance of the estimator. It can be shown formally that in the limiting case of an infinitely number of segments (infintely long signal) the variance tends to zero. As a result Welch's method is an asymptotically consistent estimator of the PSD.

For a finite segment length N, the properties of the estimated PSD $\hat{\Phi}_{xx}(e^{j\Omega})$ depend on the length N of the segments and the window function $w[k]$ due to the leakage effect.

4.4 Parametric Methods

4.4.1 Motivation

Non-parametric methods for the estimation of the power spectral density (PSD), like the periodogram or Welch's method, don't rely on a-priori information about the process generating the random signal. Often some a-priori information is available that can be used to formulate a parametric model of the random process. The goal is then to estimate these parameters in order to characterize the random signal. Such techniques are known as *'parametric methods <https://en.wikipedia.org/wiki/Spectral_density_estimation#Parametric_estimation>'__* or *model-based methods*. The incorporation of a-priori knowledge can improve the estimation of the PSD significantly, as long as the underlying model is a valid description of the random process. The parametric model of the random process can also be used to generate random signals with a desired PSD.

4.4.2 Process Models

For the remainder we assume weakly stationary real-valued random processes. For many applications the process can be modeled by a linear-time invariant (LTI) system

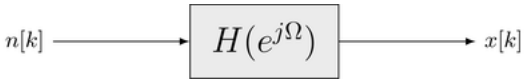


Fig. 4.1: Synthesis process model

where $n[k]$ is white noise and $H(\mathrm{e}^{\mathrm{j}\,\Omega})$ denotes the transfer function of the system. In general, the random signal $x[k]$ will be correlated as a result of the processing of the uncorrelated input signal $n[k]$ by the system $H(\mathrm{e}^{\mathrm{j}\,\Omega})$. Due to the white noise assumption $\Phi_{nn}(\mathrm{e}^{\mathrm{j}\,\Omega}) = N_0$, the PSD of the random process is given as

$$\Phi_{xx}(\mathrm{e}^{\mathrm{j}\,\Omega}) = N_0 \cdot |H(\mathrm{e}^{\mathrm{j}\,\Omega})|^2$$

Parametric methods model the system $H(\mathrm{e}^{\mathrm{j}\,\Omega})$ by a limited number of parameters. These parameters are then estimated from $x[k]$, providing an estimate $\hat{H}(\mathrm{e}^{\mathrm{j}\,\Omega})$ of the transfer function. This estimate is then used to calculate the desired estimate $\hat{\Phi}_{xx}(\mathrm{e}^{\mathrm{j}\,\Omega})$ of the PSD.

Autoregressive model

The autoregressive (AR) model assumes a recursive system with a direct path. Its output relation is given as

$$x[k] = \sum_{n=1}^{N} a_n \cdot x[k-n] + n[k]$$

where a_n denote the coefficients of the recursive path and N the order of the model. Its system function $H(z)$ is derived by z-transformation of the output relation

$$H(z) = \frac{1}{1 - \sum_{n=1}^{N} a_n z^{-n}}$$

Hence, the AR model is a pole-only model of the system.

Moving average model

The moving average (MA) model assumes a non-recursive system. The output relation is given as

$$x[k] = \sum_{m=0}^{M-1} b_m \cdot n[k-m] = h[k] * n[k]$$

with the impulse response of the system $h[k] = [b_0, b_1, \ldots, b_{M-1}]$. The MA model is a finite impulse response (FIR) model of the random process. Its system function is given as

$$H(z) = \mathcal{Z}\{h[k]\} = \sum_{m=0}^{M-1} b_m \, z^{-m}$$

Autoregressive moving average model

The autoregressive moving average (ARMA) model is a combination of the AR and MA model. It constitutes a general linear process model. Its output relation is given as

$$x[k] = \sum_{n=1}^{N} a_n \cdot x[k-n] + \sum_{m=0}^{M-1} b_m \cdot n[k-m]$$

Its system function reads

$$H(z) = \frac{\sum_{m=0}^{M-1} b_m \, z^{-m}}{1 - \sum_{n=1}^{N} a_n z^{-n}}$$

4.4.3 Parametric Spectral Estimation

The models above describe the synthesis of the samples $x[k]$ from the white noise $n[k]$. For spectral estimation only the random signal $x[k]$ is known and we are aiming at estimating the parameters of the model. This can be achieved by determining an analyzing system $G(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ such to decorrelate the signal $x[k]$

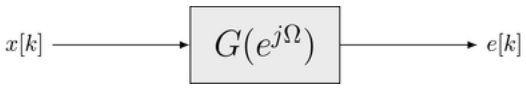


Fig. 4.2: Analysis process model

where $e[k]$ should be white noise. Due to its desired operation, the filter $G(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is also denoted as *whitening filter*. The optimal filter $G(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is given by the inverse system $\frac{1}{H(\mathrm{e}^{\,\mathrm{j}\,\Omega})}$. However, $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is in general not known. But this nevertheless implies that our linear process model of $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ also applies to $G(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. Various techniques have been developed to estimate the parameters of the filter $G(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ such that $e[k]$ becomes decorrelated. For instance, by expressing the auto-correlation function (ACF) $\varphi_{xx}[\kappa]$ in terms of the model parameters and solving with respect to these. The underlying set of equations are known as Yule-Walker equations.

Once the model parameters have been estimated, these can be used to calculate an estimate $\hat{G}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the analysis system. The desired estimate of the PSD is then given as

$$\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega})}{|\hat{G}(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2}$$

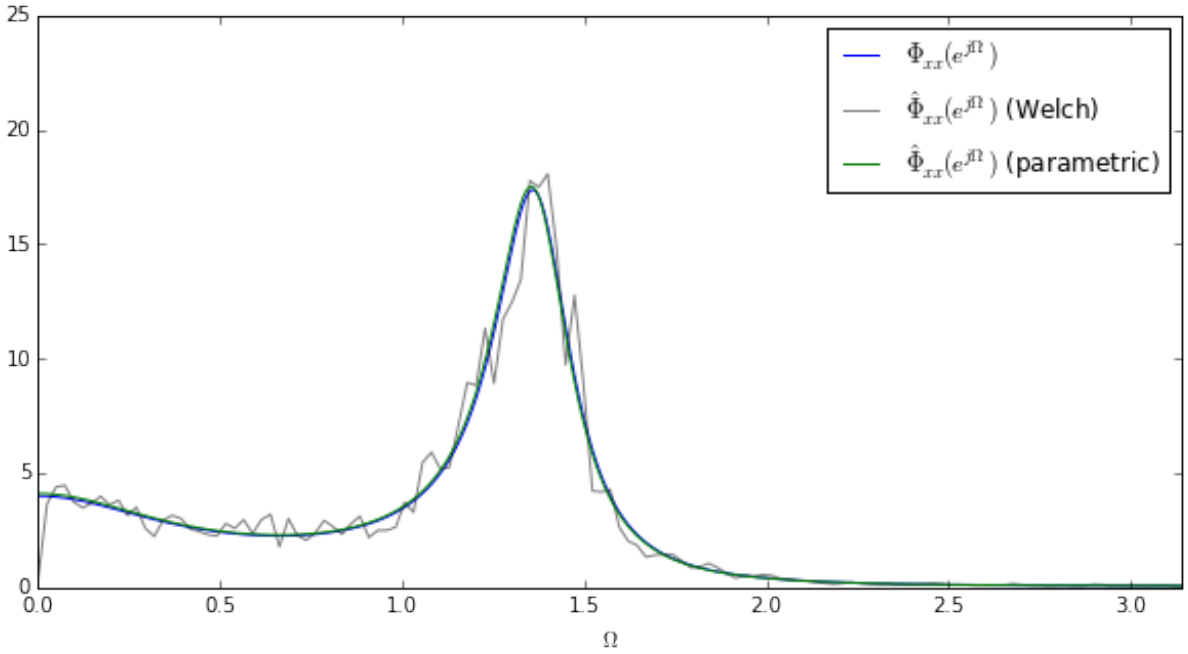
where if $e[k]$ is white noise, $\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = N_0$.

4.4.4 Example

In the follwing example $n[k]$ is drawn from normal distributed white noise with $N_0 = 1$. The Yule-Walker equations are used to estimate the parameters of an AR model of $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. The implementation provided by `statsmodels.api.regression.yule_walker` returns the estimated AR coefficients of the system $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. These parameters are then used to numerically evaluate the estimated transfer function, resulting in $\hat{\Phi}_{xx}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = 1 \cdot |\hat{H}(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2$.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import statsmodels.api as sm
        import scipy.signal as sig


        K = 4096  # length of random signal
        N = 3  # order of AR model
        a = np.array((1, -1, .5))  # coefficients of AR model

        # generate random signal n[k]
        n = np.random.normal(size=K)

        # AR model for random signal x[k]
        x = np.zeros(K)
        for k in np.arange(3, K):
            x[k] = a[0]*x[k-1] + a[1]*x[k-2] + a[2]*x[k-3] + n[k]

        # estimate AR parameters by Yule-Walker method
        rho, sigma = sm.regression.yule_walker(x, order=N, method='mle')

        # compute true and estimated transfer function
```

```
Om, H = sig.freqz(1, np.insert(-a, 0, 1))
Om, He = sig.freqz(1, np.insert(-rho, 0, 1))
# compute PSD by Welch method
Om2, Pxx = sig.welch(x, return_onesided=True)

# plot PSDs
plt.figure(figsize=(10,5))
plt.plot(Om, np.abs(H)**2, label=r'$\Phi_{xx}(e^{j\Omega})$')
plt.plot(Om2*2*np.pi, .5*np.abs(Pxx), 'k-', alpha=.5 , label=r'$\hat{\Phi}_{xx}
plt.plot(Om, np.abs(He)**2, label=r'$\hat{\Phi}_{xx}(e^{j\Omega})$ (parametric)'

plt.xlabel(r'$\Omega$')
plt.axis([0, np.pi, 0, 25])
plt.legend();
```



Exercise

- Change the order `N` of the AR model used for estimation by the Yule-Walker equations. What happens if the order is smaller or higher than the order of the true system? Why?
- Change the number of samples `K`. Is the estimator consistent?

CHAPTER 5

Quantization

5.1 Introduction

Digital signal processors and general purpose processors can only perform arithmetic operations within a limited number range. So far we considered discrete signals with continuous amplitude values. These cannot be handled by processors in a straightforward manner. Quantization is the process of mapping a continuous amplitude to a countable set of amplitude values. This refers also to the *requantization* of a signal from a large set of countable amplitude values to a smaller set. Scalar quantization is an instantaneous and memoryless operation. It can be applied to the continuous amplitude signal, also referred to as *analog signal* or to the (time-)discrete signal. The quantized discrete signal is termed as *digital signal*. The connections between the different domains is illustrated in the following for time dependent signals.

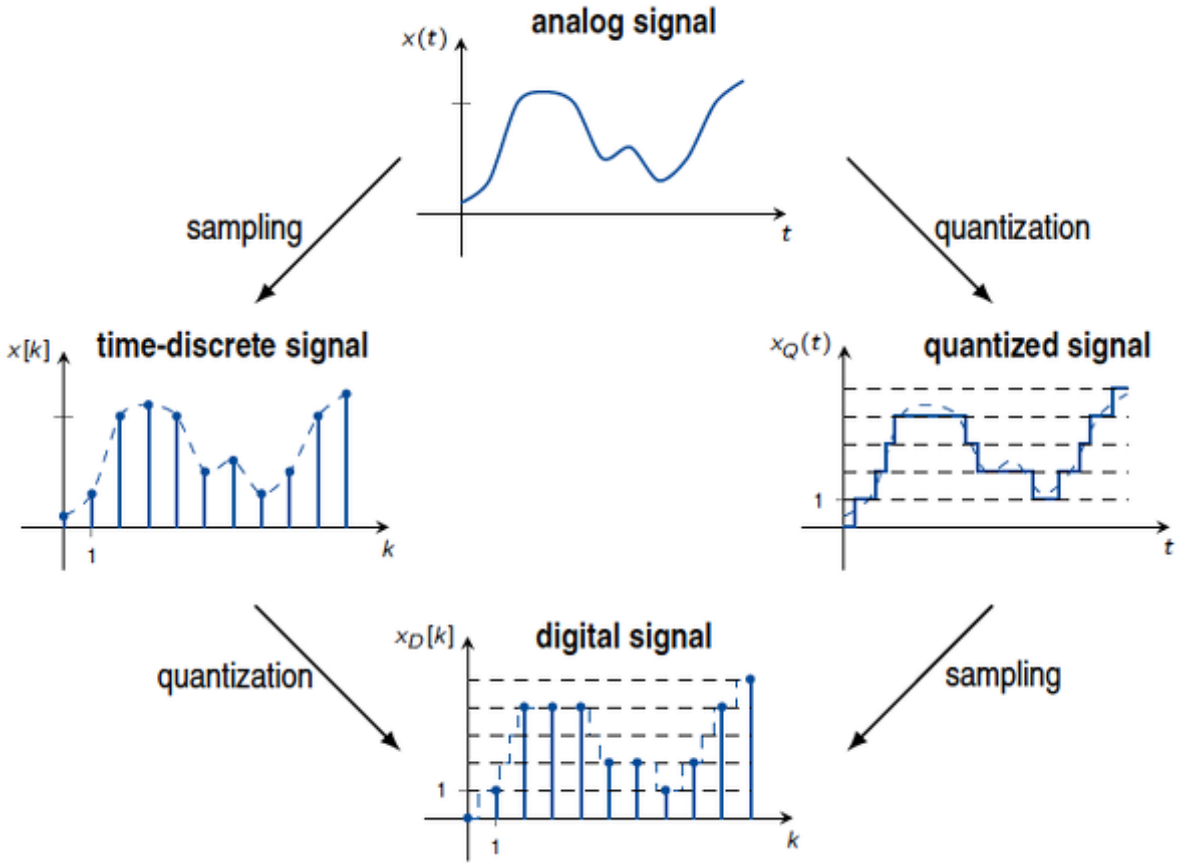


Fig. 5.1: Analog, discrete and digital signals

5.1.1 Model of the Quantization Process

In order to quantify the effects of quantizing a continuous amplitude signal, a model of the quantization process is formulated. We restrict our considerations to a discrete real-valued signal $x[k]$. In order to map the continuous amplitude to a quantized representation the following model is used

$$x_Q[k] = g(\ \lfloor\, f(x[k])\,\rfloor\ )$$

where $g(\cdot)$ and $f(\cdot)$ denote real-valued mapping functions, and $\lfloor\cdot\rfloor$ a rounding operation. The quantization process can be split into two stages

1. **Forward quantization** The mapping $f(x[k])$ maps the signal $x[k]$ such that it is suitable for the rounding operation. This may be a scaling of the signal or a non-linear mapping. The result of the rounding operation is an integer number $\lfloor\, f(x[k])\,\rfloor \in \mathbb{Z}$, which is termed as *quantization index*.
2. **Inverse quantization** The mapping $g(\cdot)$, maps the quantization index to the quantized value $x_Q[k]$ such that it is an approximation of $x[k]$. This may be a scaling or a non-linear operation.

The quantization error/quantization noise $e[k]$ is defined as

$$e[k] = x_Q[k] - x[k]$$

Rearranging yields that the quantization process can be modeled by adding the quantization noise to the discrete signal

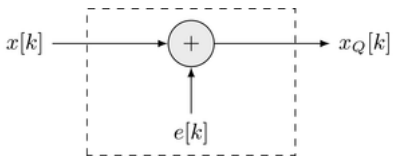


Fig. 5.2: Model of quantization process

Example

In order to illustrate the introduced model, the quantization of one period of a sine signal is considered

$$x[k] = \sin[\Omega_0 k]$$

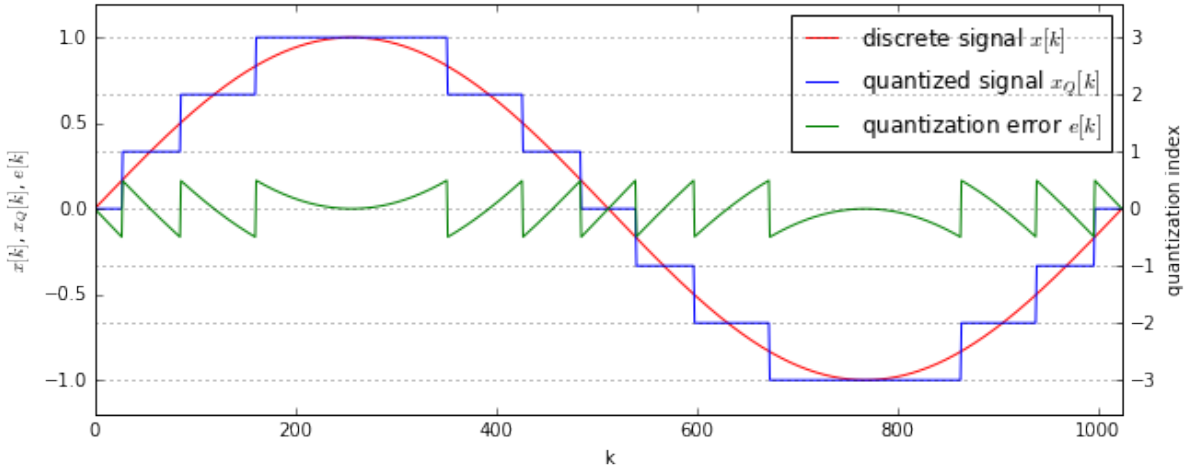
using $f(x[k]) = 3 \cdot x[k]$ and $g(i) = \frac{1}{3} \cdot i$. The rounding is realized by the nearest integer function. The quantized signal is then given as

$$x_Q[k] = \frac{1}{3} \cdot \lfloor\, 3 \cdot \sin[\Omega_0 k]\,\rfloor$$

For ease of illustration the signals are not shown by stem plots.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        N = 1024  # length of signal

        # generate signal
        x = np.sin(2*np.pi/N * np.arange(N))
        # quantize signal
        xi = np.round(3 * x)
        xQ = 1/3 * xi
        e = xQ - x

        # plot (quantized) signals
```

```
fig, ax1 = plt.subplots(figsize=(10,4))
ax2 = ax1.twinx()

ax1.plot(x, 'r', label=r'discrete signal $x[k]$')
ax1.plot(xQ, 'b', label=r'quantized signal $x_Q[k]$')
ax1.plot(e, 'g', label=r'quantization error $e[k]$')
ax1.set_xlabel('k')
ax1.set_ylabel(r'$x[k]$, $x_Q[k]$, $e[k]$')
ax1.axis([0, N, -1.2, 1.2])
ax1.legend()

ax2.set_ylim([-3.6, 3.6])
ax2.set_ylabel('quantization index')
ax2.grid()
```



Exercise

- Investigate the quantization noise $e[k]$. Is its amplitude bounded?
- If you would represent the quantization index (shown on the right side) by a binary number, how much bits would you need?
- Try out other rounding operations like `np.floor()` and `np.ceil()` instead of `np.round()`. What changes?

5.1.2 Properties

Without knowledge of the quantization error $e[k]$, the signal $x[k]$ cannot be reconstructed exactly knowing only its quantization index or quantized representation $x_Q[k]$. The quantization error $e[k]$ itself depends on the signal $x[k]$. Therefore, quantization is in general an irreversible process. The mapping from $x[k]$ to $x_Q[k]$ is furthermore non-linear, since the superposition principle does not hold in general. Summarizing, quantization is an inherently irreversible and non-linear process.

5.1.3 Applications

Quantization has widespread applications in Digital Signal Processing. For instance in

- Analog-to-Digital conversion
- Lossy compression of signals (speech, music, video, ...)
- Storage and Transmission (Pulse-Code Modulation, ...)

5.2 Characteristic of a Linear Uniform Quantizer

The characteristics of a quantizer depend on the mapping functions $f(\cdot)$, $g(\cdot)$ and the rounding operation $\lfloor \cdot \rfloor$ introduced in the previous section. A linear quantizer bases on linear mapping functions $f(\cdot)$ and $g(\cdot)$. A uniform quantizer splits the mapped input signal into quantization steps of equal size. Quantizers can be described by their nonlinear in-/output characteristic $x_Q[k] = \mathcal{Q}\{x[k]\}$, where $\mathcal{Q}\{\cdot\}$ denotes the quantization process. For linear uniform quantization it is common to differentiate between two characteristic curves, the so called mid-tread and mid-rise.

5.2.1 Mid-Tread Chacteristic Curve

The in-/output relation of the mid-tread quantizer is given as

$$x_Q[k] = Q \cdot \underbrace{\left\lfloor \frac{x[k]}{Q} + \frac{1}{2} \right\rfloor}_{index}$$

where Q denotes the quantization step size and $\lfloor \cdot \rfloor$ the floor function which maps a real number to the largest integer not greater than its argument. Without restricting $x[k]$ in amplitude, the resulting quantization indexes are countable infinite. For a finite number of quantization indexes, the input signal has to be restricted to a minimal/maximal amplitude $x_{\min} < x[k] < x_{\max}$ before quantization. The resulting quantization characteristic of a linear uniform mid-tread quantizer is shown in the following

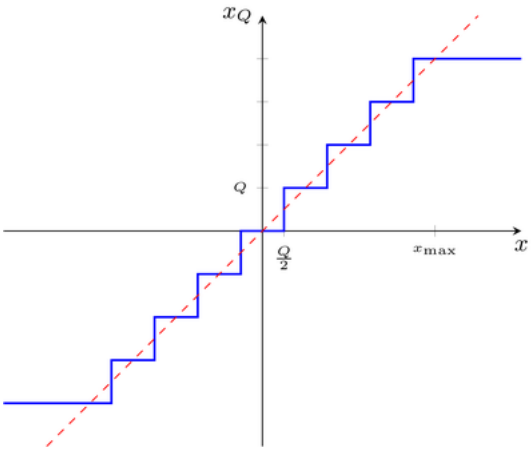


Fig. 5.3: Linear uniform mid-tread quantizer characteristic

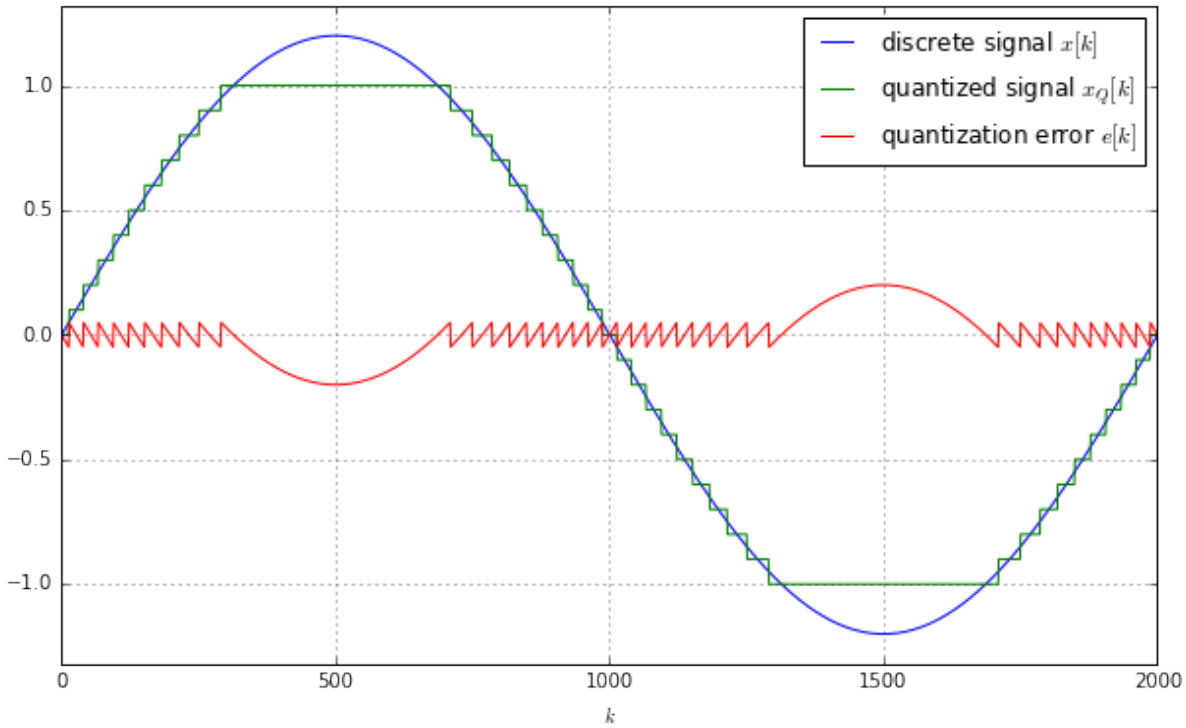
The term mid-tread is due to the fact that small values $|x[k]| < \frac{Q}{2}$ are mapped to zero.

Example

The quantization of one period of a sine signal $x[k] = A \cdot \sin[\Omega_0 k]$ by a mid-tread quantizer is simulated. A denotes the amplitude of the signal, $x_{\min} = -1$ and $x_{\max} = 1$ are the smallest and largest output values of the quantizer, respectively.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        A = 1.2  # amplitude of signal
        Q = 1/10  # quantization stepsize
        N = 2000  # number of samples
```

```
def uniform_midtread_quantizer(x, Q):
    # limiter
    x = np.copy(x)
    idx = np.where(np.abs(x) >= 1)
    x[idx] = np.sign(x[idx])
    # linear uniform quantization
    xQ = Q * np.floor(x/Q + 1/2)

    return xQ

def plot_signals(x, xQ):
    e = xQ - x
    plt.figure(figsize=(10,6))
    plt.plot(x, label=r'discrete signal $x[k]$')
    plt.plot(xQ, label=r'quantized signal $x_Q[k]$')
    plt.plot(e, label=r'quantization error $e[k]$')
    plt.xlabel(r'$k$')
    plt.axis([0, N, -1.1*A, 1.1*A])
    plt.legend()
    plt.grid()

# generate signal
x = A * np.sin(2*np.pi/N * np.arange(N))
# quantize signal
xQ = uniform_midtread_quantizer(x, Q)
# plot signals
plot_signals(x, xQ)
```



Exercise

- Change the quantization stepsize `Q` and the amplitude `A` of the signal. Which effect does this have on the quantization error?

5.2.2 Mid-Rise Chacteristic Curve

The in-/output relation of the mid-rise quantizer is given as

$$x_Q[k] = Q \cdot \Big(\underbrace{\left\lfloor \frac{x[k]}{Q} \right\rfloor}_{index} + \frac{1}{2} \Big)$$

where $\lfloor \cdot \rfloor$ denotes the floor function. The quantization characteristic of a linear uniform mid-rise quantizer is illustrated in the following

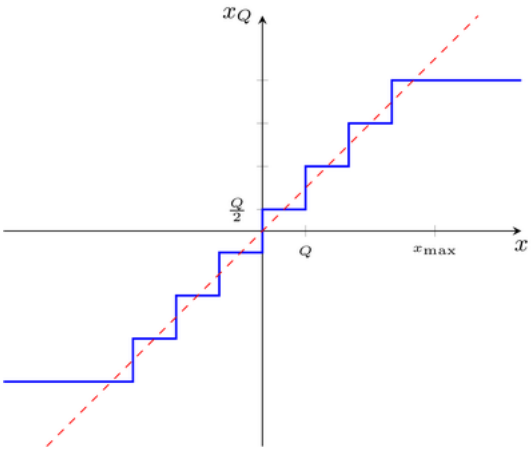


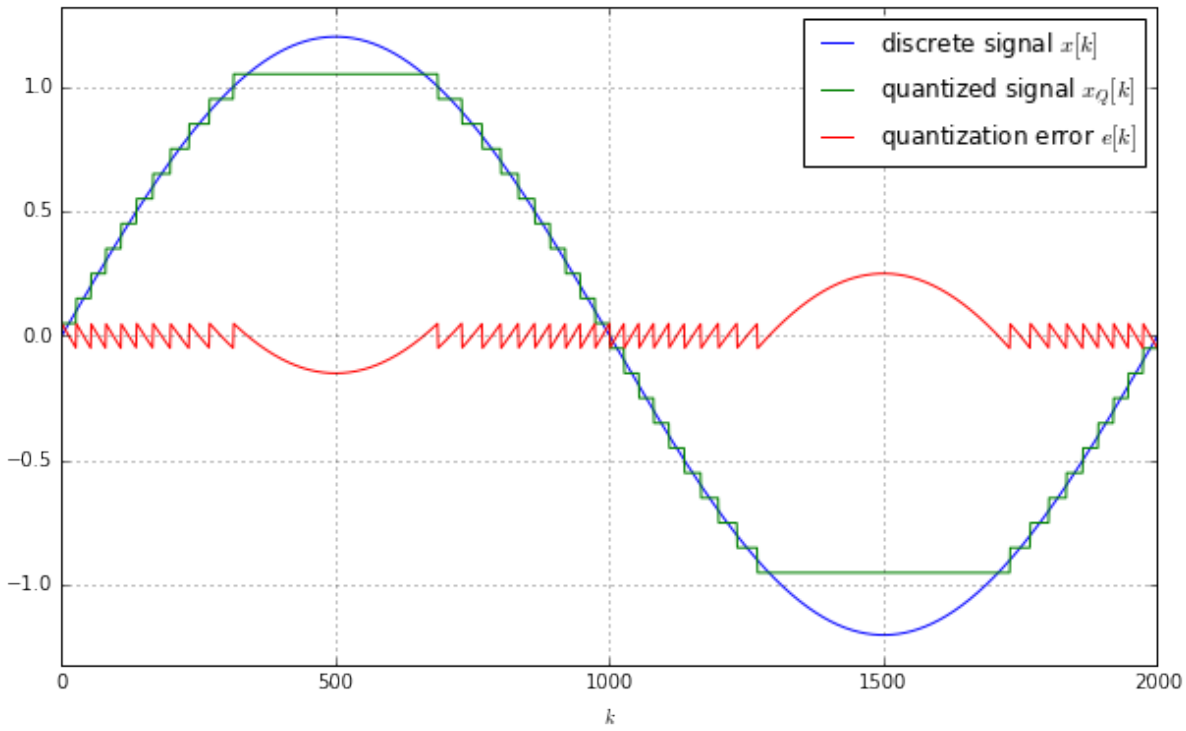
Fig. 5.4: Linear uniform mid-rise quantizer characteristic

The term mid-rise copes for the fact that $x[k] = 0$ is not mapped to zero. Small positive/negative values around zero are rather mapped to $\pm\frac{Q}{2}$.

Example

The example from above is now evaluated for the mid-rise characteristic

```
In [2]: A = 1.2  # amplitude of signal
        Q = 1/10  # quantization stepsize
        N = 2000  # number of samples

        def uniform_midrise_quantizer(x, Q):
            # limiter
            x = np.copy(x)
            idx = np.where(np.abs(x) >= 1)
            x[idx] = np.sign(x[idx])
            # linear uniform quantization
            xQ = Q * (np.floor(x/Q) + .5)

            return xQ

        # generate signal
        x = A * np.sin(2*np.pi/N * np.arange(N))
        # quantize signal
        xQ = uniform_midrise_quantizer(x, Q)
        # plot signals
        plot_signals(x, xQ)
```



Exercise

- What are the differences between the mid-tread and the mid-rise characteristic curves for the given example?

5.3 Quantization Error of a Linear Uniform Quantizer

The quantization results in two different types of distortions, as illustrated in the preceding notebook. Overload distortions are a consequence of exceeding the maximum amplitude of the quantizer. Granular distortions are a consequence of the quantization process when no clipping occurs. Various measures are used to quantify the distortions of a given quantizer. We limit ourselves to the Signal-to-Noise ratio.

5.3.1 Signal-to-Noise Ratio

A quantizer can be evaluated by its signal-to-noise ratio (SNR), which is defined as the power of the unquantized signal $x[k]$ divided by the power of the quantization error $e[k]$. Under the assumption that both signals are drawn from a zero-mean weakly stationary process, the average SNR is given as

$$SNR = 10 \cdot \log_{10} \left(\frac{\sigma_x^2}{\sigma_e^2} \right) \quad \text{in dB}$$

where σ_x^2 and σ_e^2 denote the variances of the signals $x[k]$ and $e[k]$. The statistical properties of the signal $x[k]$ and the quantization error $e[k]$ are required in order to evaluate the SNR of a quantizer.

5.3.2 Model for the Quantization Error

The statistical properties of the quantization error $e[k]$ have been derived for instance in [*Zölzer*]. We only summarize the results here. We focus on the non-clipping case first, hence on granular distortions. Here the quantization error is in general bounded $|e[k]| < \frac{Q}{2}$.

Under the assumption that the average magnitude of the input signal is much larger that the quantization step size Q, the quantization error $e[k]$ can be approximated by the following statistical model and assumptions

1. The quantization error $e[k]$ is not correlated with the input signal $x[k]$

2. The quantization error is white

$$\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sigma_e^2$$

3. The probability density function (PDF) of the quantization error is given by the zero-mean *uniform distribution*

$$p_e(\theta) = \frac{1}{Q} \cdot \mathrm{rect}\left(\frac{\theta}{Q}\right)$$

The variance of the quantization error is then *calculated from its PDF* as

$$\sigma_e^2 = \frac{Q^2}{12}$$

Let's assume that the quantization index is represented as binary or fixed-point number with w-bits. The common notation for the mid-tread quantizer is that $x_{\min}$ can be represented exactly. The quantization step is then given as

$$Q = \frac{x_{\max}}{2^{w-1} - 1} = \frac{|x_{\min}|}{2^{w-1}}$$

Using this, the variance of the quantization error can be related to the word length w

$$\sigma_e^2 = \frac{x_{\max}^2}{3 \cdot 2^{2w}}$$

From this result it can be concluded that the power of the quantization error decays by 6 dB per additional bit. This holds only for the assumptions stated above.

5.3.3 Uniformly Distributed Signal

In order to calculate the average SNR of a linear uniform quantizer, a model for the input signal $x[k]$ is required. Let's assume that the signal is modeled by a zero-mean uniform distribution

$$p_x(\theta) = \frac{1}{2x_{\max}} \mathrm{rect}\left(\frac{\theta}{2x_{\max}}\right)$$

Hence, all amplitudes between $-x_{\max}$ and $x_{\max}$ occur with the same probability. The variance of the signal is then calculated to

$$\sigma_x^2 = \frac{4x_{\max}^2}{12}$$

Introducing σ_x^2 and σ_e^2 into the definition of the SNR yields

$$SNR = 10 \cdot \log_{10}\left(2^{2w}\right) \approx 6.02\,w \quad \text{in dB}$$

This is often referred to as the 6 dB/bit rule of thumb for quantization. Note that in the derivation above it has been assumed that the signal $x[k]$ uses the full amplitude range of the quantizer. If this is not the case, the SNR will be lower since σ_x^2 is lower.

Example

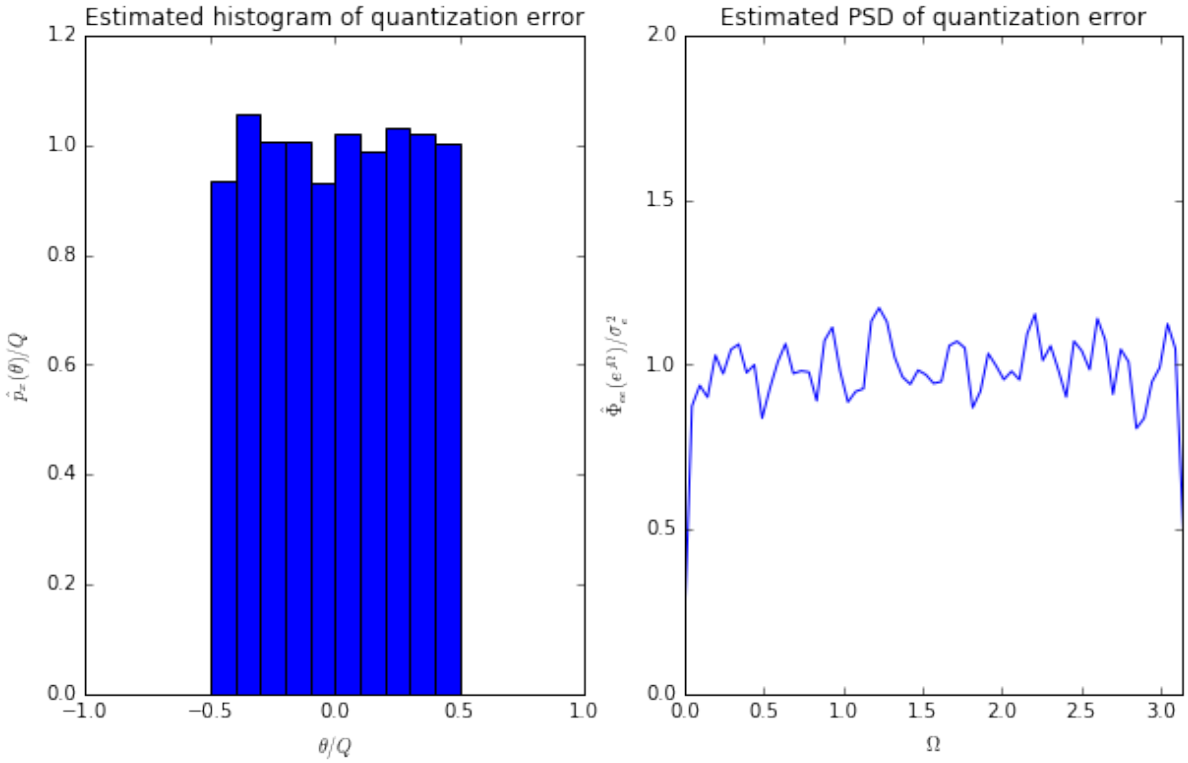
In this example the linear uniform quantization of a random signal drawn from a uniform distribution is evaluated. The amplitude range of the quantizer is $x_{\min} = -1$ and $x_{\max} = 1 - Q$.

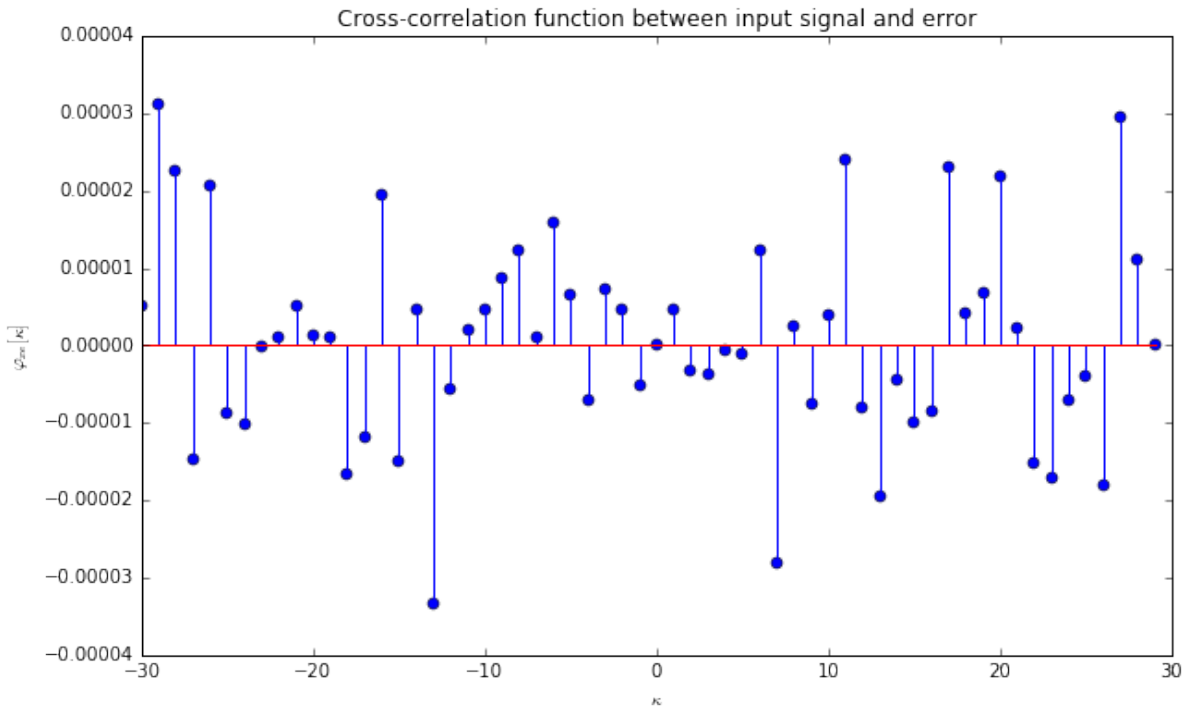
```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig
```

```
w = 8  # wordlength of the quantized signal
A = 1  # amplitude of input signal
N = 8192  # number of samples
K = 30  # maximum lag for cross-correlation


def uniform_midtread_quantizer(x, Q):
    # limiter
    x = np.copy(x)
    idx = np.where(x <= -1)
    x[idx] = -1
    idx = np.where(x > 1 - Q)
    x[idx] = 1 - Q
    # linear uniform quantization
    xQ = Q * np.floor(x/Q + 1/2)

    return xQ

def analyze_quantizer(x, e):
    # estimated PDF of error signal
    pe, bins = np.histogram(e, bins=20, normed=True, range=(-Q, Q))
    # estimate cross-correlation between input and error
    ccf = 1/len(x) * np.correlate(x, e, mode='full')
    # estimate PSD of error signal
    nf, Pee = sig.welch(e, nperseg=128)
    # estimate SNR
    SNR = 10*np.log10((np.var(x)/np.var(e)))
    print('SNR = %f in dB' %SNR)

    # plot statistical properties of error signal
    plt.figure(figsize=(10,6))

    plt.subplot(121)
    plt.bar(bins[:-1]/Q, pe*Q, width = 2/len(pe))
    plt.title('Estimated histogram of quantization error')
    plt.xlabel(r'$\theta / Q$')
    plt.ylabel(r'$\hat{p}_x(\theta) / Q$')
    plt.axis([-1, 1, 0, 1.2])

    plt.subplot(122)
    plt.plot(nf*2*np.pi, Pee*6/Q**2)
    plt.title('Estimated PSD of quantization error')
    plt.xlabel(r'$\Omega$')
    plt.ylabel(r'$\hat{\Phi}_{ee}(e^{j \Omega}) / \sigma_e^2$')
    plt.axis([0, np.pi, 0, 2]);

    plt.figure(figsize=(10,6))
    ccf = ccf[N-K-1:N+K-1]
    kappa = np.arange(-len(ccf)//2,len(ccf)//2)
    plt.stem(kappa, ccf)
    plt.title('Cross-correlation function between input signal and error')
    plt.xlabel(r'$\kappa$')
    plt.ylabel(r'$\varphi_{xe}[\kappa]$')


# quantization step
Q = 1/(2**(w-1))
# compute input signal
```

```
    x = np.random.uniform(size=N, low=-A, high=(A-Q))
    # quantize signal
    xQ = uniform_midtread_quantizer(x, Q)
    e = xQ - x
    # analyze quantizer
    analyze_quantizer(x, e)
```

```
SNR = 48.128585 in dB
```





Exercise

- Change the number of bits `w` and check if the derived SNR holds

- Change the amplitude `A` of the input signal. What happens if you make the amplitude very small? Why?

5.3.4 Harmonic Signal

For a harmonic input signal $x[k] = x_{\max} \cdot \cos[\Omega_0 k]$ the variance σ_x^2 is given by its squared root mean square (RMS) value

$$\sigma_x^2 = \frac{x_{\max}^2}{2}$$

Introducing this into the definition of the SNR together with the variance σ_e^2 of the quantization error yields

$$SNR = 10 \cdot \log_{10}\left(2^{2w} \cdot \frac{3}{2}\right) \approx 6.02\,w + 1.76 \quad \text{in dB}$$

The gain of 1.76 dB with respect to the case of a uniformly distributed input signal is due to the fact that the amplitude distribution of a harmonic signal is not uniform

$$p_x(\theta) = \frac{1}{\pi\sqrt{1 - (\frac{\theta}{x_{\max}})^2}}$$

for $|\theta| < x_{\max}$. High amplitudes are more likely to occur. The relative power of the quantization error is lower for higher amplitudes which results in an increase of the average SNR.

5.3.5 Normally Distributed Signal

So far, we did not consider clipping of the input signal $x[k]$, e.g. by ensuring that its minimum/maximum values do not exceed the limits of the quantizer. However, this cannot always be ensured for practical signals. Moreover, many practical signals cannot be modeled as a uniform distribution. For instance a *normally distributed* random signal exceeds a given maximum value with non-zero probability. Hence, clipping will occur for such an input signal. Clipping results in overload distortions whose amplitude can be much higher that $\frac{Q}{2}$. For the overall average SNR both granular and overload distortions have to be included.

For a normally distributed signal with a given probability that clipping occurs $\Pr\{|x[k]| > x_{\max}\} = 10^{-5}$ the SNR can be calculated to [*Zölzer*]

$$SNR \approx 6.02\,w - 8.5 \quad \text{in dB}$$

The reduction of the SNR by 8.5 dB results from the fact that small signal values are more likely to occur for a normally distributed signal. The relative quantization error for small signals is higher, which results in a lower average SNR. Overload distortions due to clipping result in a further reduction of the average SNR.

5.3.6 Laplace Distributed Signal

The *Laplace distribution* is a commonly applied model for speech and music signals. As for the normal distribution, clipping will occur with a non-zero probability. For a Laplace distributed signal with a given probability that clipping occurs $\Pr\{|x[k]| > x_{\max}\} = 10^{-4}$ the SNR can be calculated to [*Vary et al.*]

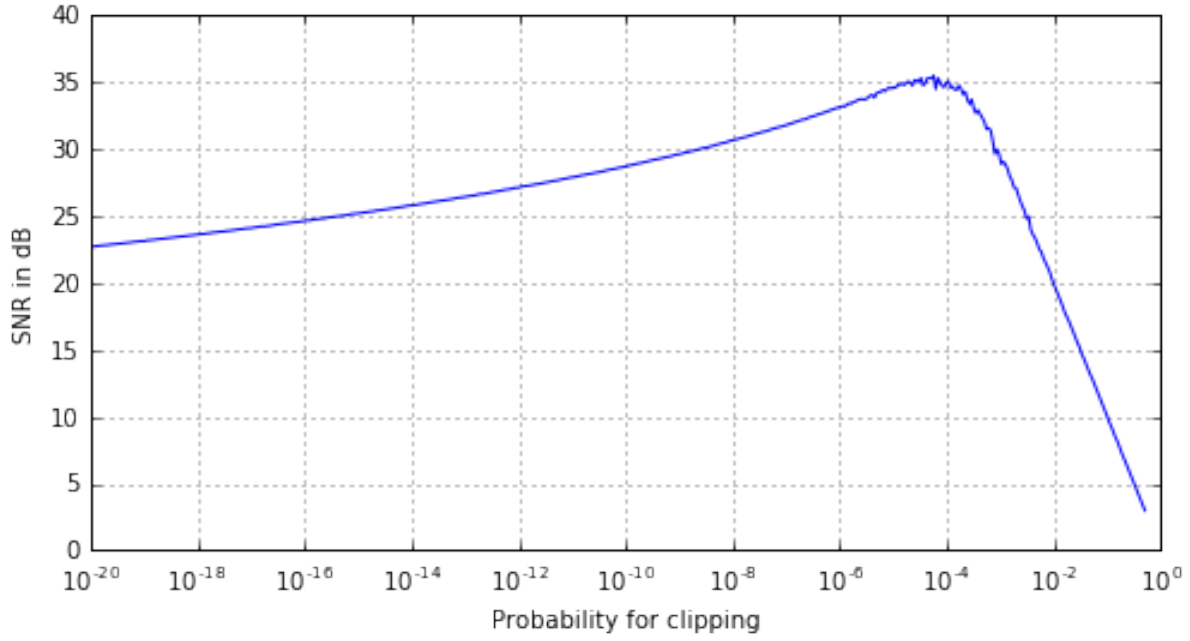
$$SNR \approx 6.02\,w - 9 \quad \text{in dB}$$

Even though the probability of clipping is higher as for the normally distributed signal above, the SNR is in the same range. The reason for this is, that the Laplace distribution features low signal values with a higher and large values with a lower probability in comparison to the normal distribution.

Example

The following example evaluates the SNR of a linear uniform quantizer with $w = 8$ for a Laplace distributed signal $x[k]$. The SNR is computed for various probabilities that clipping occurs.

```
In [2]: w = 8  # wordlength of the quantized signal
        Pc = np.logspace(-20, np.log10(.5), num=500)  # probabilities for clipping
        N = int(1e6)  # number of samples


        def compute_SNR(Pc):
            # compute input signal
            sigma_x = - np.sqrt(2) / np.log(Pc)
            x = np.random.laplace(size=N, scale=sigma_x/np.sqrt(2) )
            # quantize signal
            xQ = uniform_midtread_quantizer(x, Q)
            e = xQ - x
            # compute SNR
            SNR = 10*np.log10((np.var(x)/np.var(e)))

            return SNR


        # quantization step
        Q = 1/(2**(w-1))
        # compute SNR for given probabilities
        SNR = [compute_SNR(P) for P in Pc]

        # plot results
        plt.figure(figsize=(8,4))
        plt.semilogx(Pc, SNR)
        plt.xlabel('Probability for clipping')
        plt.ylabel('SNR in dB')
        plt.grid()
```



Exercise

- Can you explain the specific shape of the curve? What effect dominates for clipping probabilities below/above the maximum?

5.4 Requantization of a Speech Signal

The following example illustrates the requantization of a speech signal. The signal was originally recorded with a wordlength of $w = 16$ bits. It is requantized with a *uniform mid-tread quantizer* to various wordlengths. The SNR is computed and a portion of the (quantized) signal is plotted. It is further possible to listen to the requantized signal and the quantization error.

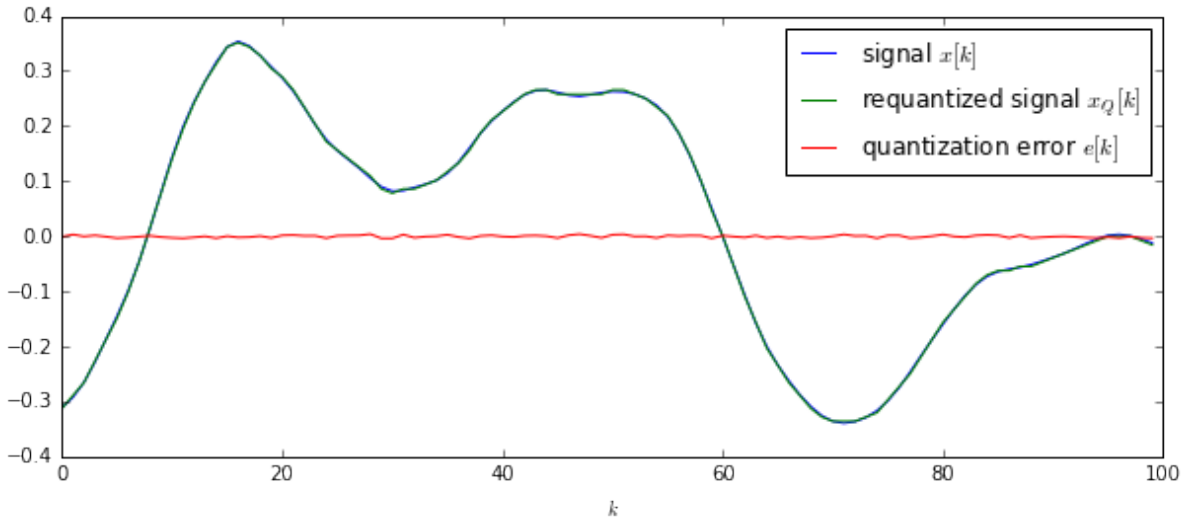
```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import soundfile as sf

        idx = 130000  # index to start plotting

        def uniform_midtread_quantizer(x, w):
            # quantization step
            Q = 1/(2**(w-1))
            # limiter
            x = np.copy(x)
            idx = np.where(x <= -1)
            x[idx] = -1
            idx = np.where(x > 1 - Q)
            x[idx] = 1 - Q
            # linear uniform quantization
            xQ = Q * np.floor(x/Q + 1/2)

            return xQ

        def evaluate_requantization(x, xQ):
            e = xQ - x
            # SNR
            SNR = 10*np.log10(np.var(x)/np.var(e))
            print('SNR: %f dB'%SNR)
            # plot signals
            plt.figure(figsize=(10, 4))
            plt.plot(x[idx:idx+100], label=r'signal $x[k]$')
            plt.plot(xQ[idx:idx+100], label=r'requantized signal $x_Q[k]$')
            plt.plot(e[idx:idx+100], label=r'quantization error $e[k]$')
            plt.xlabel(r'$k$')
            plt.legend()
            # normalize error
            e = .2 * e / np.max(np.abs(e))
            return e

        # load speech sample
        x, fs = sf.read('../data/speech.wav')
        x = x/np.max(np.abs(x))
```

Original Signal Your browser does not support the audio element. ../data/speech.wav

5.4.1 Requantization to 8 bit

```
In [2]: xQ = uniform_midtread_quantizer(x, 8)
        e = evaluate_requantization(x, xQ)
        sf.write('speech_8bit.wav', xQ, fs)
        sf.write('speech_8bit_error.wav', e, fs)
```

```
SNR: 34.021487 dB
```



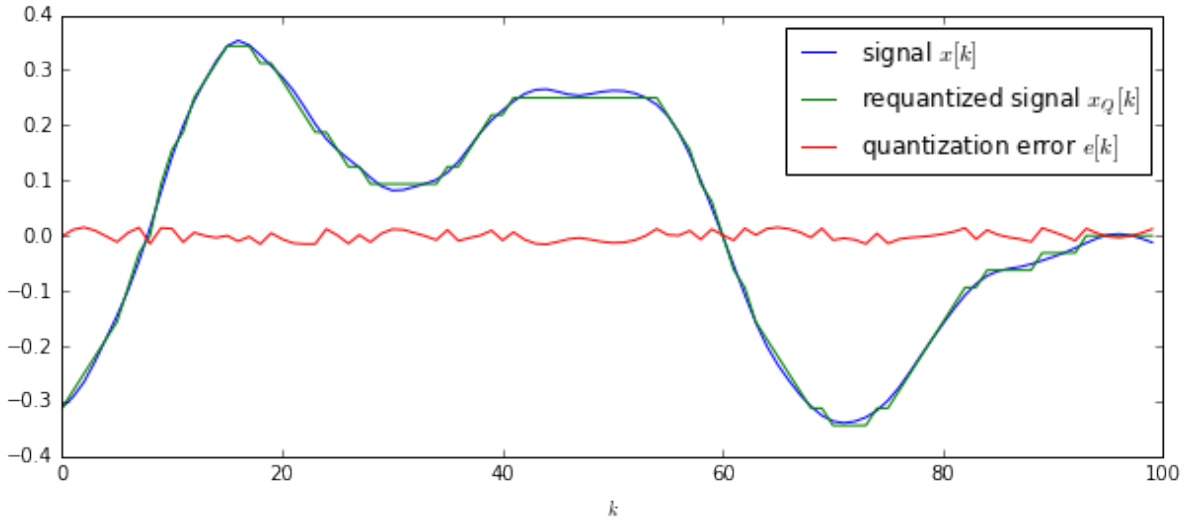
Requantized Signal Your browser does not support the audio element. speech_8bit.wav

Quantization Error Your browser does not support the audio element. speech_8bit_error.wav

5.4.2 Requantization to 6 bit

```
In [3]: xQ = uniform_midtread_quantizer(x, 6)
        e = evaluate_requantization(x, xQ)
        sf.write('speech_6bit.wav', xQ, fs)
        sf.write('speech_6bit_error.wav', e, fs)
```

```
SNR: 22.889593 dB
```



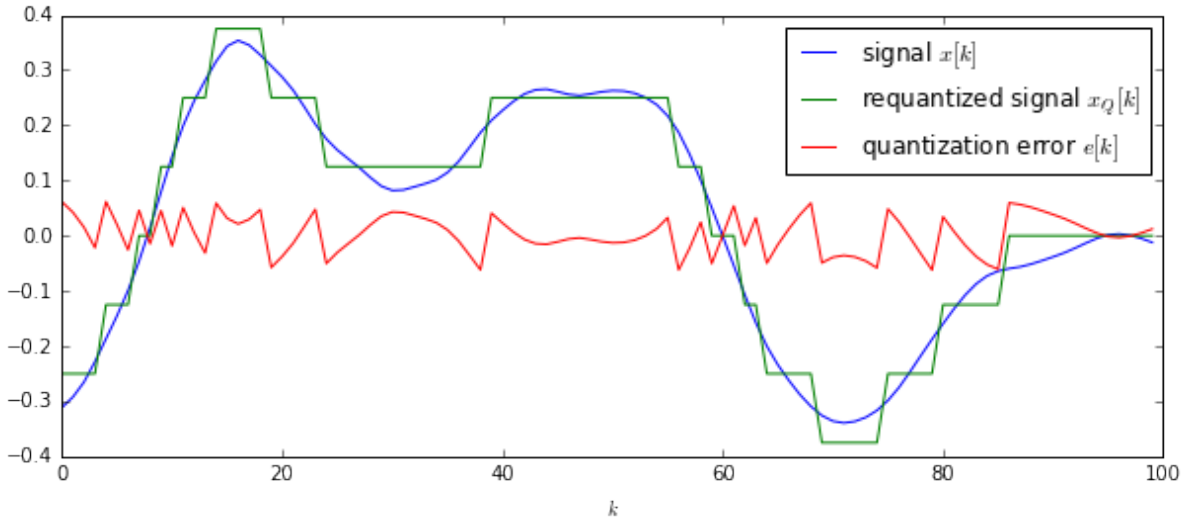
Requantized Signal Your browser does not support the audio element. speech_6bit.wav

Quantization Error Your browser does not support the audio element. speech_6bit_error.wav

5.4.3 Requantization to 4 bit

```
In [4]: xQ = uniform_midtread_quantizer(x, 4)
        e = evaluate_requantization(x, xQ)
        sf.write('speech_4bit.wav', xQ, fs)
        sf.write('speech_4bit_error.wav', e, fs)
```

```
SNR: 11.713678 dB
```



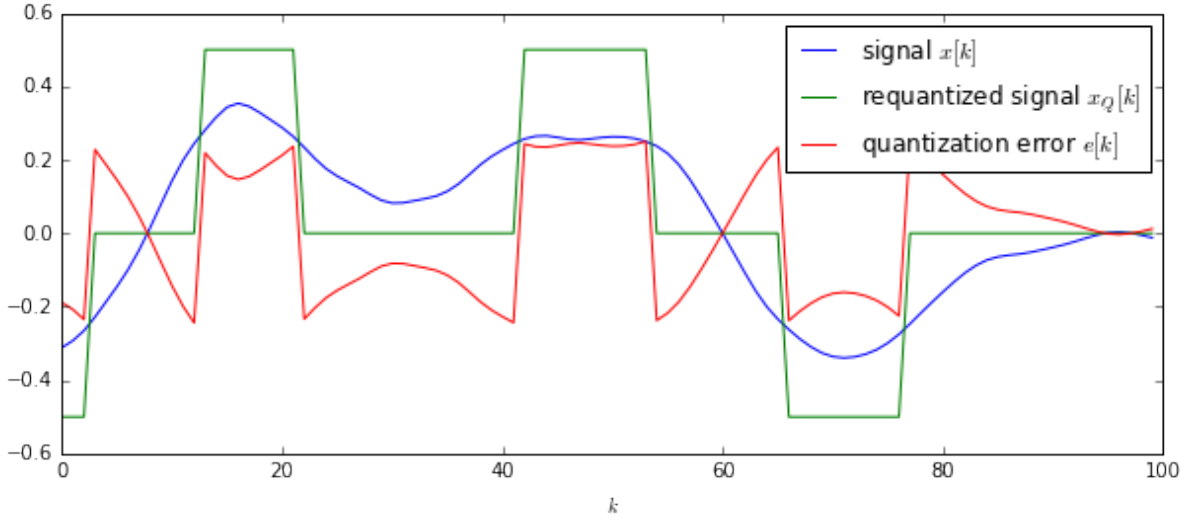
Requantized Signal Your browser does not support the audio element. speech_4bit.wav

Quantization Error Your browser does not support the audio element. speech_4bit_error.wav

5.4.4 Requantization to 2 bit

```
In [5]: xQ = uniform_midtread_quantizer(x, 2)
        e = evaluate_requantization(x, xQ)
        sf.write('speech_2bit.wav', xQ, fs)
        sf.write('speech_2bit_error.wav', e, fs)
```

```
SNR: 2.428364 dB
```



Requantized Signal Your browser does not support the audio element. speech_2bit.wav

Quantization Error Your browser does not support the audio element. speech_2bit_error.wav

5.5 Spectral Shaping of the Quantization Noise

The quantized signal as the output of a quantizer can be expressed with the quantization error $e[k]$ as

$$x_Q[k] = \mathcal{Q}\{x[k]\} = x[k] + e[k]$$

According to the *introduced model*, the quantization noise can be modeled as uniformly distributed white noise. Hence, the noise is distributed over the entire frequency range. The basic concept of noise shaping is a feedback of the quantization error to the input of the quantizer. This way the spectral characteristics of the quantization noise can be changed, i.e. spectrally shaped. Introducing a generic filter $h[k]$ into the feedback loop yields the following structure

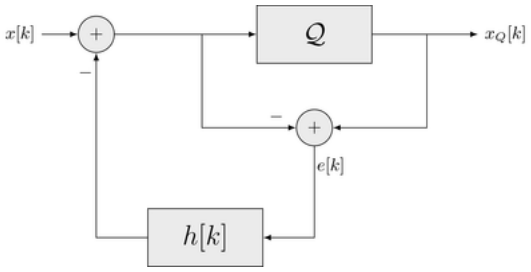


Fig. 5.5: Feedback structure for noise shaping

The quantized signal can be deduced from the block diagram above as

$$x_Q[k] = \mathcal{Q}\{x[k] - e[k] * h[k]\} = x[k] + e[k] - e[k] * h[k]$$

where the additive noise model from above has been introduced and it has been assumed that the impulse response $h[k]$ is normalized such that the magnitude of $e[k] * h[k]$ is below the quantization step Q. The overall quantization error is then

$$e_H[k] = x_Q[k] - x[k] = e[k] * (\delta[k] - h[k])$$

The power spectral density (PSD) of the quantization with noise shaping is calculated to

$$\Phi_{e_H e_H}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot \left|1 - H(\mathrm{e}^{\,\mathrm{j}\,\Omega})\right|^2$$

Hence the PSD $\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the quantizer without noise shaping is weighted by $|1 - H(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2$. Noise shaping allows a spectral modification of the quantization error. The desired shaping depends on the application scenario. For some applications, high-frequency noise is less disturbing as low-frequency noise.

5.5.1 Example

If the feedback of the error signal is delayed by one sample we get with $h[k] = \delta[k-1]$

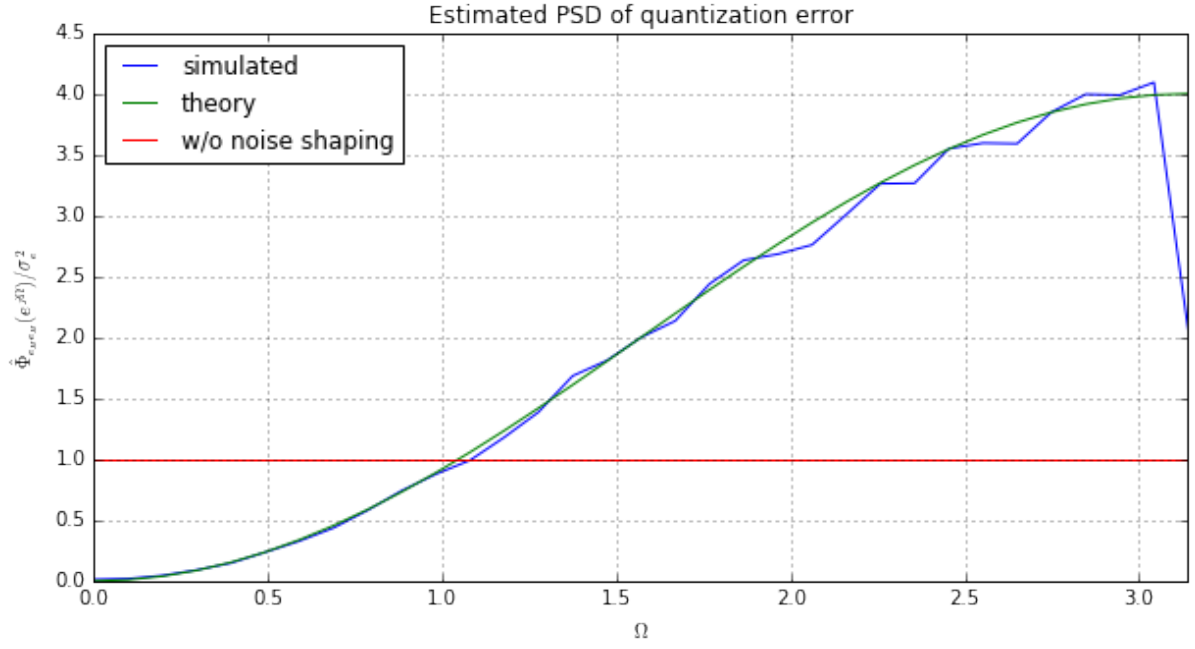
$$\Phi_{e_H e_H}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot \left|1 - \mathrm{e}^{\,-\mathrm{j}\,\Omega}\right|^2$$

For linear uniform quantization $\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sigma_e^2$ is constant. Hence, the spectral shaping constitutes a high-pass characteristic of first order. The following simulation evaluates a noise shaping quantizer of first order.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig

        w = 8  # wordlength of the quantized signal
        A = 1  # amplitude of input signal
        N = 32768  # number of samples


        def uniform_midtread_quantizer_w_ns(x, Q):
            # limiter
            x = np.copy(x)
            idx = np.where(x <= -1)
            x[idx] = -1
```

```
        idx = np.where(x > 1 - Q)
        x[idx] = 1 - Q
        # linear uniform quantization with noise shaping
        xQ = Q * np.floor(x/Q + 1/2)
        e = xQ - x
        xQ = xQ - np.concatenate(([0], e[0:-1]))

        return xQ[1:]


    # quantization step
    Q = 1/(2**(w-1))
    # compute input signal
    x = np.random.uniform(size=N, low=-A, high=(A-Q))
    # quantize signal
    xQ = uniform_midtread_quantizer_w_ns(x, Q)
    e = xQ - x[1:]
    # estimate PSD of error signal
    nf, Pee = sig.welch(e, nperseg=64)
    # estimate SNR
    SNR = 10*np.log10((np.var(x)/np.var(e)))
    print('SNR = %f in dB' %SNR)


    plt.figure(figsize=(10,5))
    Om = nf*2*np.pi
    plt.plot(Om, Pee*6/Q**2, label='simulated')
    plt.plot(Om, np.abs(1 - np.exp(-1j*Om))**2, label='theory')
    plt.plot(Om, np.ones(Om.shape), label='w/o noise shaping')
    plt.title('Estimated PSD of quantization error')
    plt.xlabel(r'$\Omega$')
    plt.ylabel(r'$\hat{\Phi}_{e_H e_H}(e^{j \Omega}) / \sigma_e^2$')
    plt.axis([0, np.pi, 0, 4.5]);
    plt.legend(loc='upper left')
    plt.grid()
```

```
SNR = 45.128560 in dB
```



Exercise

- The overall average SNR is lower than for the quantizer without noise shaping. Why?

5.6 Oversampling

Oversampling is a technique which is applied in analog-to-digital converters to lower the average power of the quantization error. It requires a joint consideration of sampling and quantization.

5.6.1 Ideal Analog-to-Digital Conversion

Let's consider the ideal sampling of a signal followed by its quantization, as given by the following block diagram

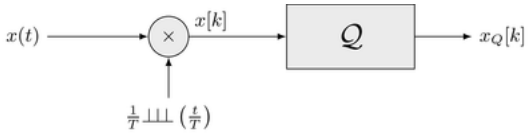


Fig. 5.6: Ideal analog to digital conversion of a signal

Ideal sampling is modeled by multiplying the continuous signal $x(t)$ with a series of equidistant Dirac functions, resulting in the discrete signal $x[k] = x(kT)$ where T denotes the sampling interval. The discrete signal $x[k]$ is then quantized. The output of the ideal analog-to-digital converter is the quantized discrete signal $x_\mathrm{Q}[k]$.

5.6.2 Nyquist Sampling

Sampling of the continuous signal $x(t)$ leads to repetitions of the spectrum $X(j\omega) = \mathcal{F}\{x(t)\}$ at multiples of $\omega_\mathrm{S} = \frac{2\pi}{T}$. We limit ourselves to a continuous real-valued $x(t) \in \mathbb{R}$, band-limted signal $|X(j\omega)| = 0$ for $|\omega| > \omega_\mathrm{C}$ where ω_C denotes its cut-off frequency. The spectral repetitions due to sampling do not overlap if the sampling theorem $\omega_\mathrm{S} \geq 2 \cdot \omega_\mathrm{C}$ is fulfilled. In the case of Nyquist (critical) sampling, the sampling frequency is chosen as $\omega_\mathrm{S} = 2 \cdot \omega_\mathrm{C}$.

5.6.3 Oversampling

The basic idea of oversampling is to sample the input signal at frequencies which are significantly higher than the Nyquist criterion dictates. After quantization, the signal is low-pass filtered by a discrete filter $H_\mathrm{LP}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and resampled back to the Nyquist rate. In order to avoid aliasing due to the resampling this filter has to be chosen as an ideal low-pass

$$H_\mathrm{LP}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathrm{rect}\left(\frac{\Omega}{2\,\Omega_\mathrm{C}}\right)$$

where $\Omega_\mathrm{C} = \omega_\mathrm{C} \cdot T$. For an oversampling of factor $L \in \mathbb{Z}$ we have $\omega_\mathrm{S} = L \cdot 2\omega_\mathrm{C}$. For this case, the resampling can be realized by keeping only every L-th sample which is known as decimation. The following block diagram illustrates the building blocks of oversampled digital-to-analog conversion, $\downarrow L$ denotes decimation by a factor of L

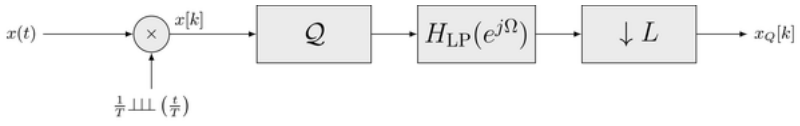


Fig. 5.7: Oversampling ideal analog to digital converter

In order the conclude on the benefits of oversampling we have to derive the average power of the overall quantization error. According to our *model of the quantization error*, the quantization error $e[k]$ can be modeled as uniformly distributed white noise. Its power spectral density (PSD) is given as

$$\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{Q^2}{12}$$

where Q denotes the quantization step. Before the discrete low-pass filter $H_{\text{LP}}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$, the power of the quantization error is uniformly distributed over the entire frequency range $-\pi < \Omega \leq \pi$. However, after the ideal low-pass filter the frequency range is limited to $-\frac{\pi}{L} < \Omega \leq \frac{\pi}{L}$. The average power of the quantization error is then given as

$$\sigma_{e,\text{LP}}^2 = \frac{1}{2\pi} \int\limits_{-\frac{\pi}{L}}^{\frac{\pi}{L}} \Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \,\mathrm{d}\Omega = \frac{1}{L} \cdot \frac{Q^2}{12}$$

The average power σ_x^2 of the sampled signal $x[k]$ is not affected, since the cutoff frequency of the low-pass filter has been chosen as the upper frequency limit ω_{C} of the input signal $x(t)$.

In order to calculate the SNR of the oversampled analog-to-digital converter we assume that the input signal is drawn from a uniformly distributed zero-mean random process with $|x[k]| < x_{\max}$. With the results from our discussion of *linear uniform quantization* and $\sigma_{e,\text{LP}}^2$ from above we get

$$SNR = 10 \cdot \log_{10}\left(2^{2w}\right) + 10 \cdot \log_{10}(L) \approx 6.02\,w + 10 \cdot \log_{10}(L) \quad \text{in dB}$$

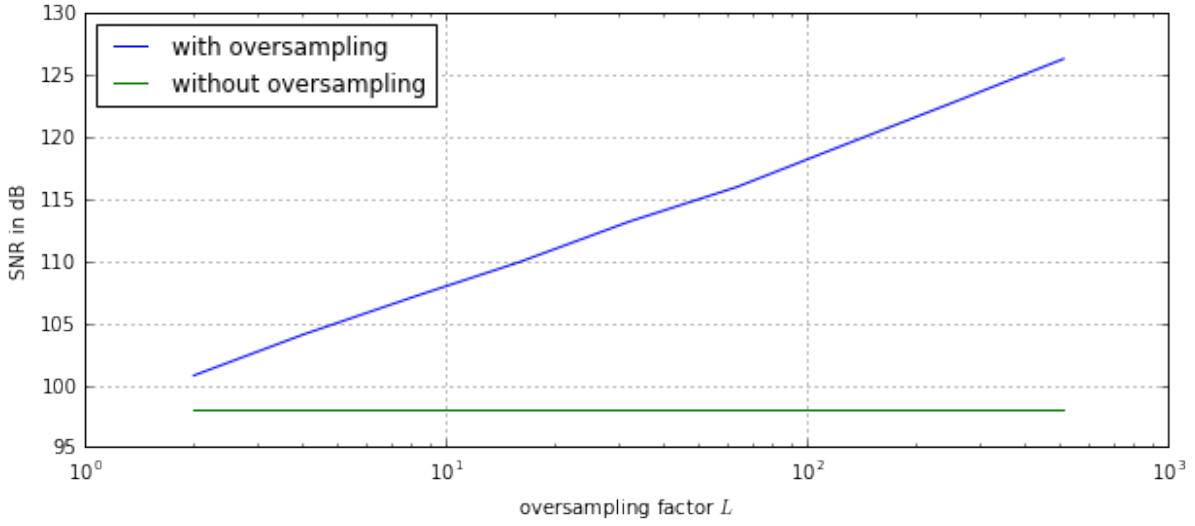
where w denotes the number of bits used for a binary representation of the quantization index. Hence, oversampling by a factor of L brings a plus of $10 \cdot \log_{10}(L)$ dB in terms of SNR. For instance, an oversampling by a factor of $L = 4$ results in a SNR which is approximately 6 dB higher. For equal SNR the quantization step Q can be chosen larger. In terms the of wordlength of a quantizer this accounts to a reduction by one bit. Consequently, there is a trade-off between accuracy of the quantizer and its sampling frequency.

5.6.4 Example

The following numerical simulation illustrates the benefit in terms of SNR for an oversampled linear uniform quantizer with $w = 16$ for the quantization of the harmonic signal $x[k] = \cos[\Omega_0 k]$.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig

        w = 16  # wordlength of the quantized signal
        L = 2**np.arange(1,10)  # oversampling factors

        N = 8192  # length of signals
        Om0 = 100*2*np.pi/N  # frequency of harmonic signal
        Q = 1/(2**(w-1))  # quantization step


        def uniform_midtread_quantizer(x, Q):
            # limiter
            x = np.copy(x)
            idx = np.where(x <= -1)
            x[idx] = -1
            idx = np.where(x > 1 - Q)
            x[idx] = 1 - Q
            # linear uniform quantization
            xQ = Q * np.floor(x/Q + 1/2)

            return xQ
```

```
def SNR_oversampled_ADC(L):
    x = (1-Q)*np.cos(Om0*np.arange(N))
    xu = (1-Q)*np.cos(Om0*np.arange(N*L)/L)
    # quantize signal
    xQu = uniform_midtread_quantizer(xu, Q)
    # low-pass filtering and decimation
    xQ = sig.resample(xQu, N)
    # estimate SNR
    e = xQ - x

    return 10*np.log10((np.var(x)/np.var(e)))


# compute SNR for oversampled ADC
SNR = [SNR_oversampled_ADC(l) for l in L]

# plot result
plt.figure(figsize=(10, 4))
plt.semilogx(L, SNR, label='with oversampling')
plt.plot(L, (6.02*w+1.76)*np.ones(L.shape), label='without oversampling' )
plt.xlabel(r'oversampling factor $L$')
plt.ylabel(r'SNR in dB')
plt.legend(loc='upper left')
plt.grid()
```



Exercise

- What SNR can be achieved for an oversampling factor of $L = 16$?
- By how many bits could the word length w be reduced in order to gain the same SNR as without oversampling?

5.6.5 Anti-Aliasing Filter

Besides an increased SNR, oversampling has also another benefit. In order to ensure that the input signal $x(t)$ is band-limited before sampling, a low-pass filter $H_{\mathrm{LP}}(\mathrm{j}\,\omega)$ is applied in typical analog-to-digital converters. This is illustrated in the following

The filter $H_{\mathrm{LP}}(\mathrm{j}\,\omega)$ is also known as anti-aliasing filter. The ideal low-pass filter is given as $H_{\mathrm{LP}}(\mathrm{j}\,\omega) = \mathrm{rect}\left(\frac{\omega}{\omega_\mathrm{S}}\right)$. The ideal $H_{\mathrm{LP}}(\mathrm{j}\,\omega)$ can only be approximated in the analog domain. Since the sampling rate is higher than the

Nyquist rate, there is no need for a steep slope of the filter in order to avoid aliasing. However, the pass-band of the filter within $|\omega| < |\omega_C|$ has to be flat.

Before decimation, the discrete filter $H_{\text{LP}}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ has to remove the spectral contributions that may lead to aliasing. However, a discrete filter $H_{\text{LP}}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ with steep slope can be realized much easier than in the analog domain.

5.7 Non-Linear Requantization of a Speech Signal

Speech signals have a *non-uniform amplitude distribution* which is often modeled by the Laplace distribution. Linear uniform quantization is not optimal for speech signals, since small signal amplitudes are more likely than higher ones. This motivates a non-linear quantization scheme, where the signal is companded before linear quantization and expanded afterwards.

The following example illustrates the A-law companding used in European telephone networks. The signal was originally recorded with a wordlength of $w = 16$ bits using linear uniform quantization. First the A-law compansion is applied, then quantization by a linear uniform quantizer with a wordlength of $w = 8$ bits. For a sampling rate of $f_s = 8$ kHz this results in a bit-rate of 64 kbits/s used in the backbone of many telephone networks.

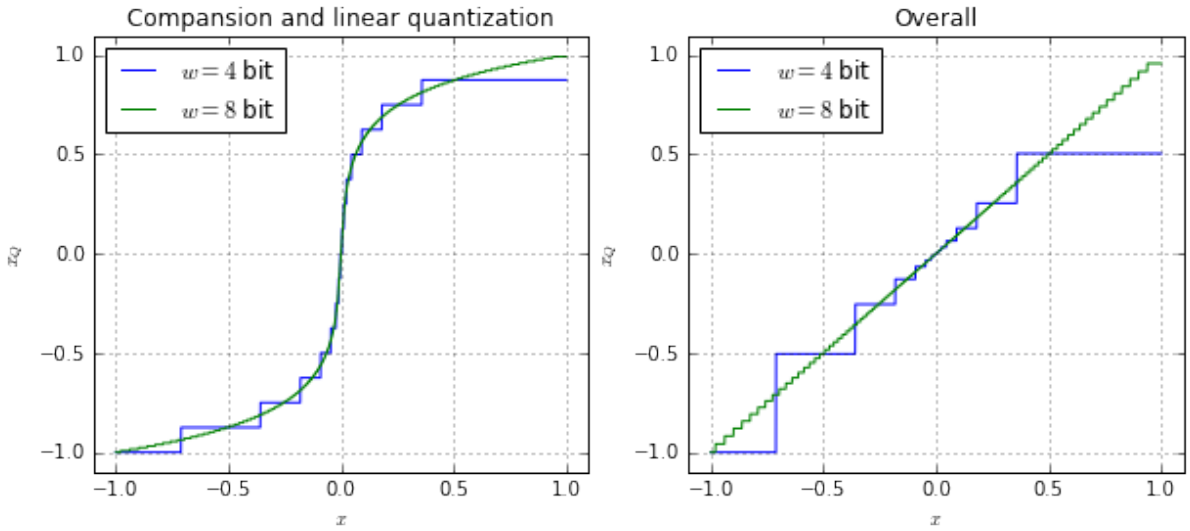
```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import soundfile as sf


        def A_law_compander(x):
            A = 87.6
            y = np.zeros_like(x)
            idx = np.where(np.abs(x) < 1/A)
            y[idx] = A*np.abs(x[idx]) / (1 + np.log(A))
            idx = np.where(np.abs(x) >= 1/A)
            y[idx] = (1 + np.log(A*np.abs(x[idx]))) / (1 + np.log(A))

            return np.sign(x)*y

        def A_law_expander(y):
            A = 87.6
            x = np.zeros_like(y)
            idx = np.where(np.abs(y) < 1/(1+np.log(A)))
            x[idx] = np.abs(y[idx])*(1+np.log(A)) / A
            idx = np.where(np.abs(y) >= 1/(1+np.log(A)))
            x[idx] = np.exp(np.abs(y[idx])*(1+np.log(A))-1)/A

            return np.sign(y)*x

        def uniform_midtread_quantizer(x, w):
            # quantization step
            Q = 1/(2**(w-1))
            # limiter
            x = np.copy(x)
            idx = np.where(x <= -1)
            x[idx] = -1
            idx = np.where(x > 1 - Q)
            x[idx] = 1 - Q
            # linear uniform quantization
            xQ = Q * np.floor(x/Q + 1/2)

            return xQ
```

```
        def evaluate_requantization(x, xQ):
            e = xQ - x
            # SNR
            SNR = 10*np.log10(np.var(x)/np.var(e))
            print('SNR: %f dB'%SNR)
            # normalize error
            e = .2 * e / np.max(np.abs(e))
            return e
```

5.7.1 Quantization Characteristic

Lets first take a look at the non-linear characteristic of the A-law requantizer. The left plot shows the characteristic of the A-law companding and linear-quantization. The right plot shows the overall characteristic for companding, linear quantization and expansion.

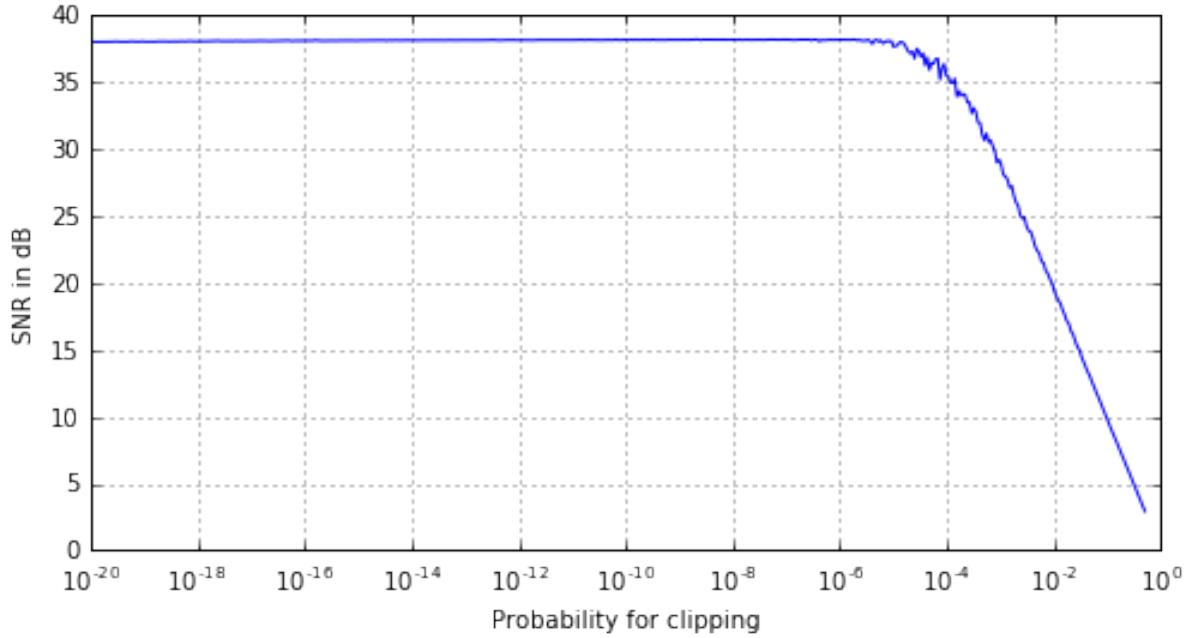
```
In [2]: x = np.linspace(-1, 1, 2**16)
        y = A_law_compander(x)
        yQ4 = uniform_midtread_quantizer(y, 4)
        yQ8 = uniform_midtread_quantizer(y, 8)
        xQ4 = A_law_expander(yQ4)
        xQ8 = A_law_expander(yQ8)

        plt.figure(figsize=(10, 4))

        plt.subplot(121)
        plt.plot(x, yQ4, label=r'$w=4$ bit')
        plt.plot(x, yQ8, label=r'$w=8$ bit')
        plt.title('Compansion and linear quantization')
        plt.xlabel(r'$x$')
        plt.ylabel(r'$x_Q$')
        plt.legend(loc=2)
        plt.axis([-1.1, 1.1, -1.1, 1.1])
        plt.grid()

        plt.subplot(122)
        plt.plot(x, xQ4, label=r'$w=4$ bit')
        plt.plot(x, xQ8, label=r'$w=8$ bit')
        plt.title('Overall')
        plt.xlabel(r'$x$')
        plt.ylabel(r'$x_Q$')
        plt.legend(loc=2)
        plt.axis([-1.1, 1.1, -1.1, 1.1])
        plt.grid()
```



5.7.2 Signal-to-Noise Ratio

Now the signal-to-noise ratio (SNR) is computed for a Laplace distributed signal for various probabilities $\Pr\{|x[k]| > x_{\max}\}$ that clipping occurs. The results show that the non-linear quantization scheme provides a constant SNR over a wide range of signal amplitudes. The SNR is additional higher as for *linear quantization* of a Laplace distributed signal.

```
In [3]: w = 8  # wordlength of the quantized signal
        Pc = np.logspace(-20, np.log10(.5), num=500)  # probabilities for clipping
        N = int(1e6)  # number of samples


        def compute_SNR(Pc):
            # compute input signal
            sigma_x = - np.sqrt(2) / np.log(Pc)
            x = np.random.laplace(size=N, scale=sigma_x/np.sqrt(2) )
            # quantize signal
            y = A_law_compander(x)
            yQ = uniform_midtread_quantizer(y, 8)
            xQ = A_law_expander(yQ)
            e = xQ - x
            # compute SNR
            SNR = 10*np.log10((np.var(x)/np.var(e)))

            return SNR


        # quantization step
        Q = 1/(2**(w-1))
        # compute SNR for given probabilities
        SNR = [compute_SNR(P) for P in Pc]

        # plot results
        plt.figure(figsize=(8,4))
        plt.semilogx(Pc, SNR)
        plt.xlabel('Probability for clipping')
        plt.ylabel('SNR in dB')
        plt.grid()
```



5.7.3 Requantization of a Speech Sample

Finally we requantize a speech sample with a linear and the A-law quantization scheme. Listen to the samples!

```
In [5]: # load speech sample
        x, fs = sf.read('../data/speech_8k.wav')
        x = x/np.max(np.abs(x))

        # linear quantization
        xQ = uniform_midtread_quantizer(x, 8)
        e = evaluate_requantization(x, xQ)
        sf.write('speech_8k_8bit.wav', xQ, fs)
        sf.write('speech_8k_8bit_error.wav', e, fs)

        # A-law quantization
        y = A_law_compander(x)
        yQ = uniform_midtread_quantizer(y, 8)
        xQ = A_law_expander(yQ)
        e = evaluate_requantization(x, xQ)
        sf.write('speech_Alaw_8k_8bit.wav', xQ, fs)
        sf.write('speech_Alaw_8k_8bit_error.wav', e, fs)
```

```
SNR: 35.749340 dB
SNR: 38.177564 dB
```

Original Signal Your browser does not support the audio element. ../data/speech_8k.wav

Linear Requantization to :math:‘w=8‘ bit

Signal Your browser does not support the audio element. speech_8k_8bit.wav

Error Your browser does not support the audio element. speech_8k_8bit_error.wav

A-law Requantization to :math:‘w=8‘ bit

Signal Your browser does not support the audio element. speech_Alaw_8k_8bit.wav

Error Your browser does not support the audio element. speech_Alaw_8k_8bit_error.wav

CHAPTER 6

Realization of Non-Recursive Filters

6.1 Introduction

Computing the output $y[k] = \mathcal{H}\{x[k]\}$ of a linear time-invariant (LTI) system is of central importance in digital signal processing. This is often referred to as *filtering* of the input signal $x[k]$. The methods for this purpose are typically classified into

- non-recursive and
- recursive

techniques. This section focuses on the realization of non-recursive filters.

6.1.1 Non-Recursive Filters

An LTI system can be characterized completely by its impulse response $h[k]$

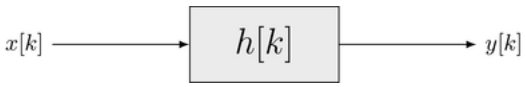


Fig. 6.1: Linear time-invariant system

The output signal $y[k]$ is given by (linear) convolution of the input signal $x[k]$ with the impulse response $h[k]$

$$y[k] = x[k] * h[k] = \sum_{\kappa=-\infty}^{\infty} x[\kappa]\ h[k-\kappa]$$

Two aspects of this representation become evident when inspecting above equation:

1. The output signal $y[k]$ is a linear combination of the input signal $x[k]$. There is no feedback of the output signal of past time-instants. Therefore, such filters are termed as *non-recursive* filters.
2. In order to compute the output signal at one particular time-instant k, the input signal needs to be known for all past and future time-instants.

The second aspect prohibits a practical realization. In order to be able to realize a non-recursive filter by convolution, the output at time-instant k should only depend on the input signal $x[k]$ up to time-index k

$$y[k] = \sum_{\kappa=-\infty}^{k} x[\kappa]\ h[k-\kappa]$$

This is the case when the impulse response is causal, hence when $h[k] = 0$ for $k < 0$. However, this still requires knowledge of the input signal for all past time-instants. If we further assume that the input signal is causal,

$x[k] = 0$ for $k < 0$, we get

$$y[k] = \sum_{\kappa=0}^{k} x[\kappa] \; h[k - \kappa]$$

6.1.2 Finite Impulse Response

Many practical systems have an impulse response of finite length N or can be approximated by an impulse response of finite length

$$h_N[k] = \begin{cases} h[k] & \text{for } 0 \leq k < N \\ 0 & \text{otherwise} \end{cases}$$

Such an impulse response is denoted as *finite impulse response* (FIR). Introducing $h_N[k]$ into above sum and rearranging terms yields

$$y[k] = \sum_{\kappa=0}^{k} x[\kappa] \; h_N[k - \kappa] = \sum_{\kappa=0}^{N-1} h_N[\kappa] \; x[k - \kappa]$$

Hence for a causal input signal $x[k]$ and a FIR the output of the system can be computed by a finite number of operations.

The evaluation of the convolution for a FIR of length N requires N multiplications and $N - 1$ additions per time index k. For the real-time convolution of an audio signal with a sampling rate of $f_S = 48$ kHz with a FIR of length $N = 48000$ we have to compute around $2 \times 2.3 \cdot 10^9$ numerical operations per second. This is a considerable numerical complexity, especially on embedded or mobile platforms. Therefore, various techniques have been developed to lower the computational complexity.

6.2 Fast Convolution

The straightforward convolution of two finite-length signals $x[k]$ and $h[k]$ is a numerical complex task. This has led to the development of various techniques with considerably lower complexity. The basic concept of the *fast convolution* is to exploit the correspondence between the convolution and the scalar multiplication in the frequency domain.

6.2.1 Convolution of Finite-Length Signals

The convolution of a causal signal $x_L[k]$ of length L with a causal impulse response $h_N[k]$ of length N is given as

$$y[k] = x_L[k] * h_N[k] = \sum_{\kappa=0}^{L-1} x_L[\kappa] \; h_N[k - \kappa] = \sum_{\kappa=0}^{N-1} h_N[\kappa] \; x_L[k - \kappa]$$

The resulting signal $y[k]$ is of finite length $M = N + L - 1$. The computation of $y[k]$ for $k = 0, 1, \ldots, M - 1$ requires $M \cdot N$ multiplications and $M \cdot (N - 1)$ additions. The computational complexity of the convolution is consequently in the order of $\mathcal{O}(M \cdot N)$. Discrete-time Fourier transformation (DTFT) of above relation yields

$$Y(e^{j\Omega}) = X_L(e^{j\Omega}) \cdot H_N(e^{j\Omega})$$

Discarding the effort of transformation, the computationally complex convolution is replaced by a scalar multiplication with respect to the frequency Ω. However, Ω is a continuous frequency variable which limits the numerical evaluation of this scalar multiplication. In practice, the DTFT is replaced by the discrete Fourier transformation (DFT). Two aspects have to be considered before a straightforward application of the DFT

1. The DFTs $X_L[\mu]$ and $H_N[\mu]$ are of length L and N respectively and cannot be multiplied straightforward
2. For $N = L$, the multiplication of the two spectra $X_L[\mu]$ and $H_L[\mu]$ would result in the periodic/circular convolution $x_L[k] \circledast h_L[k]$ due to the periodicity of the DFT. Since we aim at realizing the linear convolution $x_L[k] * h_N[k]$ with the DFT, special care has to be taken to avoid cyclic effects.

6.2.2 Linear Convolution by Periodic Convolution

The periodic convolution of the two signals $x_L[k]$ and $h_N[k]$ is defined as

$$x_L[k] \circledast h_N[k] = \sum_{\kappa=0}^{M-1} \tilde{x}_M[k-\kappa]\ h_N[\kappa]$$

where without loss of generality it is assumed that $L \geq N$ and $M \geq N$. The periodic continuation $\tilde{x}_M[k]$ of $x[k]$ with period M is given as

$$\tilde{x}_M[k] = \sum_{m=-\infty}^{\infty} x_L[m \cdot M + k]$$

The result of the circular convolution has a periodicity of M.

To compute the linear convolution by the periodic convolution one has to take care that the result of the linear convolution fits into one period of the periodic convolution. Hence, the periodicity has to be chosen as $M \geq N + L - 1$. This can be achieved by zero-padding of $x_L[k]$ and $h_N[k]$ to a total length of M

$$x_M[k] = [x_L[0], x_L[1], \ldots, x_L[L-1], 0, \ldots, 0]^{\mathrm{T}} \tag{6.1}$$

$$h_M[k] = [h[0], h[1], \ldots, h[N-1], 0, \ldots, 0]^{\mathrm{T}} \tag{6.2}$$

This results in the desired equality of linear and periodic convolution

$$x_L[k] * h_N[k] = x_M[k] \circledast h_M[k]$$

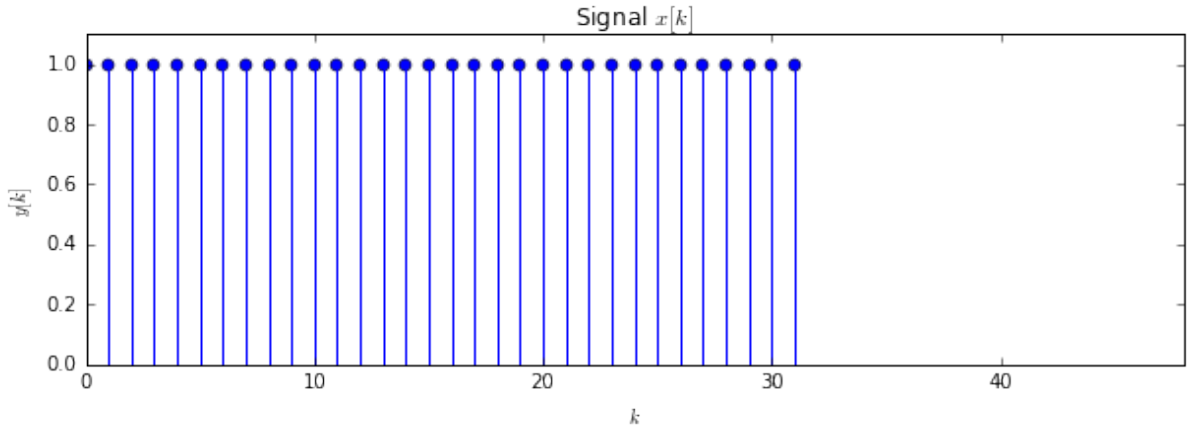
for $k = 0, 1, \ldots, M-1$ with $M = N + L - 1$.

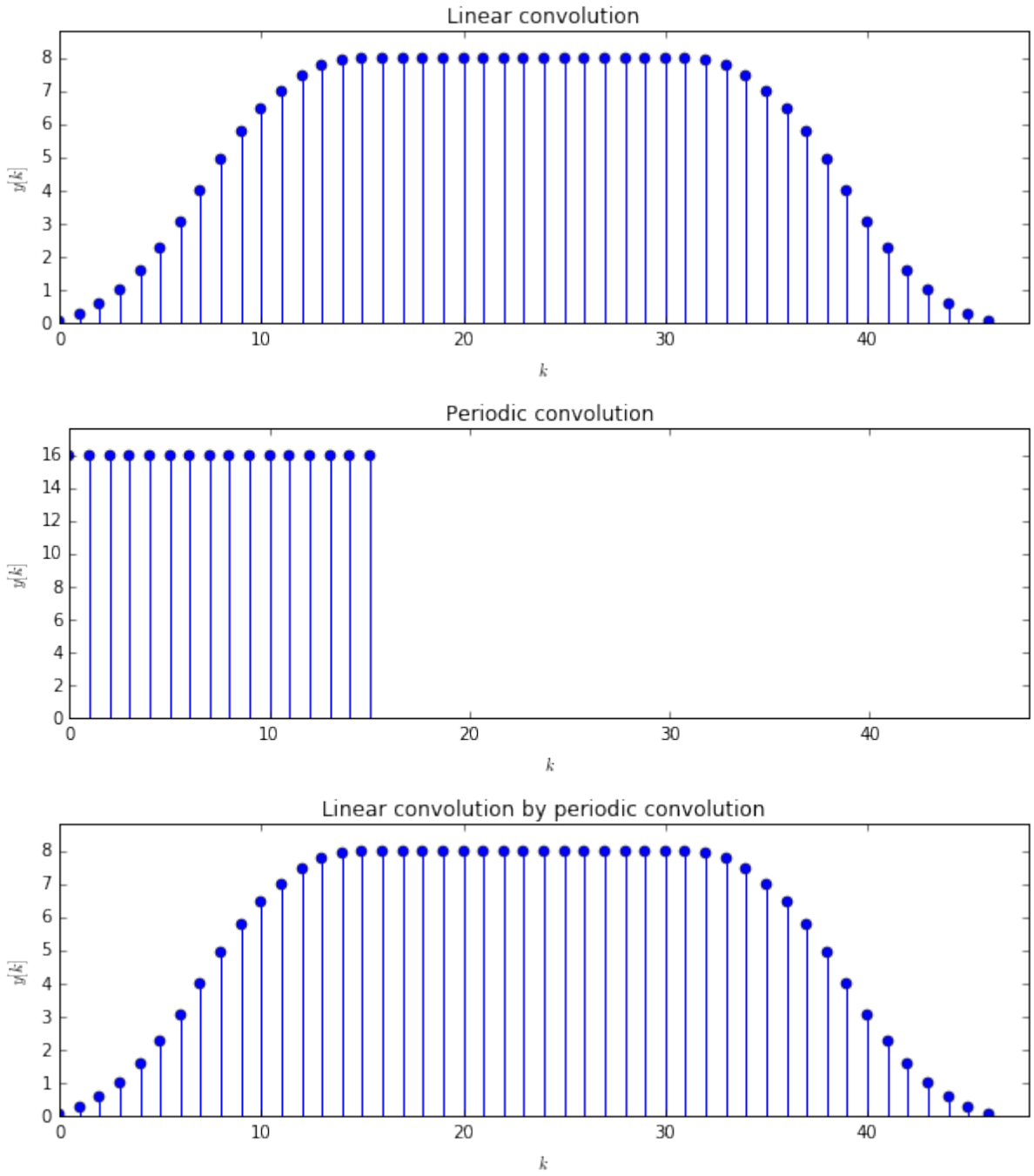
Example

The following example computes the linear, periodic and linear by periodic convolution of two signals $x[k] = \mathrm{rect}_N[k]$ and $h[k]$.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig

        L = 32  # length of signal x[k]
        N = 16  # length of signal h[k]
        M = 16  # periodicity of periodic convolution


        def cconv(x, h, M):
            L = len(x)
            N = len(h)
            # periodic continuation of x
            xc = np.copy(x)
            if M<L:
                xc[0:-M+L] += xc[M:L]
                xc = xc[0:M]
            # zero-padding of h
            hp = h
            if N<M:
                hp = np.append(hp, np.zeros(M-N))
            # circular convolution
            y = [np.dot(np.roll(xc[::-1], k+1), hp) for k in np.arange(M)]

            return np.asarray(y)
```

```
# generate signals
x = np.ones(L)
h = sig.triang(N)

# linear convolution
y1 = np.convolve(x, h, 'full')
# periodic convolution
y2 = cconv(x, h, M)
# linear convolution via periodic convolution
xp = np.append(x, np.zeros(N-1))
hp = np.append(h, np.zeros(L-1))
y3 = cconv(xp, hp, L+N-1)

# plot results
def plot_signal(x):
    plt.figure(figsize = (10, 3))
    plt.stem(x)
    plt.xlabel(r'$k$')
    plt.ylabel(r'$y[k]$')
    plt.axis([0, N+L, 0, 1.1*x.max()])


plot_signal(x)
plt.title('Signal $x[k]$')

plot_signal(y1)
plt.title('Linear convolution')

plot_signal(y2)
plt.title('Periodic convolution')

plot_signal(y3)
plt.title('Linear convolution by periodic convolution');
```





Exercise

- Change the lengths `L`, `N` and `M` within the constraints given above and check how the results for the different convolutions change

6.2.3 The Fast Convolution

Using the above derived equality of the linear and periodic convolution one can express the linear convolution $y[k] = x_L[k] * h_N[k]$ by the DFT as

$$y[k] = \text{IDFT}_M\{ \text{ DFT}_M\{x_M[k]\} \cdot \text{DFT}_M\{h_M[k]\} \ \}$$

This operation requires three DFTs of length M and M complex multiplications. On first sight this does not seem to be an improvement, since one DFT/IDFT requires M^2 complex multiplications and $M \cdot (M-1)$ complex additions. The overall numerical complexity is hence in the order of $\mathcal{O}(M^2)$. The DFT can be realized efficiently

by the fast Fourier transformation (FFT), which lowers the computational complexity to $\mathcal{O}(M \log_2 M)$. The resulting algorithm is known as *fast convolution* due to its computational efficiency.

The fast convolution algorithm is composed of the following steps

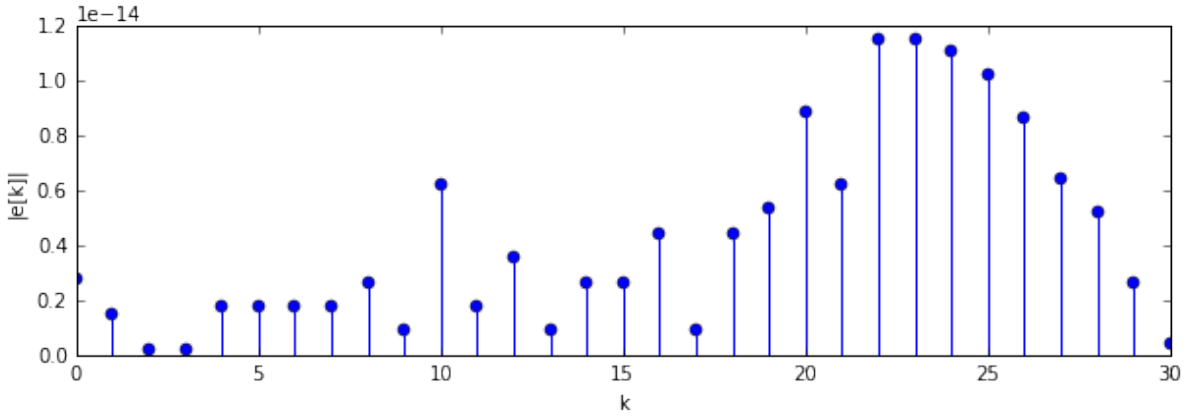
1. Zero-padding of the two input signals $x_L[k]$ and $h_N[k]$ to at least a total length of $M \geq N + L - 1$
2. Computation of the DFTs $X[\mu]$ and $H[\mu]$ using a FFT of length M
3. Multiplication of the spectra $Y[\mu] = X[\mu] \cdot H[\mu]$
4. Inverse DFT of $Y[\mu]$ using an inverse FFT of length M

The overall complexity depends on the particular implementation of the FFT. Many FFTs are most efficient for lengths which are a power of two. It therefore can make sense, in terms of computational complexity, to choose M as a power of two instead of the shortest possible length $N + L - 1$. For real valued signals $x[k] \in \mathbb{R}$ and $h[k] \in \mathbb{R}$ the computational complexity can be reduced significantly by using a real valued FFT.

Example

The implementation of the fast convolution algorithm is straightforward. Most implementations of the FFT include the zero-padding to a given length M, e.g in `numpy` by `numpy.fft.fft(x, M)`. In the following example an implementation of the fast convolution in `Python` is shown. The output of the fast convolution is compared to a straightforward implementation by means of the absolute difference $|e[k]|$. The observed differences are due to numerical effects in the convolution and the FFT. The differences can be neglected in most applications.

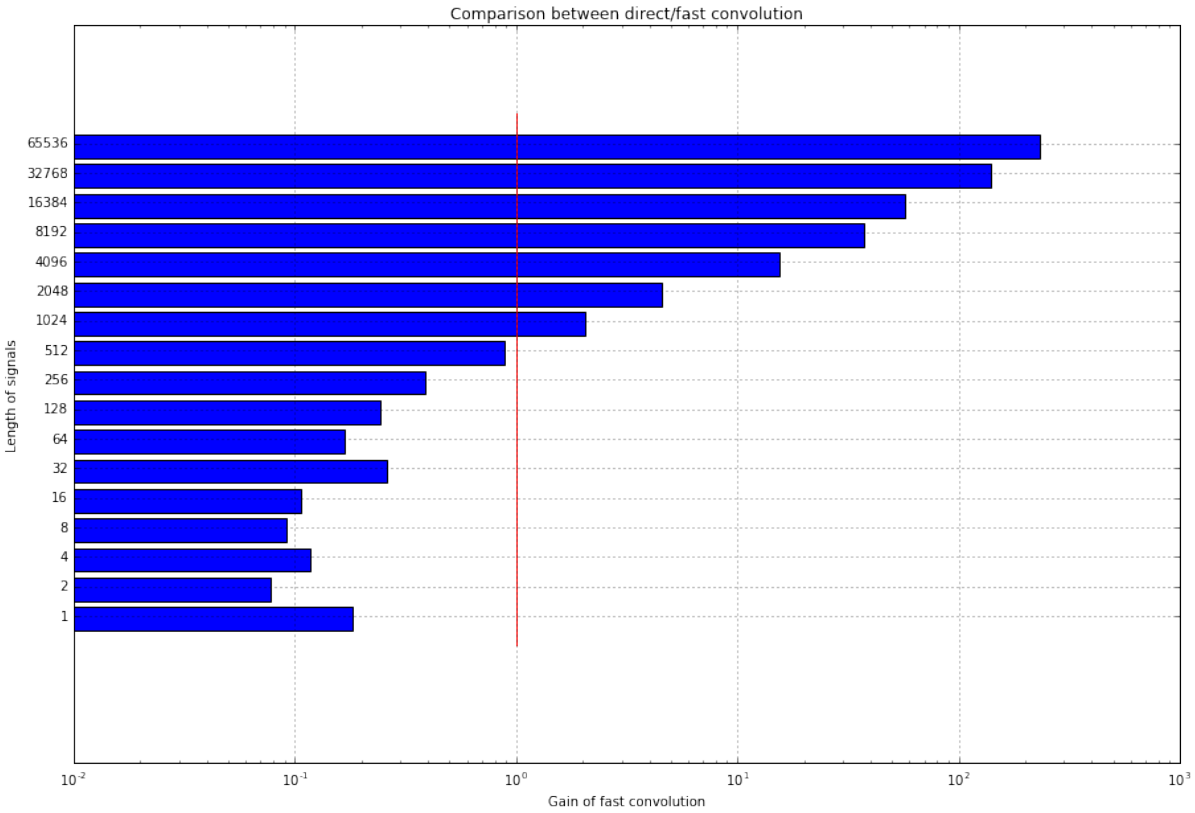
```
In [2]: L = 16  # length of signal x[k]
        N = 16  # length of signal h[k]
        M = N+L-1

        # generate signals
        x = np.ones(L)
        h = sig.triang(N)

        # linear convolution
        y1 = np.convolve(x, h, 'full')
        # fast convolution
        y2 = np.fft.ifft(np.fft.fft(x, M)*np.fft.fft(h, M))

        plt.figure(figsize=(10, 3))
        plt.stem(np.abs(y1-y2))
        plt.xlabel(r'k')
        plt.ylabel(r'|e[k]|');
```



Numerical Complexity

It was already argued that the numerical complexity of the fast convolution is considerably lower due to the usage of the FFT. The gain with respect to the convolution is evaluated in the following. In order to measure the execution times for both algorithms the `timeit` module is used. The algorithms are evaluated for the convolution of two signals $x_L[k]$ and $h_N[k]$ of length $L = N = 2^n$ for $n = 0, 1, \ldots, 16$.

```
In [3]: import timeit

        n = np.arange(17)  # lengths = 2**n to evaluate
        reps = 20  # number of repetitions for timeit

        gain = np.zeros(len(n))
        for N in n:
            length = 2**N
            # setup environment for timeit
            tsetup = 'import numpy as np; from numpy.fft import rfft, irfft; \
                    x=np.random.randn(%d); h=np.random.randn(%d)' % (length, length)
            # direct convolution
            tc = timeit.timeit('np.convolve(x, x, "full")', setup=tsetup, number=reps)
            # fast convolution
            tf = timeit.timeit('irfft(rfft(x, %d) * rfft(h, %d))' % (2*length, 2*length)
            # speedup by using the fast convolution
            gain[N] = tc/tf

        # show the results
        plt.figure(figsize = (15, 10))
        plt.barh(n-.5, gain, log=True)
        plt.plot([1, 1], [-1, n[-1]+1], 'r-')
        plt.yticks(n, 2**n)
        plt.xlabel('Gain of fast convolution')
        plt.ylabel('Length of signals')
        plt.title('Comparison between direct/fast convolution')
        plt.grid()
```



Exercise

- When is the fast convolution more efficient/faster than a direct convolution?
- Why is it slower below a given signal length?
- Is the trend of the gain as expected by the numerical complexity of the FFT?

6.3 Segmented Convolution

In many applications one of the signals of a convolution is much longer than the other. For instance when filtering a speech signal $x_L[k]$ of length L with a room impulse response $h_N[k]$ of length $N \ll L$. In such cases the fast convolution, as introduced before, does not bring a benefit since both signals have to be zero-padded to a total length of at least $N + L - 1$. Applying the fast convolution may then even be impossible in terms of memory requirements or overall delay. The filtering of a signal which is captured in real-time is also not possible by the fast convolution.

In order to overcome these limitations, various techniques have been developed that perform the filtering on limited portions of the signals. These portions are known as partitions, segments or blocks. The respective algorithms are termed as *segmented* or *block-based* algorithms. The following section introduces two techniques for the segmented convolution of signals. The basic concept of these is to divide the convolution $y[k] = x_L[k] * h_N[k]$ into multiple convolutions operating on (overlapping) segments of the signal $x_L[k]$.

6.3.1 Overlap-Add Algorithm

The overlap-add algorithm is based on splitting the signal $x_L[k]$ into non-overlapping segments $x_p[k]$ of length P

$$x_L[k] = \sum_{p=0}^{L/P-1} x_p[k - p \cdot P]$$

where the segments $x_p[k]$ are defined as

$$x_p[k] = \begin{cases} x_L[k + p \cdot P] & \text{for } k = 0, 1, \ldots, P-1 \\ 0 & \text{otherwise} \end{cases}$$

Note that $x_L[k]$ might have to be zero-padded so that its total length is a multiple of the segment length P. Introducing the segmentation of $x_L[k]$ into the convolution yields

$$\begin{aligned} y[k] &= x_L[k] * h_N[k] && (6.3) \\ &= \sum_{p=0}^{L/P-1} x_p[k - p \cdot P] * h[k] && (6.4) \\ &= \sum_{p=0}^{L/P-1} y_p[k - p \cdot P] && (6.5) \end{aligned}$$

where $y_p[k] = x_p[k] * h_N[k]$. This result states that the convolution of $x_L[k] * h_N[k]$ can be split into a series of convolutions $y_p[k]$ operating on the samples of one segment only. The length of $y_p[k]$ is $N + P - 1$. The result of the overall convolution is given by summing up the results from the segments shifted by multiples of the segment length P. This can be interpreted as an overlapped superposition of the results from the segments, as illustrated in the following diagram

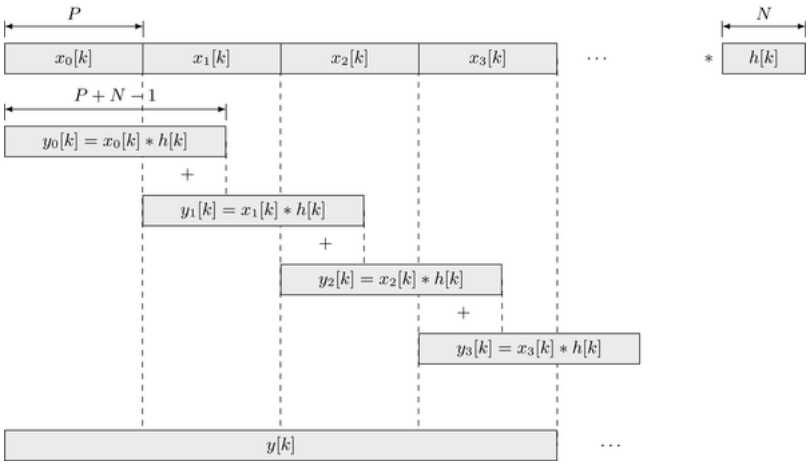


Fig. 6.2: Signal flow of overlap-add algorithm

The overall procedure is denoted by the name *overlap-add* technique. The convolutions $y_p[k] = x_p[k] * h_N[k]$ can be realized efficiently by the fast convolution using zero-padding and fast Fourier transformations (FFTs) of length $M \geq P + N - 1$.

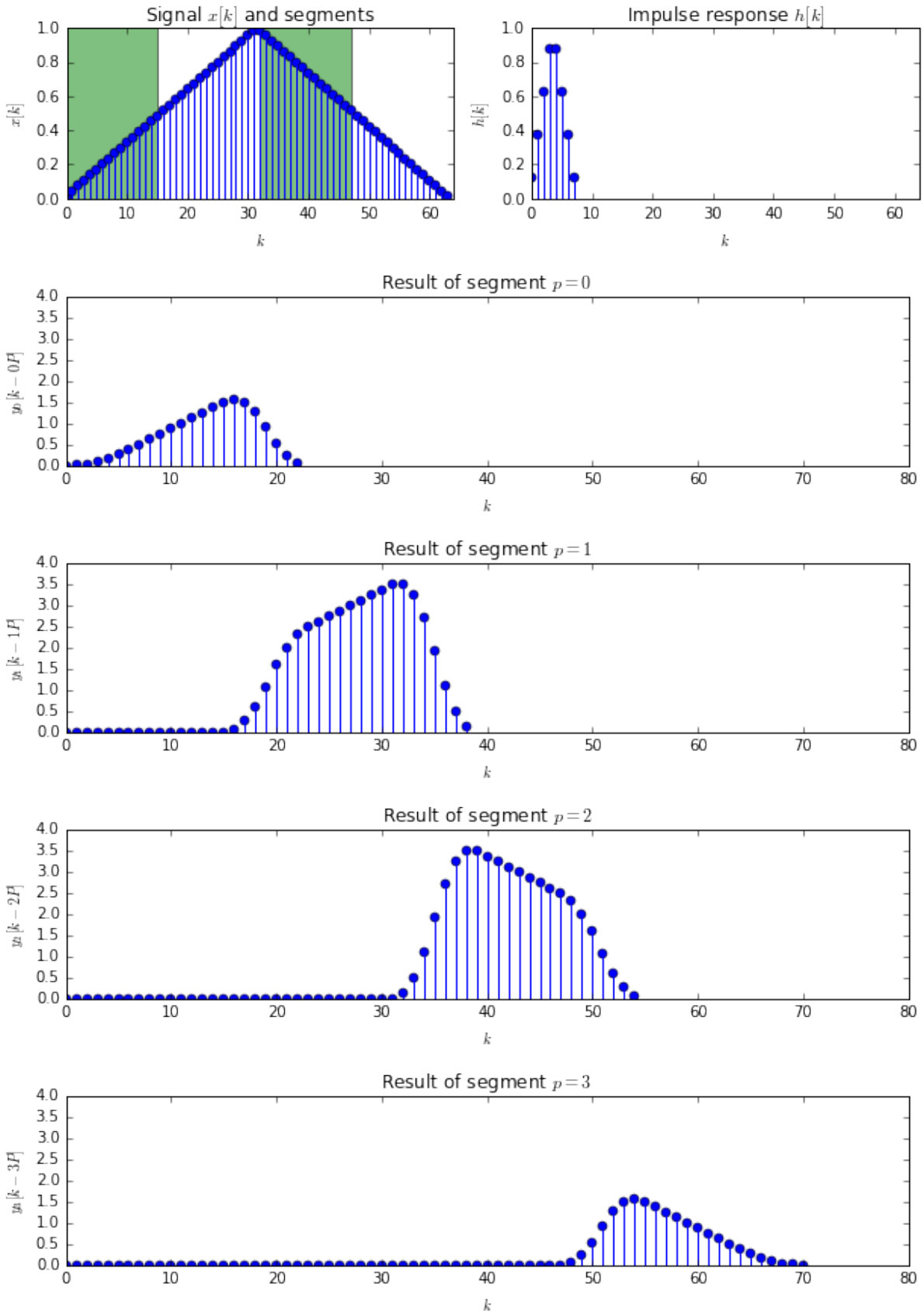
A drawback of the overlap-add technique is that the next input segment is required to compute the result for the actual segment of the output. For real-time applications this introduces an algorithmic delay of one segment.

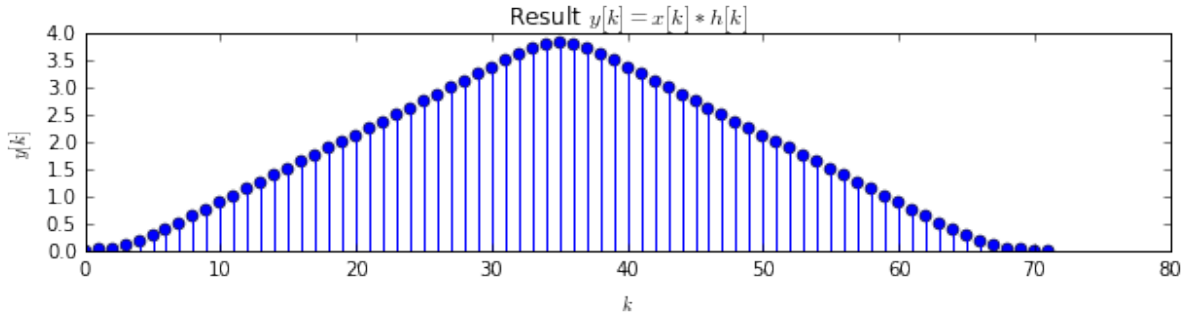
Example

The following example illustrates the overlap-add algorithm by showing the (convolved) segments and the overall result.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig


        L = 64  # length of input signal
```

```
N = 8  # length of impulse response
P = 16  # length of segments


# generate input signal
x = sig.triang(L)
# generate impulse response
h = sig.triang(N)

# overlap-add convolution
xp = np.zeros((L//P, P))
yp = np.zeros((L//P, N+P-1))
y = np.zeros(L+P-1)
for n in range(L//P):
    xp[n, :] = x[n*P:(n+1)*P]
    yp[n, :] = np.convolve(xp[n,:], h, mode='full')
    y[n*P:(n+1)*P+N-1] += yp[n, :]
y = y[0:N+L]


# plot signals
plt.figure(figsize = (10,2))

plt.subplot(121)
plt.stem(x)
for n in np.arange(L//P)[::2]:
    plt.axvspan(n*P, (n+1)*P-1, facecolor='g', alpha=0.5)
plt.title(r'Signal $x[k]$ and segments')
plt.xlabel(r'$k$')
plt.ylabel(r'$x[k]$')
plt.axis([0, L, 0, 1])

plt.subplot(122)
plt.stem(h)
plt.title(r'Impulse response $h[k]$')
plt.xlabel(r'$k$')
plt.ylabel(r'$h[k]$')
plt.axis([0, L, 0, 1])

for p in np.arange(L//P):
    plt.figure(figsize = (10,2))

    plt.stem(np.concatenate((np.zeros(p*P), yp[p, :])))
    plt.title(r'Result of segment $p=%d$' %(p))
    plt.xlabel(r'$k$')
    plt.ylabel(r'$y_%d[k - %d P]$' %(p,p))
    plt.axis([0, L+P, 0, 4])


plt.figure(figsize = (10,2))
plt.stem(y)
plt.title(r'Result $y[k] = x[k] * h[k]$')
plt.xlabel(r'$k$')
plt.ylabel(r'$y[k]$')
plt.axis([0, L+P, 0, 4]);
```


Signal $x[k]$ and segments
Impulse response $h[k]$
Result of segment $p=0$
Result of segment $p=1$
Result of segment $p=2$
Result of segment $p=3$
$x[k]$
$h[k]$
$y_0[k-0P]$
$y_1[k-1P]$
$y_2[k-2P]$
$y_3[k-3P]$
k



Exercises

- Change the length `N` of the impulse response and the length `P` of the segments. What changes?
- What influence have these two lengths on the numerical complexity of the overlap-add algorithm?

6.3.2 Overlap-Save Algorithm

The overlap-save algorithm, also known as *overlap-discard algorithm*, follows a different strategy as the overlap-add technique introduced above. It is based on an overlapping segmentation of the input $x_L[k]$ and application of the periodic convolution for the individual segments.

Lets take a closer look at the result of the periodic convolution $x_p[k] \circledast h_N[k]$, where $x_p[k]$ denotes a segment of length P of the input signal and $h_N[k]$ the impulse response of length N. The result of a linear convolution $x_p[k] * h_N[k]$ would be of length $P+N-1$. The result of the periodic convolution of period P for $P > N$ would suffer from a circular shift (time aliasing) and superposition of the last $N-1$ samples to the beginning. Hence, the first $N-1$ samples are not equal to the result of the linear convolution. However, the remaining $P-N+1$ do so.

This motivates to split the input signal $x_L[k]$ into overlapping segments of length P where the p-th segment overlaps its preceding $(p-1)$-th segment by $N-1$ samples

$$x_p[k] = \begin{cases} x_L[k + p \cdot (P-N+1) - (N-1)] & \text{for } k = 0, 1, \ldots, P-1 \\ 0 & \text{otherwise} \end{cases}$$

The part of the circular convolution $x_p[k] \circledast h_N[k]$ of one segment $x_p[k]$ with the impulse response $h_N[k]$ that is equal to the linear convolution of both is given as

$$y_p[k] = \begin{cases} x_p[k] \circledast h_N[k] & \text{for } k = N-1, N, \ldots, P-1 \\ 0 & \text{otherwise} \end{cases}$$

The output $y[k]$ is simply the concatenation of the $y_p[k]$

$$y[k] = \sum_{p=0}^{L/P-1} y_p[k - p \cdot (P-N+1) + (N-1)]$$

The overlap-save algorithm is illustrated in the following diagram

For the first segment $x_0[k]$, $N-1$ zeros have to be appended to the beginning of the input signal $x_L[k]$ for the overlapped segmentation. From the result of the periodic convolution $x_p[k] \circledast h_N[k]$ the first $N-1$ samples are discarded, the remaining $P-N+1$ are copied to the output $y[k]$. This is indicated by the alternative notation *overlap-discard* used for the technique. The periodic convolution can be realized efficiently by a FFT/IFFT of length P.

Example

The following example illustrates the overlap-save algorithm by showing the results of the periodic convolutions of the segments. The discarded parts are indicated by the red background.

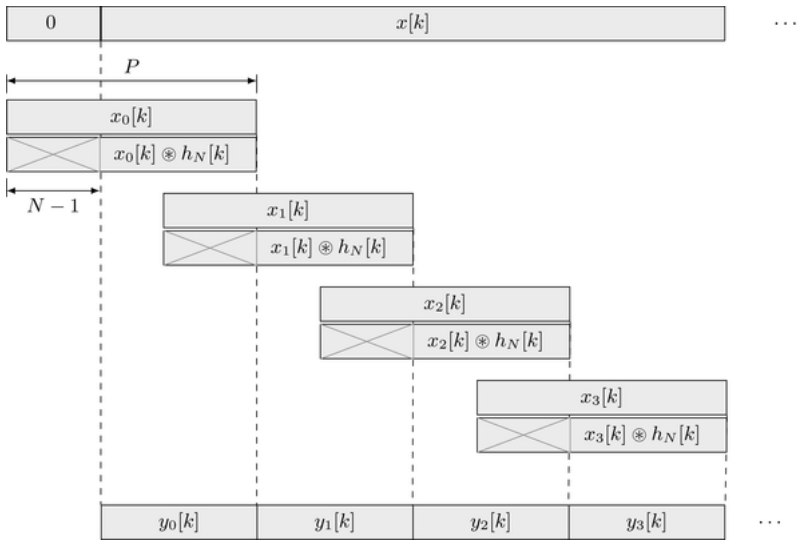
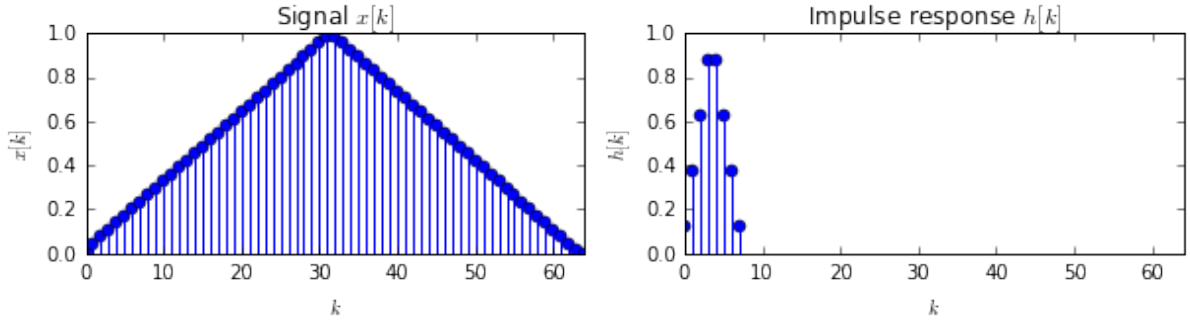


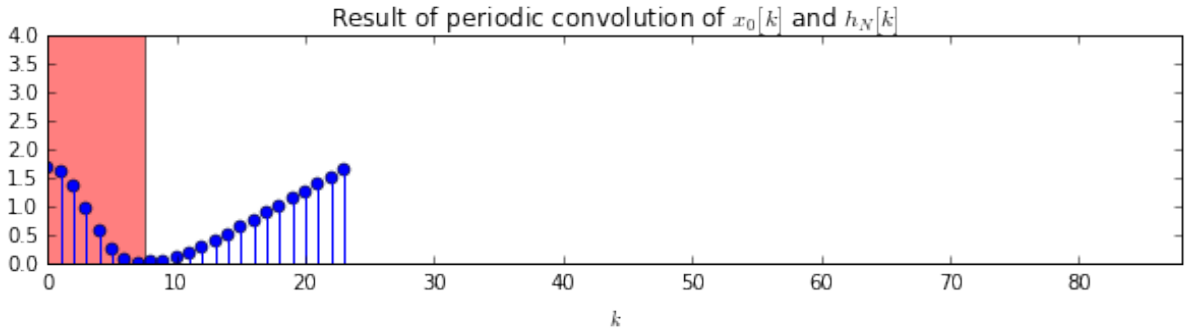
Fig. 6.3: Signal flow of overlap-save algorithm

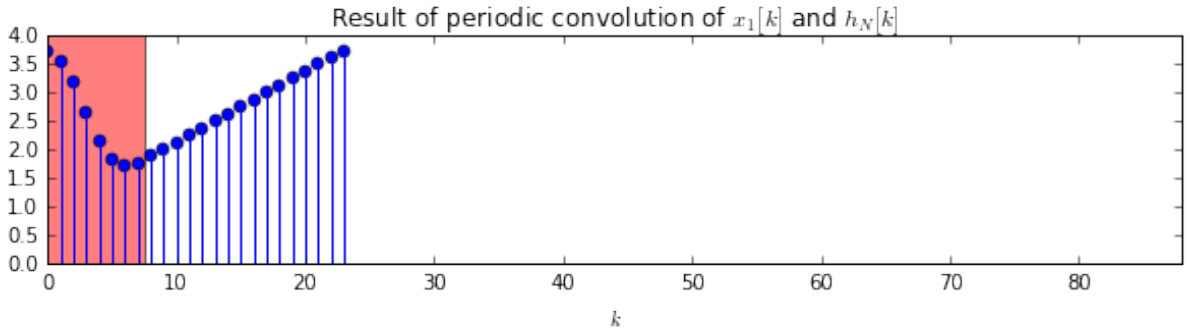
```
In [2]: L = 64  # length of input signal
        N = 8  # length of impulse response
        P = 24  # length of segments


        # generate input signal
        x = sig.triang(L)
        # generate impulse response
        h = sig.triang(N)

        # overlap-save convolution
        nseg = (L+N-1)//(P-N+1) + 1
        x = np.concatenate((np.zeros(N-1), x, np.zeros(P)))
        xp = np.zeros((nseg, P))
        yp = np.zeros((nseg, P))
        y = np.zeros(nseg*(P-N+1))

        for p in range(nseg):
            xp[p, :] = x[p*(P-N+1):p*(P-N+1)+P]
            yp[p, :] = np.fft.irfft(np.fft.rfft(xp[p, :]) * np.fft.rfft(h, P))
            y[p*(P-N+1):p*(P-N+1)+P-N+1] = yp[p, N-1:]
        y = y[0:N+L]

        plt.figure(figsize = (10,2))

        plt.subplot(121)
        plt.stem(x[N-1:])
        plt.title(r'Signal $x[k]$')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$x[k]$')
        plt.axis([0, L, 0, 1])

        plt.subplot(122)
        plt.stem(h)
        plt.title(r'Impulse response $h[k]$')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$h[k]$')
        plt.axis([0, L, 0, 1])
```

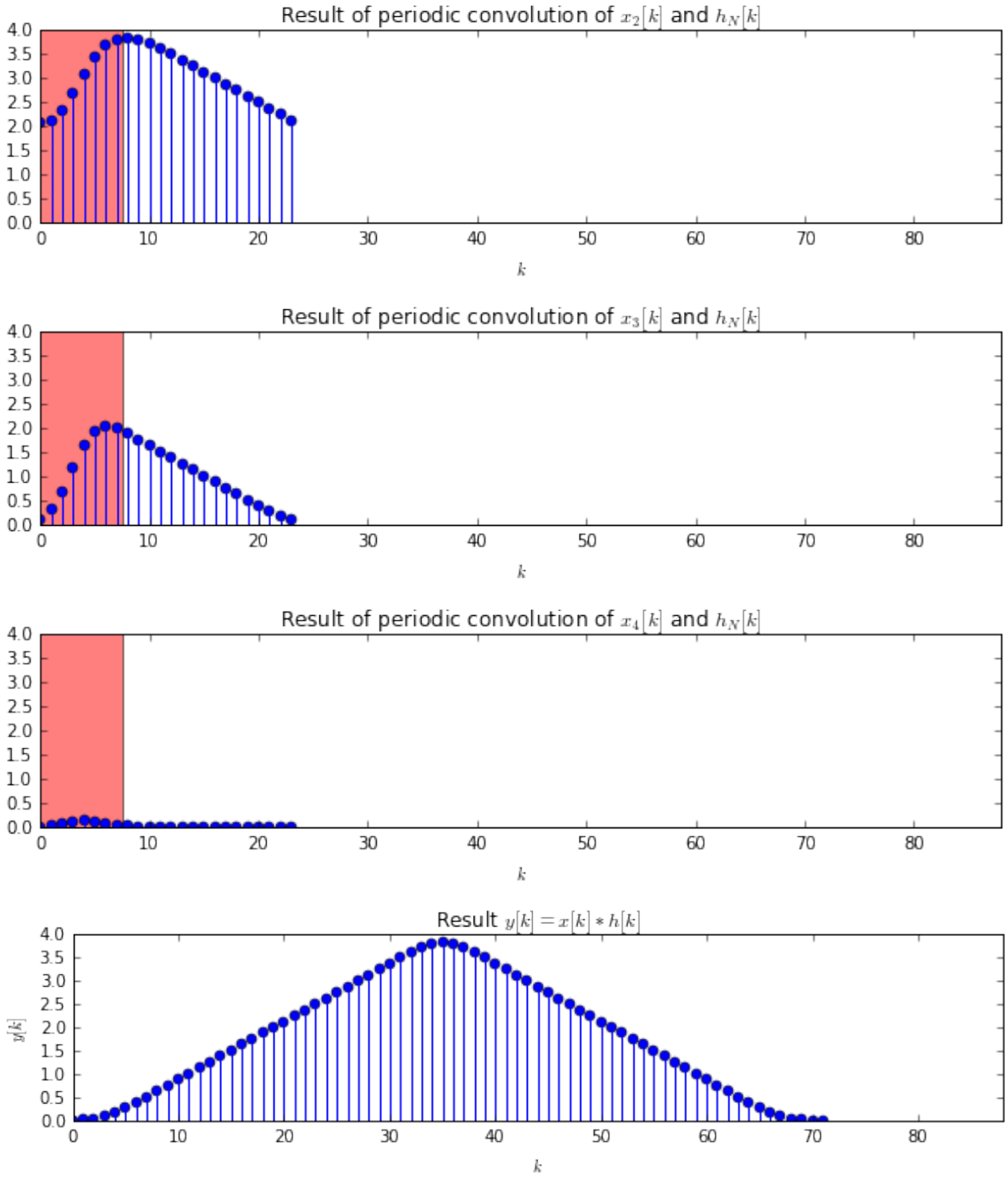
```
for p in np.arange(nseg):
    plt.figure(figsize = (10,2))
    plt.stem(yp[p, :])
    plt.axvspan(0, N-1+.5, facecolor='r', alpha=0.5)
    plt.title(r'Result of periodic convolution of $x_%d[k]$ and $h_N[k]$' %(p))
    plt.xlabel(r'$k$')
    plt.axis([0, L+P, 0, 4])
```

```
plt.figure(figsize = (10,2))
plt.stem(y)
plt.title(r'Result $y[k] = x[k] * h[k]$')
plt.xlabel(r'$k$')
plt.ylabel(r'$y[k]$')
plt.axis([0, L+P, 0, 4]);
```









Exercise

- Change the length `N` of the impulse response and the length `P` of the segments. What changes?
- How many samples of the output signal $y[k]$ are computed per segment for a particular choice of these two values?
- What would be a good choice for the segment length `P` with respect to the length `N` of the impulse response?

6.3.3 Practical Aspects and Extensions

- For both the overlap-add and overlap-save algorithm the length P of the segments influences the lengths of the convolutions, FFTs and the number of output samples per segment. The segment length is often chosen as

  - $P = N$ for overlap-add and
  - $P = 2N$ for overlap-save.

For both algorithms this requires FFTs of length $2N$ to compute P output samples. The overlap-add algorithm requires P additional additions per segment in comparison to overlap-save.

- For real-valued signals $x_L[k]$ and impulse responses $h_N[k]$ real-valued FFTs lower the computational complexity significantly. As alternative, the $2N$ samples in the FFT can be distributed into the real and complex part of a FFT of length N [*Zölzer*].
- The impulse response can be changed in each segment in order to simulate time-variant linear systems. This is often combined with an overlapping computation of the output in order to avoid artifacts due to instationarities.
- For long impulse responses $h_N[k]$ or low-delay applications, algorithms have been developed which base on an additional segmentation of the impulse response. This is known as *partitioned convolution*.

6.4 Quantization Effects

Numbers and numerical operations are represented with a finite numerical resolution in digital processors. The same holds for the amplitude values of signals and the algorithmic operations applied to them. Hence, the intended characteristics of an digital filter may deviate in practice due to the finite numerical resolution. The double-precision floating point representation used in numerical environments like MATLAB or Python/numpy is assumed to be quasi-continuous. This representation serves therefore as reference for the evaluation of quantization effects.

This section investigates the consequences of quantization in non-recursive filters. We first take a look on the quantization of the filter coefficients, followed by the effects caused by a finite numerical resolution on the operations. The realization of non-recursive filters is subject to both effects.

6.4.1 Quantization of Filter Coefficients

The output signal $y[k]$ of a non-recursive filter with a finite impulse response (FIR) $h[k]$ of length N is given as

$$y[k] = h[k] * x[k] = \sum_{\kappa=0}^{N-1} h[\kappa]\; x[k-\kappa]$$

where $x[k]$ denotes the input signal. The quantized impulse response $h_Q[k]$ (quantized filter coefficients) is yielded by quantizing the impulse response $h[k]$

$$h_Q[k] = \mathcal{Q}\{h[k]\} = h[k] + e[k]$$

where $e[k] = h_Q[k] - h[k]$ denotes the *quantization error*. Introducing $h_Q[k]$ into above equation and rearranging results in

$$y_Q[k] = \sum_{\kappa=0}^{N-1} h[k]\; x[k-\kappa] + \sum_{\kappa=0}^{N-1} e[k]\; x[k-\kappa]$$

The input signal $x[k]$ is filtered by the quantization noise $e[k]$ and superimposed to the desired output of the filter. The overall transfer function $H_Q(e^{j\Omega})$ of the filter with quantized filter coefficients is given as

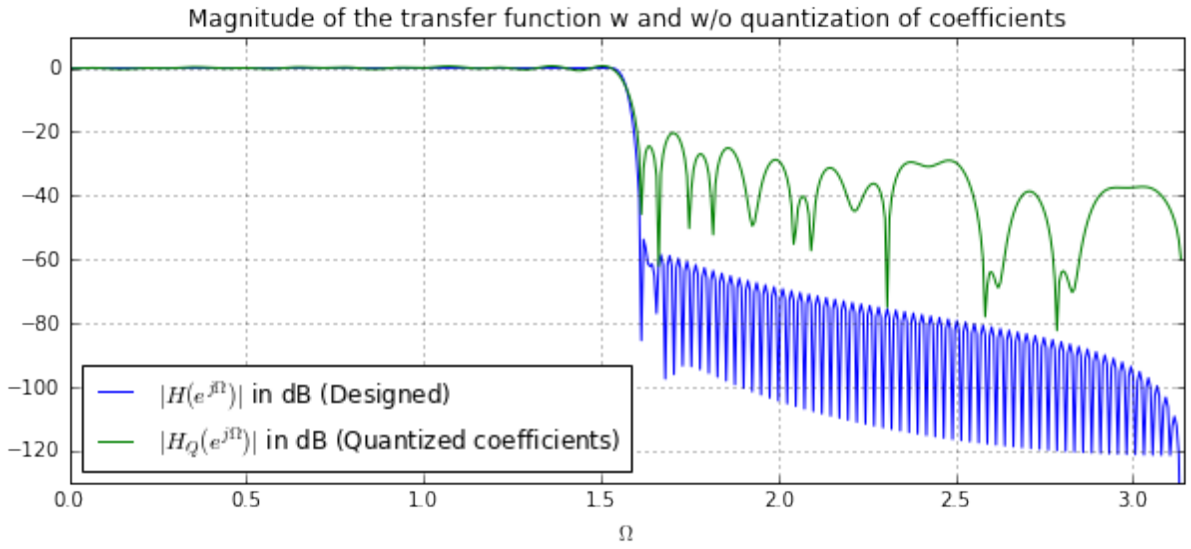
$$H_Q(e^{j\Omega}) = \sum_{k=0}^{N-1} h[k]\; e^{-j\Omega k} + \sum_{k=0}^{N-1} e[k]\; e^{-j\Omega k} = H(e^{j\Omega}) + E(e^{j\Omega})$$

Hence, the quantization of filter coefficients results in a linear distortion of the desired frequency response. To some extend this distortion can be incorporated into the design of the filter. However, the magnitude of the quantization error $|E(e^{j\Omega})|$ cannot get arbitrarily small for a finite quantization step Q. This limits the achievable attenuation of a digital filter with quantized coefficients. It is therefore important to normalize the filter coefficients $h[k]$ before quantization in order to keep the relative power of the quantization noise small.

Example

The coefficients of a digital lowpass filter with a cutoff frequency of $\Omega_0 = \frac{\pi}{2}$ are quantized in the following example.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig

        w = 8  # wordlength of quantized coefficients
        A = 1  # attenuation of filter coefficients
        N = 256  # number of coefficients for filter
        Q = 1/(2**(w-1))  # quantization stepsize

        def uniform_midtread_quantizer(x, Q):
            # limiter
            x = np.copy(x)
            idx = np.where(x <= -1)
            x[idx] = -1
            idx = np.where(x > 1 - Q)
            x[idx] = 1 - Q
            # linear uniform quantization
            xQ = Q * np.floor(x/Q + 1/2)

            return xQ

        # design lowpass
        h = A * sig.firwin(N, .5)
        # quantize coefficients
        hQ = uniform_midtread_quantizer(h, Q)

        # plot frequency response
        Om, H = sig.freqz(h)
        Om, HQ = sig.freqz(hQ)
        Om, E = sig.freqz(hQ-h)

        plt.figure(figsize=(10, 4))
        plt.plot(Om, 20*np.log10(np.abs(H)), label=r'$| H(e^{j \Omega}) |$ in dB (Design
        plt.plot(Om, 20*np.log10(np.abs(HQ)), label=r'$| H_Q(e^{j \Omega}) |$ in dB (Qua
        plt.title('Magnitude of the transfer function w and w/o quantization of coeffici
        plt.xlabel(r'$\Omega$')
        plt.axis([0, np.pi, -130, 10])
        plt.legend(loc=3)
        plt.grid()
```



Exercise

- Change the wordlength `w` of the quantized filter coefficients. How does the magnitude response $|H_Q(e^{j\Omega})|$ of the quantized filter change?
- Change the attenuation `A` of the filter coefficients. What changes?
- Why does the magnitude response of the quantized filter $|H_Q(e^{j\Omega})|$ deviate more from the magnitude response of the designed filter $|H(e^{j\Omega})|$ in the frequency ranges with high attenuation?

6.4.2 Quantization of Signals and Operations

Besides the quantization of filter coefficients $h[k]$, also the quantization of the signals, state variables and operations has to be considered in a practical implementation of filters. The computation of the output signal $y[k] = h[k] * x[k]$ of a non-recursive filter by the convolution involves multiplications and additions. In digital signal processors numbers are often represented in fixed-point arithmetic using [two's complement](https://en.wikipedia.org/wiki/Two's_complement). When multiplying two numbers with a wordlength of w-bits in this representation the result would require $2w$-bits. Hence the result has to be requantized to w-bits. The rounding operation in the quantizer is often realized as truncation of the w least significant bits. The resulting quantization error is known as round-off error. The addition of two numbers may fall outside the maximum/minimum values of the representation and may suffer from clipping. Similar considerations hold also for other number representations, like e.g. floating point.

As for the *quantization noise*, a statistical model for the round-off error in multipliers is used to quantify the average impact of round-off noise in a non-recursive filter.

Model for round-off errors in multipliers

As outlined above, multipliers require a requantization of the result in order to keep the wordlength constant. The multiplication of a quantized signal $x_Q[k]$ with a quantized factor a_Q can be written as

$$y_Q[k] = \mathcal{Q}\{a_Q \cdot x_Q[k]\} = a_Q \cdot x_Q[k] + e[k]$$

where the round-off error $e[k]$ is defined as

$$e[k] = y_Q[k] - a_Q \cdot x_Q[k]$$

This leads to the following model of a multiplier including round-off effects

The round-off error can be modeled statistically in the same way as quantization noise [*Zölzer*]. Under the assumption that the average magnitude of $a_Q \cdot x_Q[k]$ is much larger that the quantization step size Q, the round-off error $e[k]$ can be approximated by the following statistical model

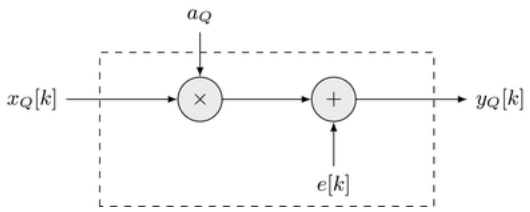


Fig. 6.4: Model for round-off noise in a multiplier

1. The round-off error $e[k]$ is not correlated with the input signal $x_Q[k]$
2. The round-off error is white

$$\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sigma_e^2$$

3. The probability density function (PDF) of the round-off error is given by the zero-mean *uniform distribution*

$$p_e(\theta) = \frac{1}{Q} \cdot \mathrm{rect}\left(\frac{\theta}{Q}\right)$$

The variance (power) of the round-off error is *derived from its PDF* as

$$\sigma_e^2 = \frac{Q^2}{12}$$

Round-off noise

Using above model of a multiplier and discarding clipping, a straightforward realization of the convolution with quantized signals would be to requantize after every multiplication

$$y_Q[k] = \sum_{\kappa=0}^{N-1} \mathcal{Q}\{h_Q[\kappa]\; x_Q[k-\kappa]\} = \sum_{\kappa=0}^{N-1} h_Q[\kappa]\; x_Q[k-\kappa] + e[\kappa]$$

The round-off errors for each multiplication are uncorrelated to each other. The overall power of the round-off error is then given as

$$\sigma_e^2 = N \cdot \frac{Q^2}{12}$$

Many digital signal processors allow to perform the multiplications and additions in an internal register with double wordlength. In this case only the result has to be requantized

$$y_Q[k] = \mathcal{Q}\left\{\sum_{\kappa=0}^{N-1} h_Q[\kappa]\; x_Q[k-\kappa]\right\}$$

and the power of the round-off noise in this case is

$$\sigma_e^2 = \frac{Q^2}{12}$$

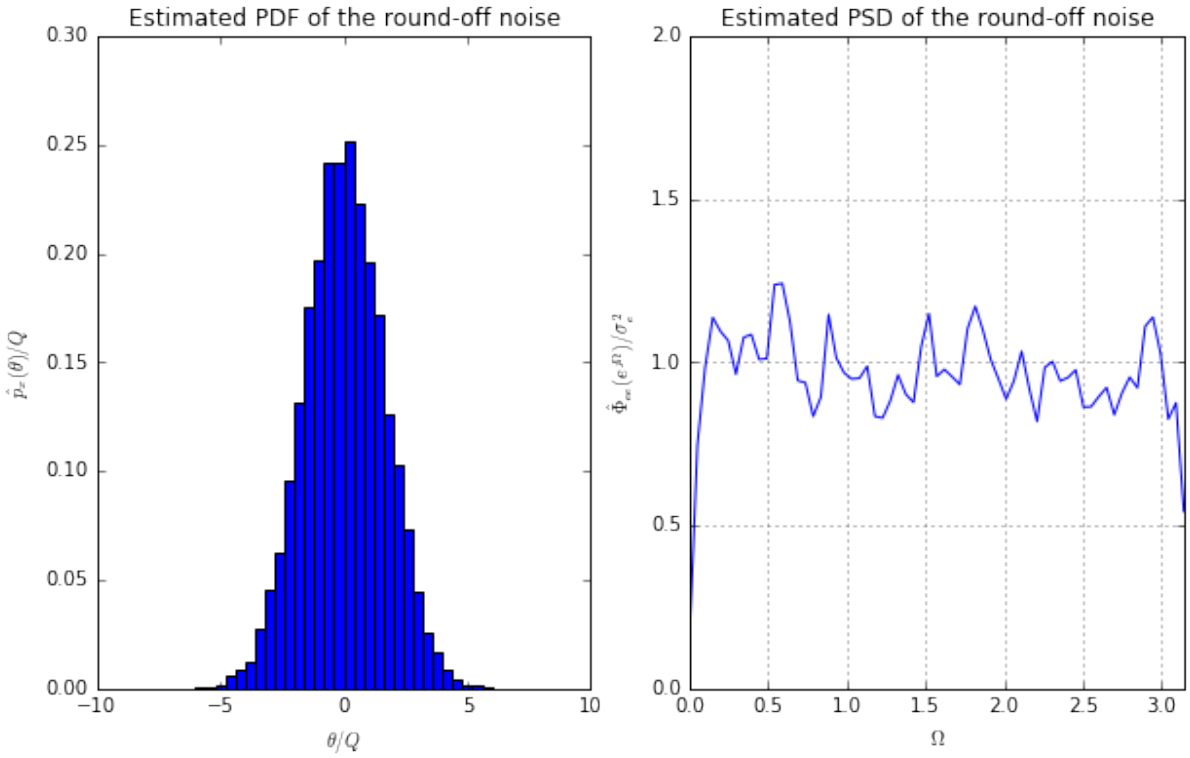
It is evident that this realization is favorable due to the lower round-off noise, especially for filters with a large number N of coefficients.

Example

The following example simulates the round-off noise of a non-recursive filter when requantization is performed after each multiplication. Clipping is not considered. The input signal $x[k]$ is drawn from a *uniform distribution* with $a = -1$ and $b = 1 - Q$. Both the input signal and filter coefficients are quantized. The output signal $y[k]$ without requantization of the multiplications is computed, as well as the output signal $y_Q[k]$ with requantization. The statistical properties of the round-off noise $e[k] = y_Q[k] - y[k]$ are evaluated.

In [2]:
```
w = 16  # wordlength of quantized coefficients/operations
N = 32  # number of coefficients for filter
L = 8192  # length of input signal
Q = 1/(2**(w-1))  # quantization stepsize


def uniform_midtread_quantizer(x, Q):
    xQ = Q * np.floor(x/Q + 1/2)

    return xQ


# random impulse response
h = np.random.uniform(size=N, low=-1, high=1)
hQ = uniform_midtread_quantizer(h, Q)
# input signal
x = np.random.uniform(size=L, low=-1, high=1-Q)
xQ = uniform_midtread_quantizer(x, Q)
# output signal by convolution
y = np.zeros(L+N-1)
yQ = np.zeros(L+N-1)
for k in np.arange(L):
    for kappa in np.arange(N):
        if (k-kappa) >= 0:
            y[k] += hQ[kappa] * xQ[k-kappa]
            yQ[k] += uniform_midtread_quantizer(hQ[kappa] * xQ[k-kappa], Q)

# overall round-off error
e = yQ - y

# estimate power of round-off error
sx = 10*np.log10(np.var(e))
print('Power of overall round-off noise is %f dB' %sx)
# estimated PDF of round-off error
pe, bins = np.histogram(e, bins=50, normed=True, range=(-10*Q, 10*Q))
# estimate PSD of round-off error
nf, Pee = sig.welch(e, nperseg=128)


# plot statistical properties of error signal
plt.figure(figsize=(10,6))

plt.subplot(121)
plt.bar(bins[:-1]/Q, pe*Q, width = 20/len(pe))
plt.title('Estimated PDF of the round-off noise')
plt.xlabel(r'$\theta / Q$')
plt.ylabel(r'$\hat{p}_x(\theta) / Q$')
#plt.axis([-1, 1, 0, 1.2])

plt.subplot(122)
plt.plot(nf*2*np.pi, Pee*6/Q**2/N)
plt.title('Estimated PSD of the round-off noise')
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$\hat{\Phi}_{ee}(e^{j \Omega}) / \sigma_e^2$')
plt.axis([0, np.pi, 0, 2])
plt.grid();
```

```
Power of overall round-off noise is -86.141757 dB
```



Exercise

- Change the wordlength `w` and check if the σ_e^2 derived by numerical simulation is equal to its theoretic value derived above?
- Can you explain the shape of the estimated PDF for the round-off noise?

CHAPTER 7

Realization of Recursive Filters

7.1 Introduction

Computing the output $y[k] = \mathcal{H}\{x[k]\}$ of a linear time-invariant (LTI) system is of central importance in digital signal processing. This is often referred to as *filtering* of the input signal $x[k]$. We already have discussed the realization of non-recursive filters. This section focuses on the realization of recursive filters.

7.1.1 Recursive Filters

Linear difference equations with constant coefficients represent linear-time invariant (LTI) systems

$$\sum_{n=0}^{N} a_n \; y[k-n] = \sum_{m=0}^{M} b_m \; x[k-m]$$

where $y[k] = \mathcal{H}\{x[k]\}$ denotes the response of the system to the input signal $x[k]$, N the order, a_n and b_m constant coefficients, respectively. Above equation can be rearranged with respect to the output signal $y[k]$ by extracting the first element ($n = 0$) of the left hand sum

$$y[k] = \frac{1}{a_0} \left(\sum_{m=0}^{M} b_m \; x[k-m] - \sum_{n=1}^{N} a_n \; y[k-n] \right)$$

It is evident that the output signal $y[k]$ at time instant k is given as a linear combination of past output samples $y[k-n]$ superimposed by a linear combination of the actual $x[k]$ and past $x[k-m]$ input samples. Hence, the actual output $y[k]$ is composed from the two contributions

1. a *non-recursive part*, and
2. a recursive part where a linear combination of past output samples is fed back.

The impulse response of the system is given as the response of the system to a Dirac impulse at the input $h[k] = \mathcal{H}\{\delta[k]\}$. Using above result and the properties of the discrete Dirac impulse we get

$$h[k] = \frac{1}{a_0} \left(b_k - \sum_{n=1}^{N} a_n \; h[k-n] \right)$$

Due to the feedback, the impulse response will in general be of infinite length. The impulse response is termed as infinite impulse response (IIR) and the system as recursive system/filter.

7.1.2 Transfer Function

Applying a z-transform to the left and right hand side of the difference equation and rearranging terms yields the transfer function $H(z)$ of the system

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{m=0}^{M} b_m \; z^{-m}}{\sum_{n=0}^{N} a_n \; z^{-n}}$$

The transfer function is given as a rational function in z. The polynominals of the numerator and denominator can expressed alternatively by their roots as

$$H(z) = \frac{b_M}{a_N} \cdot \frac{\prod_{\mu=1}^{P}(z - z_{0\mu})^{m_\mu}}{\prod_{\nu=1}^{Q}(z - z_{\infty\nu})^{n_\nu}}$$

where $z_{0\mu}$ and $z_{\infty\nu}$ denote the μ-th zero and ν-th pole of degree m_μ and n_ν of $H(z)$, respectively. The total number of zeros and poles is denoted by P and Q. Due to the symmetries of the z-transform, the transfer function of a real-valued system $h[k] \in \mathbb{R}$ exhibits complex conjugate symmetry

$$H(z) = H^*(z^*)$$

Poles and zeros are either real valued or conjugate complex pairs for real-valued systems ($b_m \in \mathbb{R}$, $a_n \in \mathbb{R}$). For the poles of a causal and stable system $H(z)$ the following condition has to hold

$$\max_\nu |z_{\infty\nu}| < 1$$

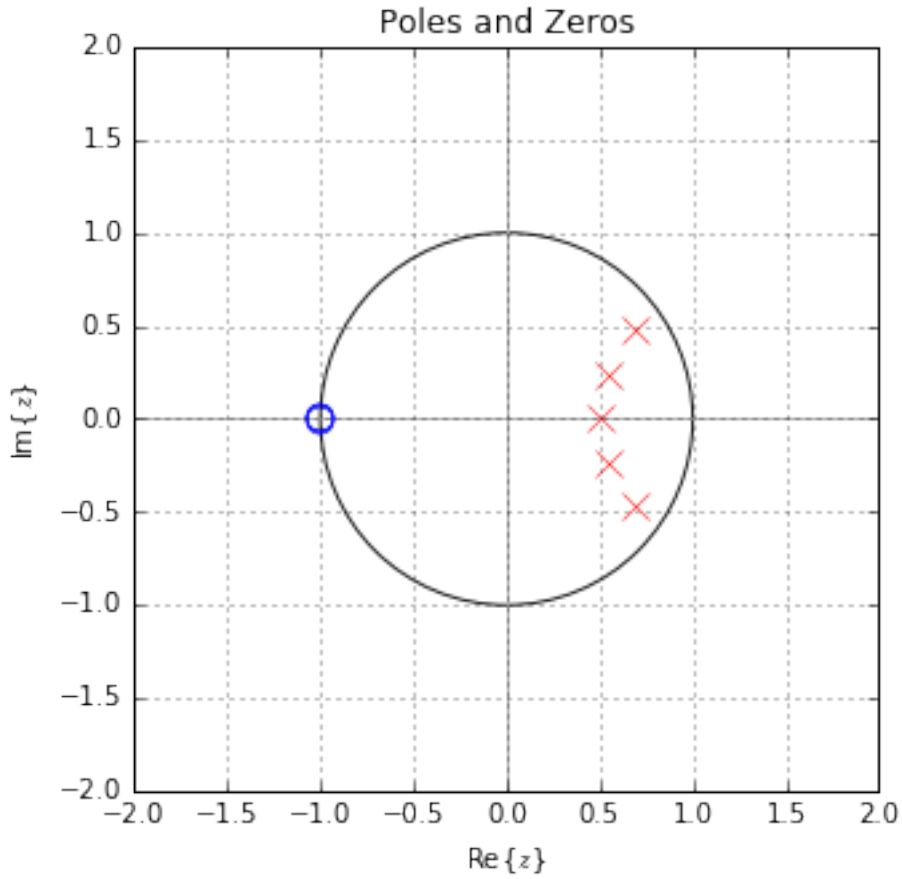
Hence all poles have to be located inside the unit circle $|z| = 1$. Amongst others, this implies that $M \leq N$.

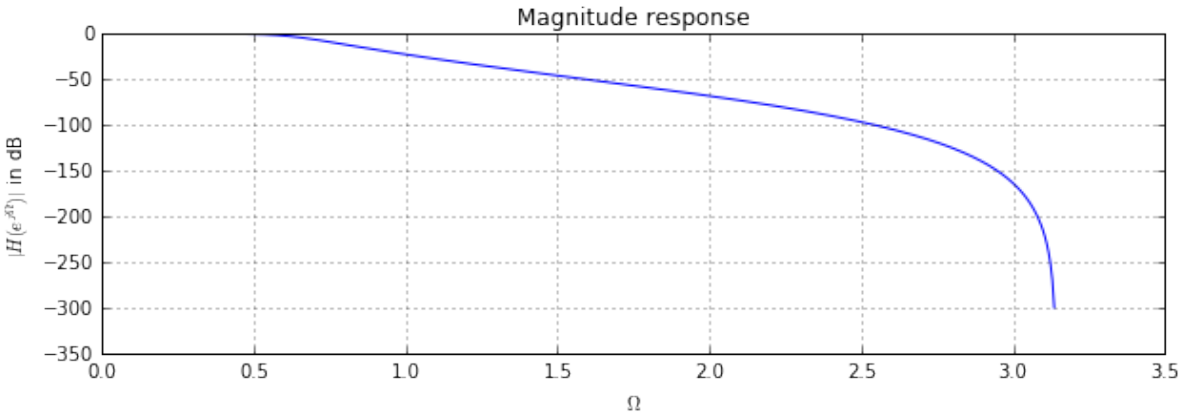
7.1.3 Example

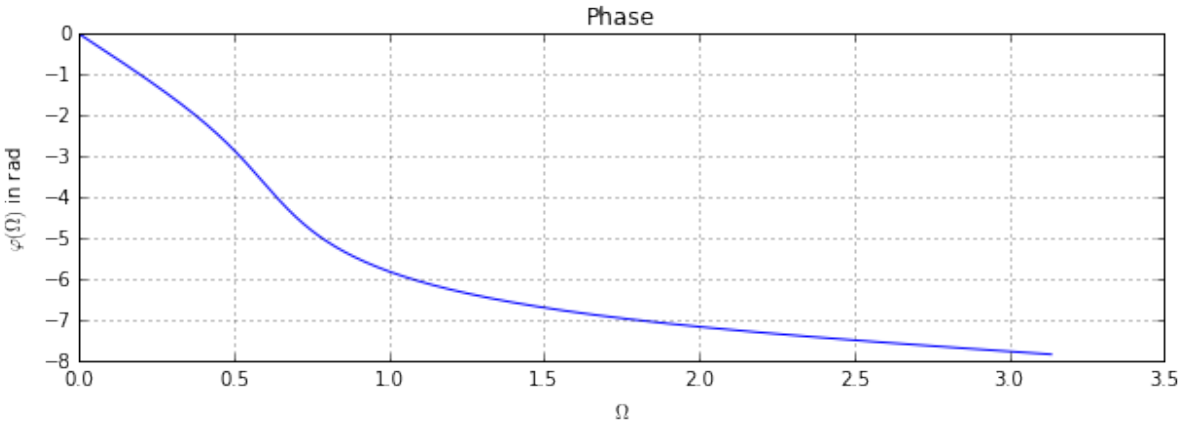
The following example shows the pole/zero diagram, the magnitude and phase response, and impulse response of a recursive filter with so called Butterworth lowpass characteristic.

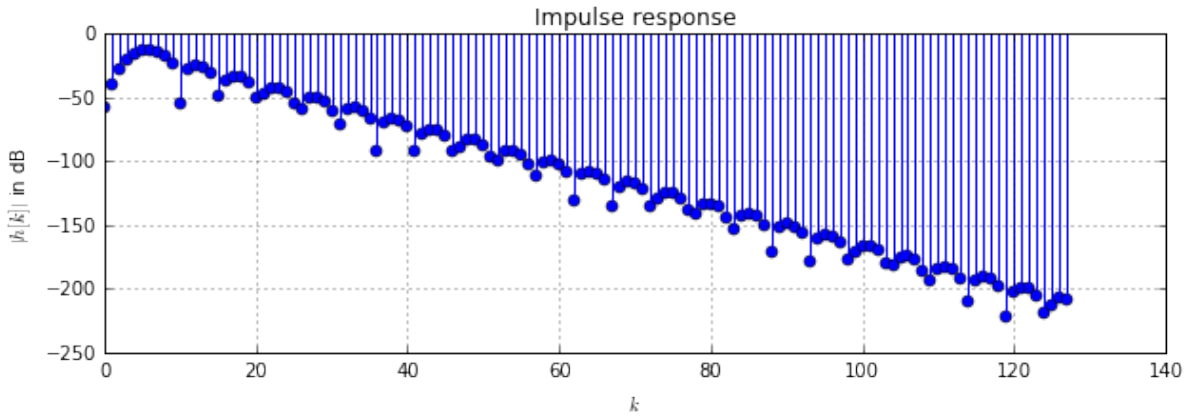
```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        from matplotlib.markers import MarkerStyle
        from matplotlib.patches import Circle
        import scipy.signal as sig

        N = 5  # order of recursive filter


        def zplane(z, p):

            fig = plt.figure(figsize=(5,5))
            ax = fig.gca()
            plt.hold(True)

            unit_circle = Circle((0,0), radius=1, fill=False,
                                 color='black', ls='solid', alpha=0.9)
            ax.add_patch(unit_circle)
            ax.axvline(0, color='0.7')
            ax.axhline(0, color='0.7')
            plt.axis('equal')
            plt.xlim((-2, 2))
            plt.ylim((-2, 2))
            plt.grid()
            plt.title('Poles and Zeros')
            plt.xlabel(r'Re{$z$}')
            plt.ylabel(r'Im{$z$}')

            ax.plot(np.real(z), np.imag(z), 'bo', fillstyle='none', ms = 10)
            ax.plot(np.real(p), np.imag(p), 'rx', fillstyle='none', ms = 10)

            plt.hold(False)
```

```
# coefficients of recursive filter
b, a = sig.butter(N, 0.2, 'low')
# compute transfer function of filter
Om, H = sig.freqz(b, a)
# compute impulse response
k = np.arange(128)
x = np.where(k==0, 1.0, 0)
h = sig.lfilter(b, a, x)

# plot pole/zero-diagram
zplane(np.roots(b), np.roots(a))
# plot magnitude response
plt.figure(figsize=(10, 3))
plt.plot(Om, 20 * np.log10(abs(H)))
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$|H(e^{j \Omega})|$ in dB')
plt.grid()
plt.title('Magnitude response')
# plot phase response
plt.figure(figsize=(10, 3))
plt.plot(Om, np.unwrap(np.angle(H)))
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$\varphi (\Omega)$ in rad')
plt.grid()
plt.title('Phase')
# plot impulse response
plt.figure(figsize=(10, 3))
plt.stem(20*np.log10(np.abs(np.squeeze(h))))
plt.xlabel(r'$k$')
plt.ylabel(r'$|h[k]|$ in dB')
plt.grid()
plt.title('Impulse response');
```


Poles and Zeros
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
Im{z}
Re{z}


Magnitude response
0
−50
−100
−150
−200
−250
−300
−350
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5
Ω
|H(e^jΩ)| in dB


Phase
0
−1
−2
−3
−4
−5
−6
−7
−8
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5
Ω
φ(Ω) in rad



Exercise

- Does the system have an IIR?
- What happens if you increase the order `N` of the filter?

7.2 Direct Form Structures

The output signal $y[k] = \mathcal{H}\{x[k]\}$ of a recursive linear-time invariant (LTI) system is given by

$$y[k] = \frac{1}{a_0} \left(\sum_{m=0}^{M} b_m \, x[k-m] - \sum_{n=1}^{N} a_n \, y[k-n] \right)$$

where a_n and b_m denote constant coefficients and N the order. Note that systems with $M > N$ are in general not stable. The computational realization of above equation requires additions, multiplications, the actual and past samples of the input signal $x[k]$, and the past samples of the output signal $y[k]$. Technically this can be realized by

- adders
- multipliers, and
- unit delays or storage elements.

These can be arranged in different topologies. A certain class of structures, which is introduced in the following, is known as *direct form structures*. Other known forms are for instance cascaded sections, parallel sections, lattice structures and state-space structures.

For the following it is assumed that $a_0 = 1$. This can be achieved for instance by dividing the remaining coefficients by a_0.

7.2.1 Direct Form I

The direct form I is derived by rearranging above equation for $a_0 = 1$

$$y[k] = \sum_{m=0}^{M} b_m \, x[k-m] + \sum_{n=1}^{N} -a_n \, y[k-n]$$

It is now evident that we can realize the recursive filter by a superposition of a non-recursive and a recursive part. With the elements given above, this results in the following block-diagram

This representation is not canonical since $N + M$ unit delays are required to realize a system of order N. A benefit of the direct form I is that there is essentially only one summation point which has to be taken care of when considering quantized variables and overflow. The output signal $y[k]$ for the direct form I is computed by realizing above equation.

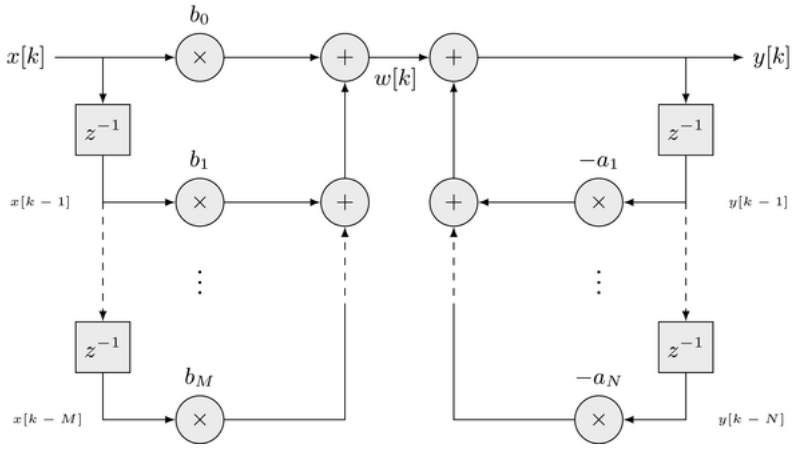


Fig. 7.1: Direct form I filter

The block diagram of the direct form I can be interpreted as the cascade of two systems. Denoting the signal in between both as $w[k]$ and discarding initial values we get

$$w[k] = \sum_{m=0}^{M} b_m \, x[k-m] = h_1[k] * x[k] \tag{7.1}$$

$$y[k] = w[k] + \sum_{n=1}^{N} -a_n \, w[k-n] = h_2[k] * w[k] = h_2[k] * h_1[k] * x[k] \tag{7.2}$$

where $h_1[k] = [b_0, b_1, \ldots, b_M]$ denotes the impulse response of the non-recursive part and $h_2[k] = [1, -a_1, \ldots, -a_N]$ for the recursive part. From the last equality of the second equation and the commutativity of the convolution it becomes clear that the order of the cascade can be exchanged.

7.2.2 Direct Form II

The direct form II is yielded by exchanging the two systems in above block diagram and noticing that there are two parallel columns of delays which can be combined, since they are redundant. For $N = M$ it is given as

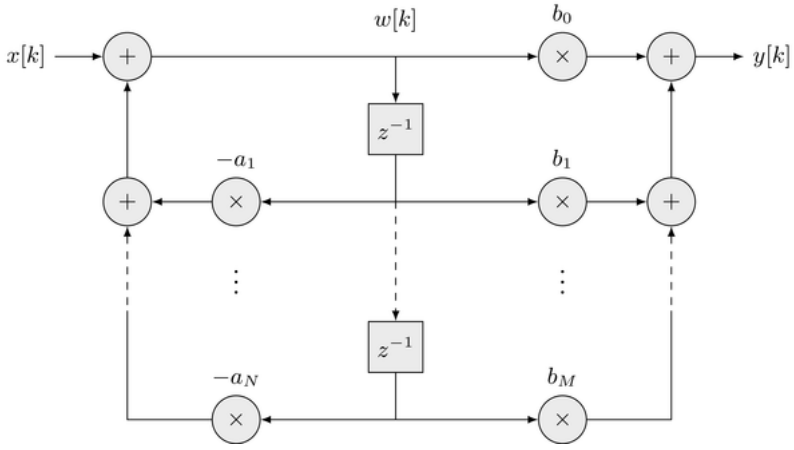


Fig. 7.2: Direct form II filter

Other cases with $N \neq M$ can be considered for by setting coefficients to zero. This form is a canonical structure since it only requires N unit delays for a recursive filter of order N. The output signal $y[k]$ for the direct form II

is computed by the following equations

$$w[k] = x[k] + \sum_{n=1}^{N} -a_n \, w[k-n] \tag{7.3}$$

$$y[k] = \sum_{m=0}^{M} b_m \, w[k-m] \tag{7.4}$$

The samples $w[k-m]$ are termed *state* (variables) of a digital filter.

7.2.3 Transposed Direct Form II

The block diagrams above can be interpreted as linear signal flow graphs. The theory of these graphs provides useful transformations into different forms which preserve the overall transfer function. Of special interest is the *transposition* or *reversal* of a graph which can be achieved by

- exchanging in- and output,
- exchanging signal split and summation points, and
- reversing the directions of the signal flows.

Applying this procedure to the direct form II shown above for $N = M$ yields the transposed direct form II

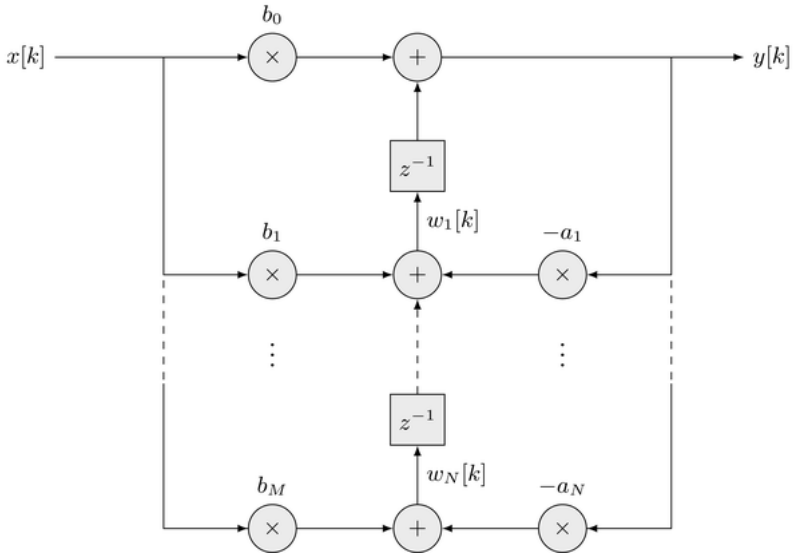


Fig. 7.3: Transposed direct form II filter

The output signal of the transposed direct form II is given as

$$y[k] = b_0 x[k] + \sum_{m=1}^{M} b_m x[k-n] - \sum_{n=1}^{N} a_n y[k-n]$$

Using the signal before the n-th delay unit as internal state $w_n[k]$ we can reformulate this into a set of difference equations for computation of the output signal

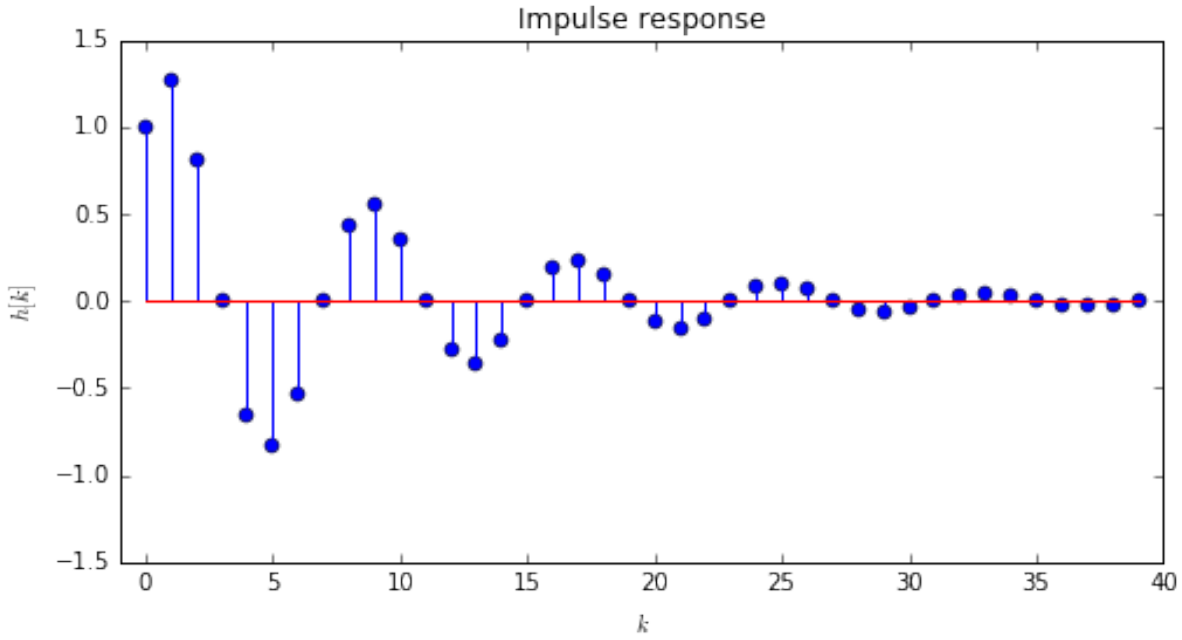
$$w_n[k] = \begin{cases} w_{n+1}[k-1] - a_n y[k] + b_n x[k] & \text{for } n = 0, 1, \ldots, N-1 \\ -a_N y[k] + b_N x[k] & \text{for } n = N \end{cases} \tag{7.5}$$

$$y[k] = w_1[k-1] + b_0 x[k] \tag{7.6}$$

7.2.4 Example

The following example illustrates the computation of the impulse response $h[k]$ of a 2nd-order recursive system using the transposed direct form II as realized by `scipy.signal.lfilter`.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig

        p = 0.90*np.exp(-1j*np.pi/4)
        a = np.poly([p, np.conj(p)])  # denominator coefficients
        b = [1, 0, 0]  # numerator coefficients
        N = 40  # number of samples

        # generate input signal (=Dirac impulse)
        k = np.arange(N)
        x = np.where(k==0, 1.0, 0.0)

        # filter signal using transposed direct form II
        h = sig.lfilter(b, a, x)

        # plot output signal
        plt.figure(figsize=(8, 4))
        plt.stem(h)
        plt.title('Impulse response')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$h[k]$')
        plt.axis([-1, N, -1.5, 1.5]);
```



7.3 Cascaded Structures

The realization of recursive filters with a high order may be subject to numerical issues. For instance, when the coefficients span a wide amplitude range, their quantization may require a small quantization step or may impose a large relative error for small coefficients. The basic concept of cascaded structures is to decompose a high order filter into a cascade of lower order filters, typically first and second order recursive filters.

7.3.1 Decomposition into Second-Order Sections

The rational transfer function $H(z)$ of a linear time-invariant (LTI) recursive system can be *expressed by its zeros and poles* as

$$H(z) = \frac{b_M}{a_N} \cdot \frac{\prod_{\mu=1}^{P}(z - z_{0\mu})^{m_\mu}}{\prod_{\nu=1}^{Q}(z - z_{\infty\nu})^{n_\nu}}$$

where $z_{0\mu}$ and $z_{\infty\nu}$ denote the μ-th zero and ν-th pole of degree m_μ and n_ν of $H(z)$, respectively. The total number of zeros and poles is denoted by P and Q.

The poles and zeros of a real-valued filter $h[k] \in \mathbb{R}$ are either single real valued or conjugate complex pairs. This motivates to split the transfer function into

- first order filters constructed from a single pole and zero
- second order filters constructed from a pair of conjugated complex poles and zeros

Decomposing the transfer function into these two types by grouping the poles and zeros into single poles/zeros and conjugate complex pairs of poles/zeros results in

$$H(z) = K \cdot \prod_{\eta=1}^{S_1} \frac{(z - z_{0\eta})}{(z - z_{\infty\eta})} \cdot \prod_{\eta=1}^{S_2} \frac{(z - z_{0\eta})(z - z_{0\eta}^*)}{(z - z_{\infty\eta})(z - z_{\infty\eta}^*)}$$

where K denotes a constant and $S_1 + 2S_2 = N$ with N denoting the order of the system. The cascade of two systems results in a multiplication of their transfer functions. Above decomposition represents a cascade of first- and second-order recursive systems. The former can be treated as a special case of second-order recursive systems. The decomposition is therefore known as decomposition into second-order sections (SOSs) or biquad filters. Using a cascade of SOSs the transfer function of the recursive system can be rewritten as

$$H(z) = \prod_{\mu=1}^{S} \frac{b_{0,\mu} + b_{1,\mu}\, z^{-1} + b_{2,\mu}\, z^{-2}}{1 + a_{1,\mu}\, z^{-1} + a_{2,\mu}\, z^{-2}}$$

where $S = \lceil \frac{N}{2} \rceil$ denotes the total number of SOSs. These results state that any real valued system of order $N > 2$ can be decomposed into SOSs. This has a number of benefits

- quantization effects can be reduced by sensible grouping of poles/zeros, e.g. such that the spanned amplitude range of the filter coefficients is limited
- A SOS may be extended by a gain factor to further reduce quantization effects by normalization of the coefficients
- efficient and numerically stable SOSs serve as generic building blocks for higher-order recursive filters

7.3.2 Example

The following example illustrates the decomposition of a higher-order recursive Butterworth lowpass filter into a cascade of second-order sections.

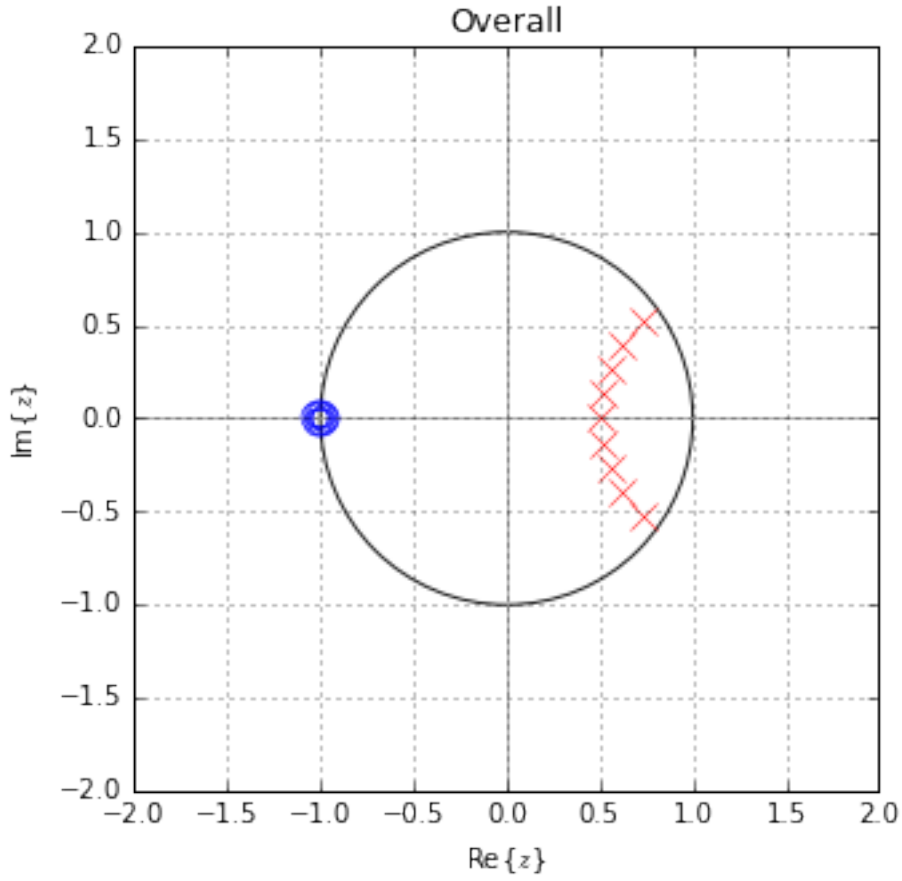
```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        from matplotlib.markers import MarkerStyle
        from matplotlib.patches import Circle
        import scipy.signal as sig

        N = 9  # order of recursive filter

        def zplane(z, p):

            ax = plt.gca()
```

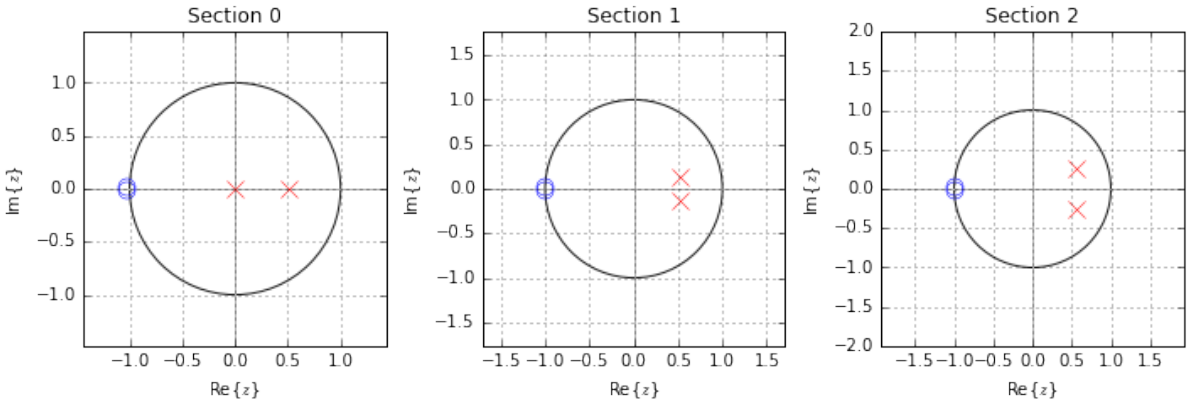
```
    plt.hold(True)

    unit_circle = Circle((0,0), radius=1, fill=False,
                         color='black', ls='solid', alpha=0.9)
    ax.add_patch(unit_circle)
    ax.axvline(0, color='0.7')
    ax.axhline(0, color='0.7')
    plt.axis('equal')
    plt.xlim((-2, 2))
    plt.ylim((-2, 2))
    plt.grid()
    plt.xlabel(r'Re{$z$}')
    plt.ylabel(r'Im{$z$}')

    ax.plot(np.real(z), np.imag(z), 'bo', fillstyle='none', ms = 10)
    ax.plot(np.real(p), np.imag(p), 'rx', fillstyle='none', ms = 10)

    plt.hold(False)


# design filter
b, a = sig.butter(N, 0.2)
# decomposition into SOS
sos = sig.tf2sos(b, a, pairing='nearest')


# print filter coefficients
print('Coefficients of the recursive part')
print(['%1.2f'%ai for ai in a])
print('\n')
print('Coefficients of the recursive part of the SOSs')
print('Section \t a1 \t\t a2')
for n in range(sos.shape[0]):
    print('%d \t\t %1.5f \t %1.5f'%(n, sos[n, 4], sos[n, 5]))

# plot poles and zeros
plt.figure(figsize=(5,5))
zplane(np.roots(b), np.roots(a))
plt.title('Overall')

for n in range(sos.shape[0]):
    if not n%3:
        plt.figure(figsize=(10, 3.5))
    plt.subplot(131+n%3)
    zplane(np.roots(sos[n, 0:3]), np.roots(sos[n, 3:6]))
    plt.title('Section %d'%n)
    plt.tight_layout()

# compute and plot frequency response of sections
plt.figure(figsize=(10,5))
for n in range(sos.shape[0]):
    Om, H = sig.freqz(sos[n, 0:3], sos[n, 3:6])
    plt.plot(Om, 20*np.log10(np.abs(H)), label=r'Section $n=%d$'%n)
    plt.hold(True)
plt.hold(False)
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$|H_n(e^{j \Omega})|$ in dB')
plt.legend()
```

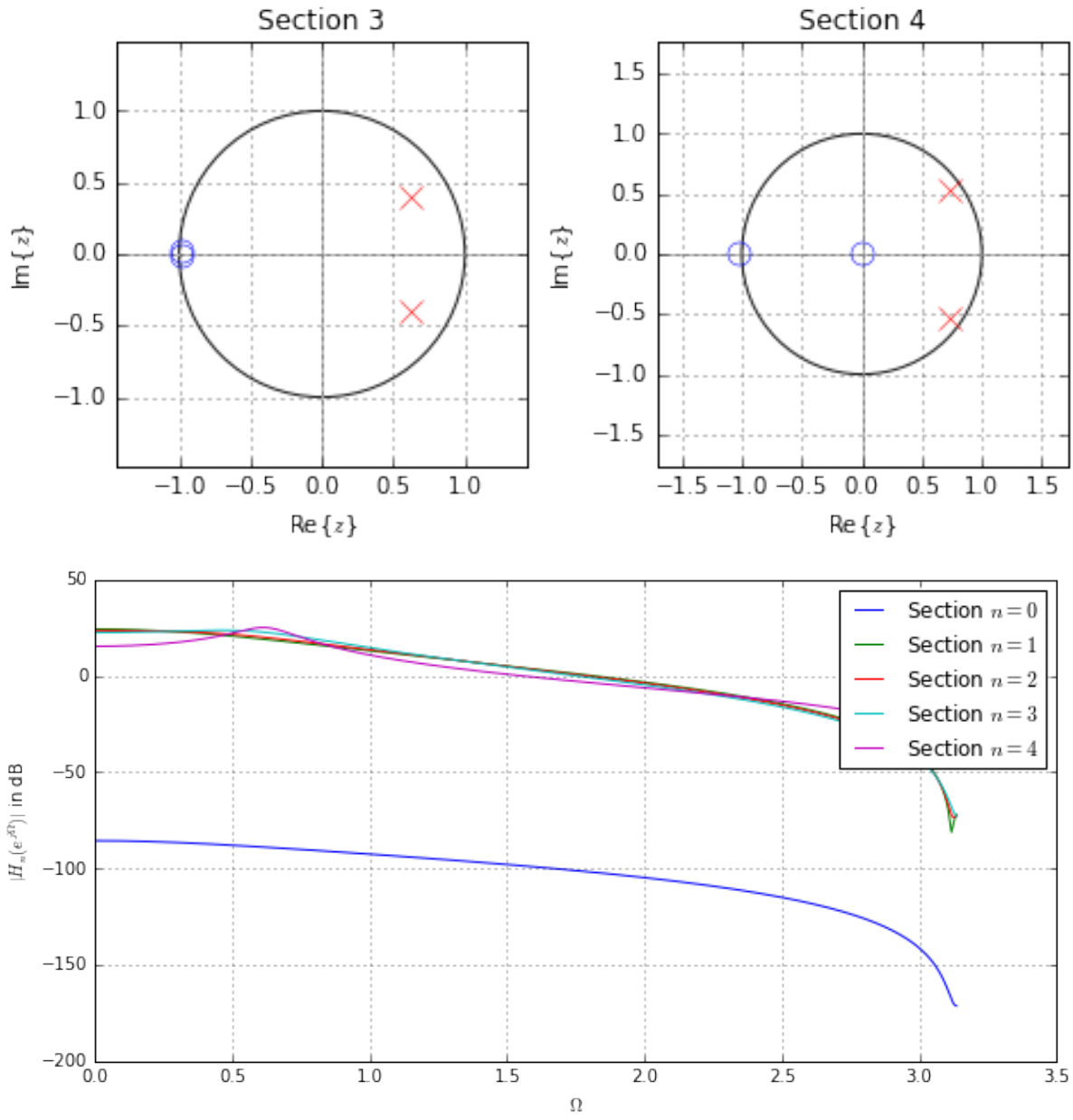
```
        plt.grid();
```

```
Coefficients of the recursive part
['1.00', '-5.39', '13.38', '-19.96', '19.62', '-13.14', '5.97', '-1.78', '0.31', '-0.02'
```

```
Coefficients of the recursive part of the SOSs
Section          a1              a2
0                -0.50953        0.00000
1                -1.04232        0.28838
2                -1.11568        0.37905
3                -1.25052        0.54572
4                -1.46818        0.81477
```







Exercise

- What amplitude range is spanned by the filter coefficients?
- What amplitude range is spanned by the SOS coefficients?
- Change the pole/zero grouping strategy from `pairing='nearest'` to `pairing='keep_odd'`. What changes?
- Increase the order `N` of the filter. What changes?

7.4 Quantization of Filter Coefficients

The finite numerical resolution of digital number representations has impact on the properties of filters, as already discussed for *non-recursive filters*. The quantization of coefficients, state variables, algebraic operations and signals plays an important role in the design of recursive filters. Compared to non-recursive filters, the impact of quantization is often more prominent due to the feedback. Severe degradations from the desired characteristics and instability are potential consequences of a finite word length in practical implementations.

A recursive filter of order $N \geq 2$ can be decomposed into second-order sections (SOS). Due to the grouping of poles/zeros to filter coefficients with a limited amplitude range, a realization by cascaded SOS is favorable in practice. We therefore limit our investigation of quantization effects to SOS. The transfer function of a SOS is given as

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

This can be *split into a non-recursive part and a recursive part*. The quantization effects of non-recursive filters have already been discussed. We therefore focus here on the recursive part given by the transfer function

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

This section investigates the consequences of quantization in recursive filters. As for non-recursive filters, we first take a look at the quantization of filter coefficients. The structure used for the realization of the filter has impact on the quantization effects. We begin with the direct form followed by the coupled form, as example for an alternative structure.

7.4.1 Direct Form

Above transfer function of the recursive part of a SOS can be rewritten in terms of its complex conjugate poles z_∞ and z_∞^* as

$$H(z) = \frac{1}{(z - z_\infty)(z - z_\infty^*)} = \frac{z^{-2}}{1 \underbrace{-2r\cos(\varphi)}_{a_1} z^{-1} + \underbrace{r^2}_{a_2} z^{-2}}$$

where $r = |z_\infty|$ and $\varphi = \arg\{z_\infty\}$ denote the absolute value and phase of the pole z_∞, respectively. Let's assume a *linear uniform quantization* of the coefficients a_1 and a_2 with quantization step Q. Discarding clipping, the following relations for the locations of the poles can be found

$$r_n = \sqrt{n \cdot Q} \tag{7.7}$$

$$\varphi_{nm} = \arccos\left(\sqrt{\frac{m^2 Q}{4n}}\right) \tag{7.8}$$

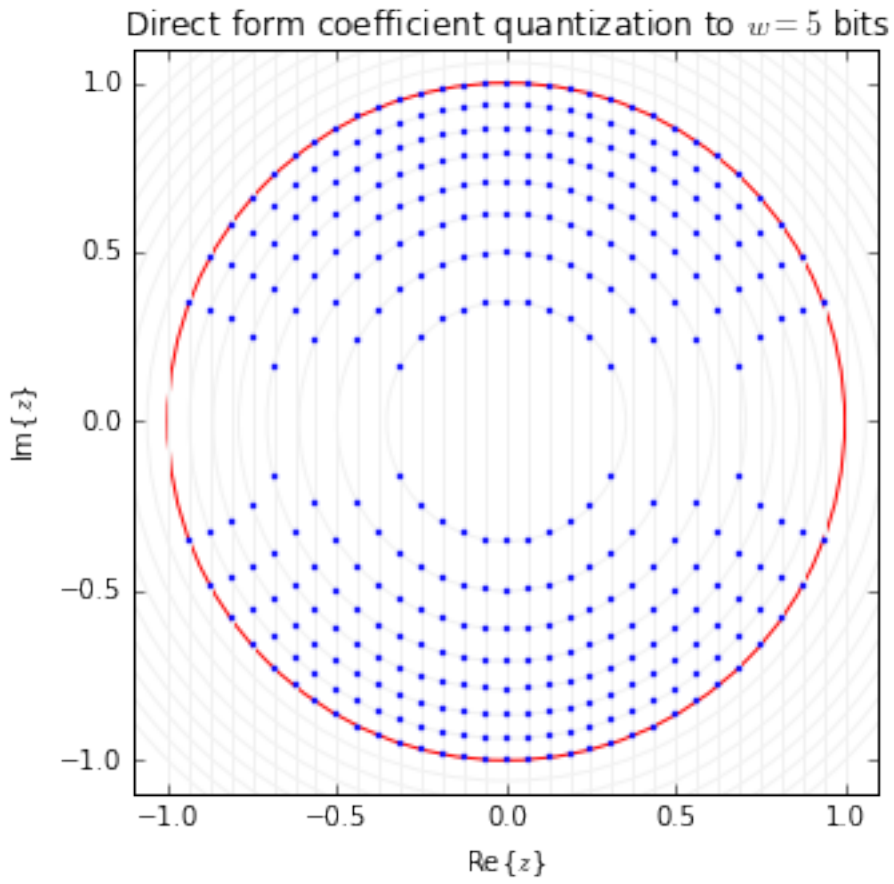
for $n \in \mathbb{N}_0$ and $m \in \mathbb{Z}$. Quantization of the filter coefficients a_1 and a_2 into a finite number of amplitude values leads to a finite number of pole locations. In the z-plane the possible pole locations are given by the intersections of

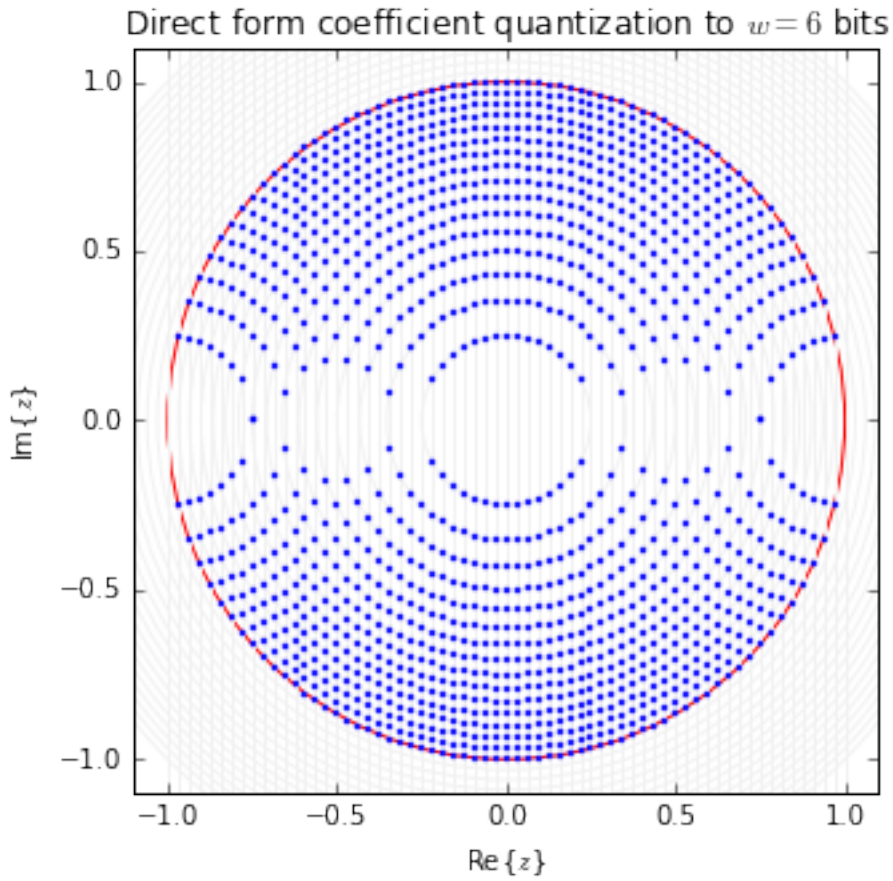
- circles whose radii r_n are given by $r_n = \sqrt{n \cdot Q}$ with
- equidistant vertical lines which intersect the horizontal axis at $\frac{1}{2} m \cdot Q$.

The finite number of pole locations may lead to deviations from a desired filter characteristic since a desired pole location is moved to the next possible pole location. The filter may even get unstable, when poles are moved outside the unit circle. For illustration, the resulting pole locations for a SOS realized in direct form are computed and plotted.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        from matplotlib.patches import Circle
        import scipy.signal as sig
        import itertools


        def compute_pole_locations(Q):
            a1 = np.arange(-2, 2+Q, Q)
            a2 = np.arange(0, 1+Q, Q)
```

```
        p = np.asarray([np.roots([1, n, m]) for (n,m) in itertools.product(a1, a2)])
        p = p[np.imag(p)!=0]

        return p

    def plot_pole_locations(p, Q):
        ax = plt.gca()
        for n in np.arange(np.ceil(2/Q)+1):
            circle = Circle((0,0), radius=np.sqrt(n*Q), fill=False, color='black', l
            ax.add_patch(circle)
            ax.axvline(.5*n*Q, color='0.95')
            ax.axvline(-.5*n*Q, color='0.95')

        unit_circle = Circle((0,0), radius=1, fill=False, color='red', ls='solid')
        ax.add_patch(unit_circle)

        plt.plot(np.real(p), np.imag(p), 'b.', ms = 4)
        plt.xlabel(r'Re{$z$}')
        plt.ylabel(r'Im{$z$}')
        plt.axis([-1.1, 1.1, -1.1, 1.1])

    # compute and plot pole locations
    for w in [5,6]:
        Q = 2/(2**(w-1))  # quantization stepsize
        plt.figure(figsize=(5, 5))
        p = compute_pole_locations(Q)
        plot_pole_locations(p, Q)
        plt.title(r'Direct form coefficient quantization to $w=%d$ bits'%w)
```





Exercise

- What consequences has the distribution of pole locations on the desired characteristics of a filter for e.g. low/high frequencies?

7.4.2 Coupled Form

Besides the quantization step Q, the pole distribution depends also on the topology of the filter. In order to gain a different distribution of pole locations after quantization, one has to derive structures where the coefficients of the multipliers are given by other values than the direct form coefficients a_1 and a_2.

One of these alternative structures is the coupled form (also known as Gold & Rader structure)

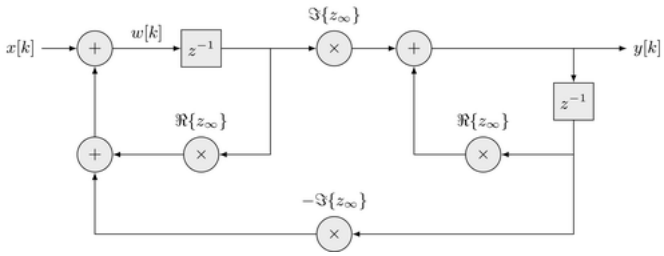


Fig. 7.4: Coupled form second order section

where $\Re\{z_\infty\} = r \cdot \cos\varphi$ and $\Im\{z_\infty\} = r \cdot \sin\varphi$ denote the real- and imaginary part of the complex pole z_∞, respectively. Analysis of the structure reveals its difference equation as

$$w[k] = x[k] + \Re\{z_\infty\}\, w[k-1] - \Im\{z_\infty\}\, y[k-1] \tag{7.9}$$

$$y[k] = \Im\{z_\infty\}\, w[k-1] + \Re\{z_\infty\}\, y[k-1] \tag{7.10}$$

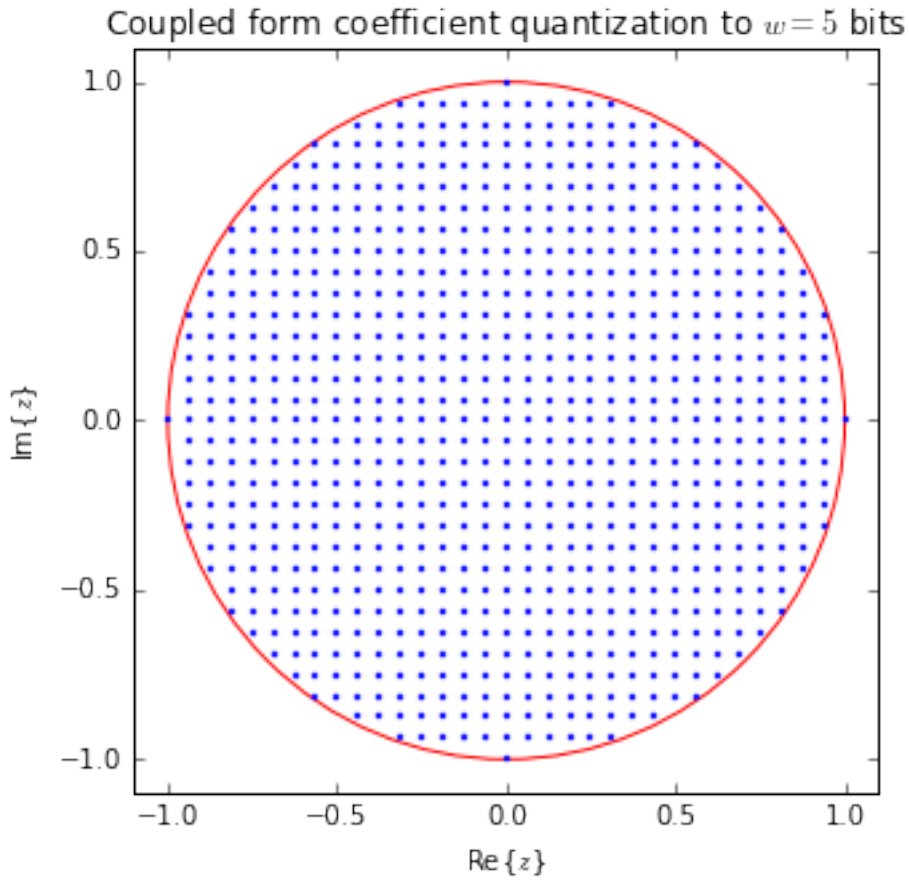
and its transfer function as

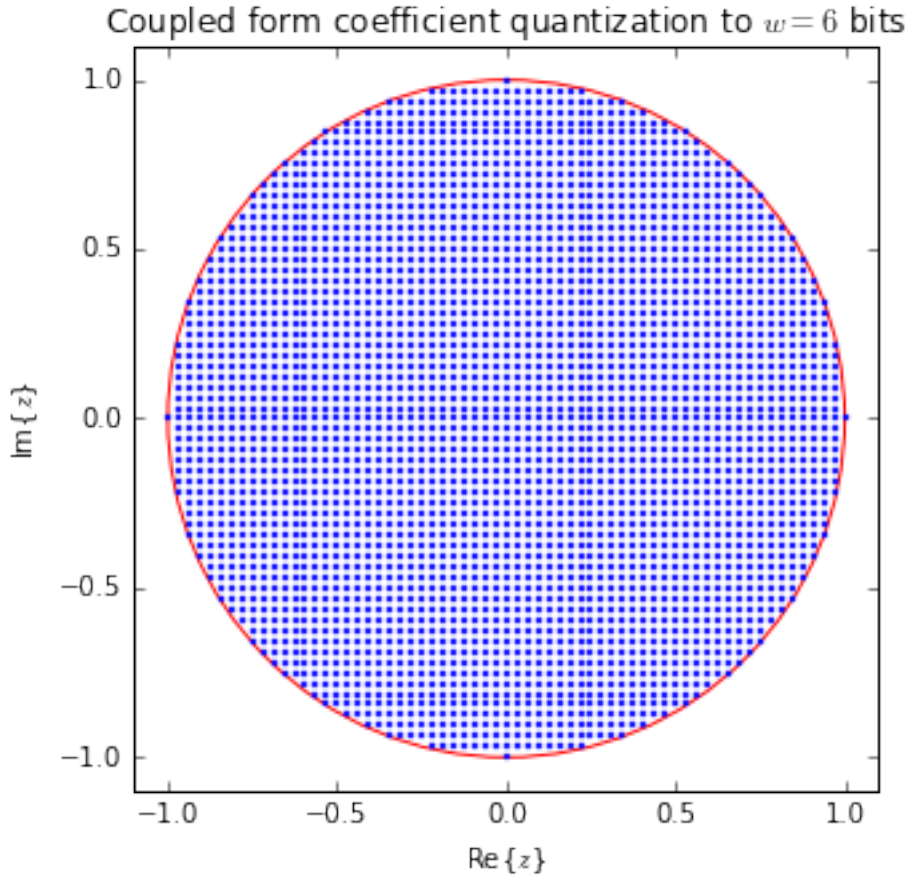
$$H(z) = \frac{\Im\{z_\infty\}\, z^{-1}}{1 - 2\Re\{z_\infty\}\, z^{-1} + (\Re\{z_\infty\}^2 + \Im\{z_\infty\}^2)\, z^{-2}}$$

Note that the numerator of the transfer function differs from the recursive only SOS given above. However, this can be considered in the design of the transfer function of a general SOS.

The real- and imaginary part of the pole z_∞ occur directly as coefficients for the multipliers in the coupled form. Quantization of these coefficients results therefore in a Cartesian grid of possible pole locations in the z-plane. This is illustrated in the following.

```
In [2]: def compute_pole_locations(w):
            Q = 1/(2**(w-1))  # quantization stepsize
            a1 = np.arange(-1, 1+Q, Q)
            a2 = np.arange(-1, 1+Q, Q)

            p = np.asarray([n+1j*m for (n,m) in itertools.product(a1, a2) if n**2+m**2 <

            return p

        def plot_pole_locations(p):
            ax = plt.gca()

            unit_circle = Circle((0,0), radius=1, fill=False, color='red', ls='solid')
            ax.add_patch(unit_circle)

            plt.plot(np.real(p), np.imag(p), 'b.', ms = 4)
            plt.xlabel(r'Re{$z$}')
            plt.ylabel(r'Im{$z$}')
            plt.axis([-1.1, 1.1, -1.1, 1.1])


        # compute and plot pole locations
        for w in [5,6]:
            plt.figure(figsize=(5, 5))
            p = compute_pole_locations(w)
            plot_pole_locations(p)
            plt.title(r'Coupled form coefficient quantization to $w=%d$ bits'%w)
```


Coupled form coefficient quantization to $w = 5$ bits
1.0
0.5
0.0
−0.5
−1.0
Im{z}
−1.0
−0.5
0.0
0.5
1.0
Re{z}


Coupled form coefficient quantization to $w = 6$ bits
1.0
0.5
0.0
−0.5
−1.0
Im{z}
−1.0
−0.5
0.0
0.5
1.0
Re{z}

Excercise

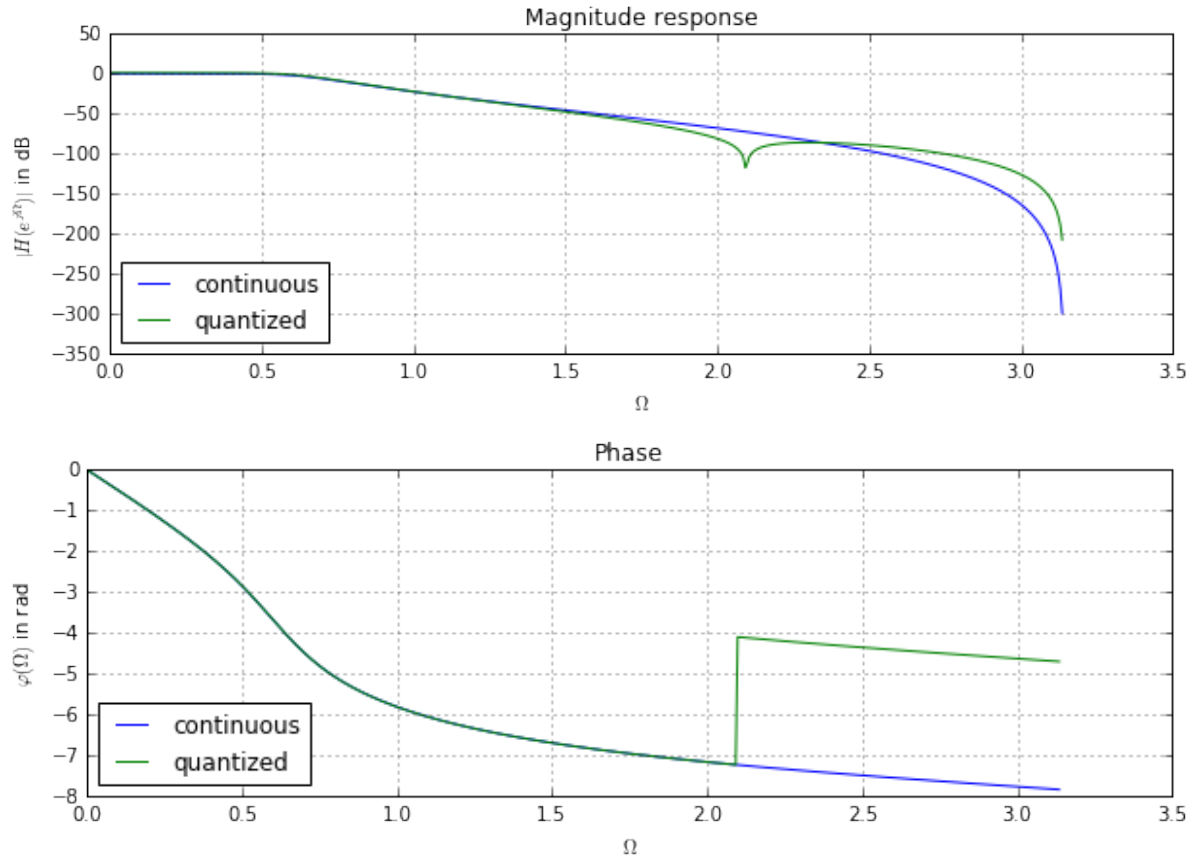
- What is the benefit of this representation in comparison to the direct from discussed in the previous section?

7.4.3 Example

The following example illustrates the effects of coefficient quantization for a recursive Butterworth filter realized in cascaded SOSs in transposed direct form II.

```
In [3]: w = 12  # wordlength of filter coefficients
        N = 5  # order of filter

        def uniform_midtread_quantizer(x, w, xmin=1):
            # quantization step
            Q = xmin/(2**(w-1))
            # limiter
            x = np.copy(x)
            idx = np.where(x <= -xmin)
            x[idx] = -1
            idx = np.where(x > xmin - Q)
            x[idx] = 1 - Q
            # linear uniform quantization
            xQ = Q * np.floor(x/Q + 1/2)

            return xQ

        # coefficients of recursive filter
        b, a = sig.butter(N, 0.2, 'low')
        # decomposition into SOS
        sos = sig.tf2sos(b, a, pairing='nearest')
        sos = sos/np.amax(np.abs(sos))
        # quantization of SOS coefficients
        sosq = uniform_midtread_quantizer(sos, w, xmin=2)
        # compute overall transfer function of (quantized) filter
        H = np.ones(512)
        Hq = np.ones(512)
        for n in range(sos.shape[0]):
            Om, Hn = sig.freqz(sos[n, 0:3], sos[n, 3:6])
            H = H * Hn
            Om, Hn = sig.freqz(sosq[n, 0:3], sosq[n, 3:6])
            Hq = Hq * Hn


        # plot magnitude responses
        plt.figure(figsize=(10, 3))
        plt.plot(Om, 20 * np.log10(abs(H)), label='continuous')
        plt.plot(Om, 20 * np.log10(abs(Hq)), label='quantized')
        plt.title('Magnitude response')
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$|H(e^{j \Omega})|$ in dB')
        plt.legend(loc=3)
        plt.grid()
        # plot phase responses
        plt.figure(figsize=(10, 3))
        plt.plot(Om, np.unwrap(np.angle(H)), label='continuous')
        plt.plot(Om, np.unwrap(np.angle(Hq)), label='quantized')
        plt.title('Phase')
        plt.xlabel(r'$\Omega$')
```

```
    plt.ylabel(r'$\varphi (\Omega)$ in rad')
    plt.legend(loc=3)
    plt.grid()
```



Exercise

- Decrease the word length `w` of the filter. What happens? At what word length does the filter become unstable?
- Increase the order `N` of the filter for a fixed word length `w`. What happens?

7.5 Quantization of Variables and Operations

As for *non-recursive filters*, the practical realization of recursive filters may suffer from the quantization of variables and algebraic operations. The effects of coefficient quantization were already discussed. This section takes a look at the quantization of variables. We limit the investigations to the recursive part of a second-order section (SOS), since any recursive filter of order $N \geq 2$ can be decomposed into SOSs.

The computation of the output signal $y[k] = \mathcal{H}\{x[k]\}$ by a difference equation involves a number of multiplications and additions. As discussed already for *non-recursive filters*, multiplying two numbers in a binary representation (e.g. [two's complement](https://en.wikipedia.org/wiki/Two's_complement) or floating point) requires requantization of the result to keep the word length constant. The addition of two numbers may fall outside the maximum/minimum values of the representation and may suffer from clipping.

The resulting round-off and clipping errors depend on the number and sequence of algebraic operations. These depend on the structure used for implementation of the SOSs. For ease of illustration we limit our discussion to the direct form I and II. Similar insights can be achieved in a similar manner for other structures.

7.5.1 Analysis of Round-Off Errors

Round-off errors are a consequence of reducing the word length after a multiplication. In order to investigate the influence of these errors on a recursive filter, the statistical model for *round-off errors in multipliers* as introduced for non-recursive filters is used. We furthermore neglect clipping.

The difference equation for the recursive part of a SOS realized in direct form I or II is given as

$$y[k] = x[k] - a_1\, y[k-1] - a_2\, y[k-2]$$

where $a_0 = 1$, a_1 and a_2 denote the coefficients of the recursive part. Introducing the requantization after the multipliers into the difference equation yields the output signal $y_Q[k]$

$$y_Q[k] = x[k] - \mathcal{Q}\{a_1\, y[k-1]\} - \mathcal{Q}\{a_2\, y[k-2]\}$$

where $\mathcal{Q}\{\cdot\}$ denotes the requantizer. Requantization is a non-linear process which results in a requantization error. If the value to be requantized is much larger that the quantization step Q, the average statistical properties of this error can be modeled as additive uncorrelated white noise. Introducing the error into above difference equation gives

$$y_Q[k] = x[k] - a_1\, y[k-1] - e_1[k] - a_2\, y[k-2] - e_2[k]$$

where the two white noise sources $e_1[k]$ and $e_2[k]$ are assumed to be uncorrelated to each other. This difference equation can be split into a set of two difference equations

$$y_Q[k] = y[k] + e[k] \tag{7.11}$$
$$y[k] = x[k] - a_1\, y[k-1] - a_2\, y[k-2] \tag{7.12}$$
$$e[k] = -e_1[k] - e_2[k] - a_1\, e[k-1] - a_2\, e[k-2] \tag{7.13}$$

The first difference equation computes the desired output signal $y[k]$ as a result of the input signal $x[k]$. The second one the additive error $e[k]$ due to requantization as a result of the requantization error $-(e_1[k] + e_2[k])$ injected into the recursive filter. The power spectral density (PSD) $\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the error $e[k]$ is then given as

$$\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = |H(\mathrm{e}^{\,\mathrm{j}\,\Omega})|^2 \cdot (\Phi_{e_1 e_1}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) + \Phi_{e_2 e_2}(\mathrm{e}^{\,\mathrm{j}\,\Omega}))$$

According to the model for the requantization errors, their PSDs are given as $\Phi_{e_1 e_1}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \Phi_{e_2 e_2}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{Q^2}{12}$. Introducing this together with the transfer function of the SOS yields

$$\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \left| \frac{1}{1 + a_1\, \mathrm{e}^{\,-\mathrm{j}\,\Omega} + a_2\, \mathrm{e}^{\,-\mathrm{j}\,2\,\Omega}} \right|^2 \cdot \frac{Q^2}{6}$$

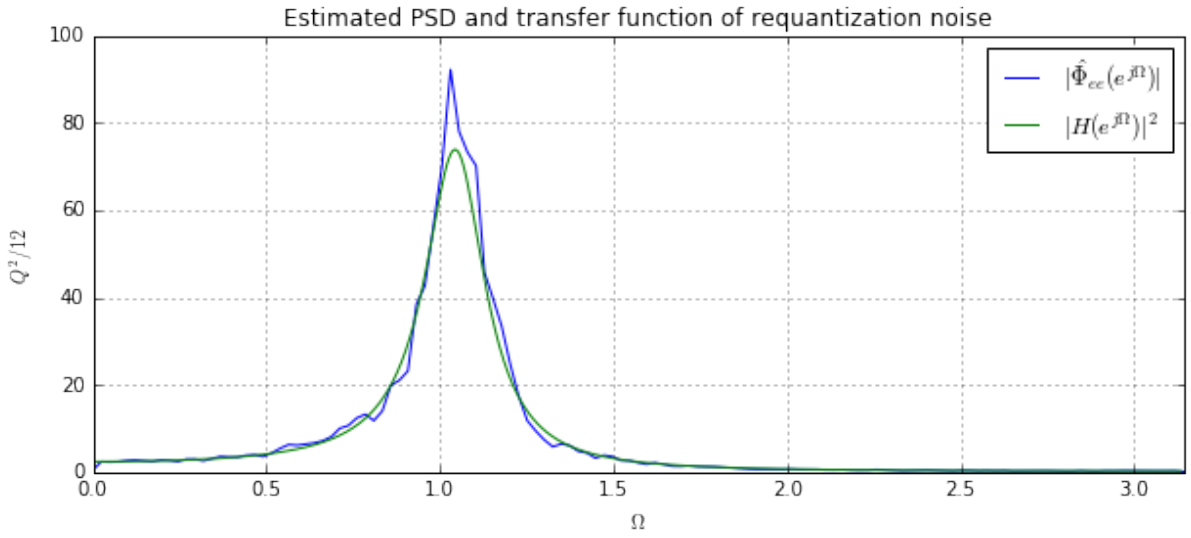
Example

The following example evaluates the error $e[k] = y_Q[k] - y[k]$ for a SOS which only consists of a recursive part. The desired system response $y[k]$ is computed numerically by floating point operations with double precision, $y_Q[k]$ is computed by applying a uniform midtread quantizer after the multiplications. The system is excited by uniformly distributed white noise. Besides the PSD $\Phi_{ee}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$, the signal-to-noise ratio (SNR) $10 \cdot \log_{10}\left(\frac{\sigma_y^2}{\sigma_e^2}\right)$ in dB of the filter is evaluated.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig


        N = 8192  # length of signals
        w = 8  # wordlength for requantization of multiplications

        def uniform_midtread_quantizer(x):
```

```
        # linear uniform quantization
        xQ = Q * np.floor(x/Q + 1/2)

        return xQ


    def no_quantizer(x):

        return x


    def sos_df1(x, a, requantize=None):
        y = np.zeros(len(x)+2)  # initial value appended
        for k in range(len(x)):
            y[k] = x[k] - requantize(a[1]*y[k-1])  - requantize(a[2]*y[k-2])

        return y[0:-2]


    # cofficients of SOS
    p = 0.90*np.array([np.exp(1j*np.pi/3), np.exp(-1j*np.pi/3)])
    a = np.poly(p)
    # quantization step
    Q = 1/(2**(w-1))

    # compute input signal
    x = np.random.uniform(low=-1, high=1, size=N)
    # compute output signals w and w/o requantization
    yQ = sos_df1(x, a, requantize=uniform_midtread_quantizer)
    y = sos_df1(x, a, requantize=no_quantizer)
    # compute requantization error
    e = yQ-y
    # Signal-to-noise ratio
    SNR = 10*np.log10(np.var(y)/np.var(e))
    print('SNR due to requantization: %f dB'%SNR)

    # estimate PSD of requantization error
    nf, Pxx = sig.welch(e, window='hamming', nperseg=256, noverlap=128)
    Pxx = .5*Pxx  # due to normalization in scipy.signal
    Om = 2*np.pi*nf
    # compute frequency response of system
    w, H = sig.freqz([1,0,0], a)


    # plot results
    plt.figure(figsize=(10,4))
    plt.plot(Om, Pxx/Q**2 * 12, 'b', label=r'$|\hat{\Phi}_{ee}(e^{j \Omega})|$')
    plt.plot(w, np.abs(H)**2 * 2, 'g', label=r'$|H(e^{j \Omega})|^2$')
    plt.title('Estimated PSD and transfer function of requantization noise')
    plt.xlabel(r'$\Omega$')
    plt.ylabel(r'$Q^2/12$')
    plt.axis([0, np.pi, 0, 100])
    plt.legend()
    plt.grid();
```

```
SNR due to requantization: 44.778398 dB
```



7.5.2 Small Limit Cycles

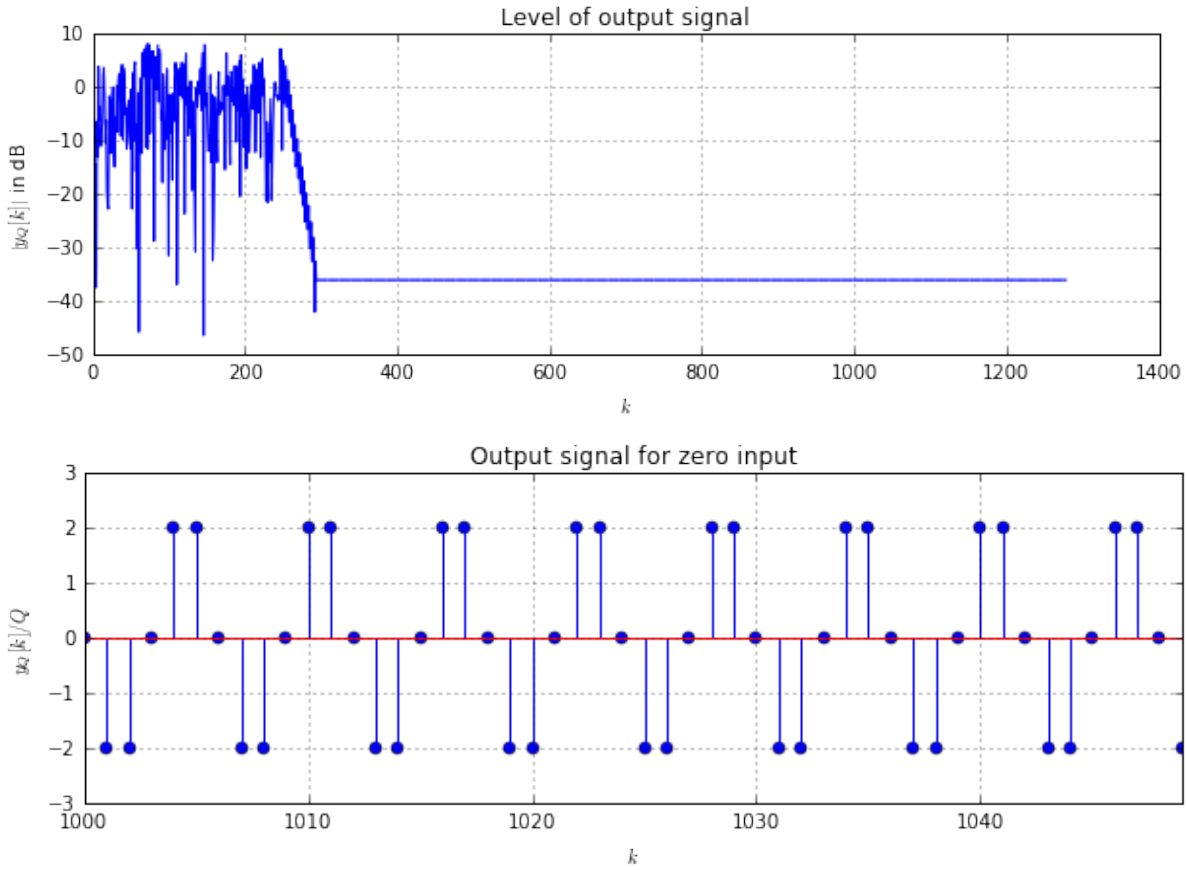
Besides the requantization noise, recursive filters may be subject to periodic oscillations present at the output. These undesired oscillations are termed *limit cycles*. Small limit cycles emerge from the additive round-off noise due to requantization after a multiplication. The feedback in a recursive filter leads to a feedback of the requantization noise. This may lead to a periodic output signal with an amplitude range of some quantization steps Q, even after the input signal is zero. The presence, amplitude and frequency of small limit cycles depends on the location of poles and the structure of the filter. A detailed treatment of this phenomenon is beyond the scope of this notebook and can be found in the literature.

Example

The following example illustrates small limit cycles for the system investigated in the previous example. The input signal is uniformly distributed white noise till time-index $k = 256$ and zero for the remainder.

```
In [2]: # compute input signal
        x = np.random.uniform(low=-1, high=1, size=256)
        x = np.concatenate((x, np.zeros(1024)))
        # compute output signal
        yQ = sos_df1(x, a, requantize=uniform_midtread_quantizer)

        # plot results
        plt.figure(figsize=(10, 3))
        plt.plot(20*np.log10(np.abs(yQ)))
        plt.title('Level of output signal')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$|y_Q[k]|$ in dB')
        plt.grid()

        plt.figure(figsize=(10, 3))
        k = np.arange(1000, 1050)
        plt.stem(k, yQ[k]/Q)
        plt.title('Output signal for zero input')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$y_Q[k] / Q$ ')
        plt.axis([k[0], k[-1], -3, 3])
        plt.grid();
```

```
/opt/local/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
```



Exercise

- Estimate the period of the small limit cycles. How is it related to the poles of the system?
- What amplitude range is spanned?

7.5.3 Large Limit Cycles

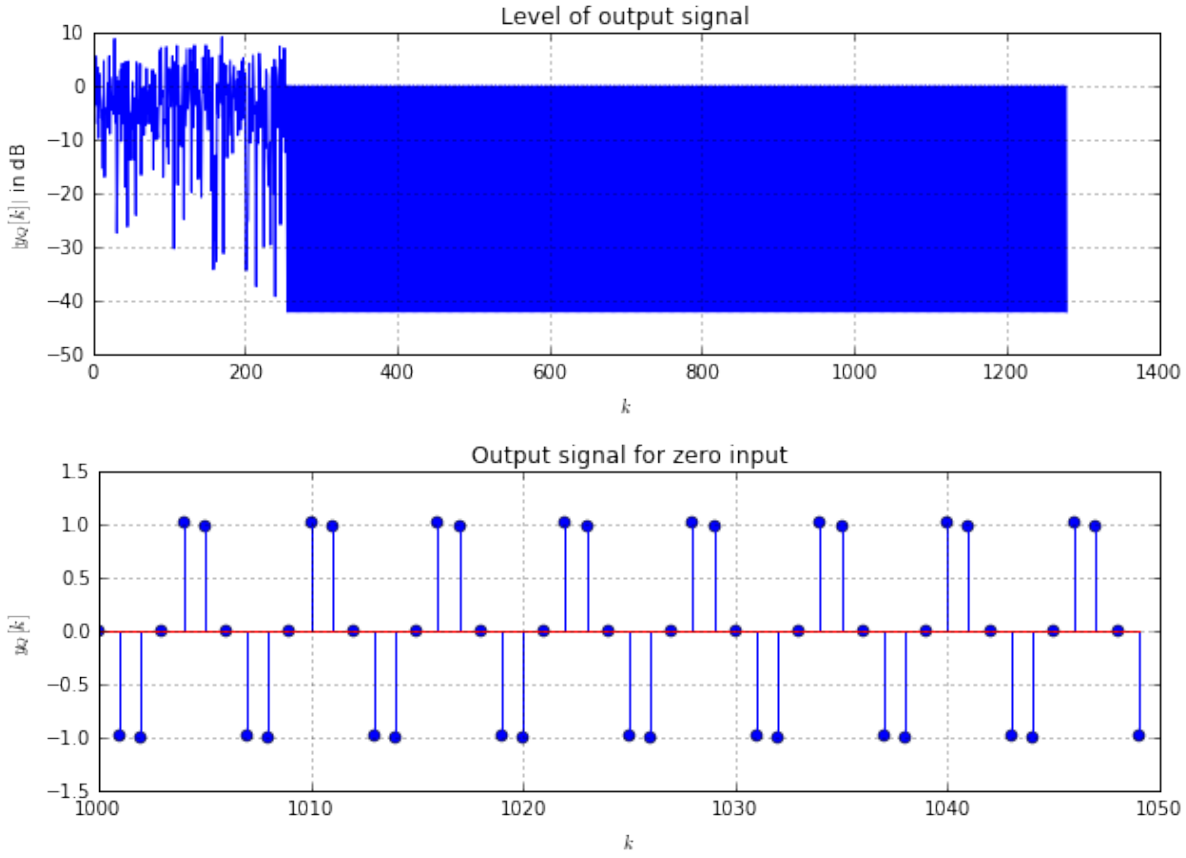
Large limit cycles are periodic oscillations of a recursive filter due to overflows in the multiplications/additions. As for small limit cycles, large limit cycles may be present even after the input signal is zero. Their level is typically in the range of the minimum/maximum value of the requantizer. Large limit cycles should therefore be avoided in a practical implementation. The presence of large limit cycles depends on the scaling of input signal and coefficients, as well as the strategy used to cope for clipping. Amongst others, they can be avoided by proper scaling of the coefficients to prevent overflow. Again, a detailed treatment of this phenomenon is beyond the scope of this notebook and can be found in the literature.

Example

The following example illustrates large limit cycles for the system investigated in the first example. In order to trigger large limit cycles, the coefficients of the filter have been doubled. The input signal is uniformly distributed white noise till time-index $k = 256$ and zero for the remainder.

```
In [3]: def uniform_midtread_quantizer(x, xmin=1):
            # limiter
            x = np.copy(x)
            if x <= -xmin:
                x = -1
            if x > xmin - Q:
                x = 1 - Q
            # linear uniform quantization
```

```
        xQ = Q * np.floor(x/Q + 1/2)

        return xQ


    # compute input signal
    x = np.random.uniform(low=-1, high=1, size=256)
    x = np.concatenate((x, np.zeros(1024)))
    # compute output signal
    yQ = sos_df1(x, 2*a, requantize=uniform_midtread_quantizer)

    # plot results
    plt.figure(figsize=(10, 3))
    plt.plot(20*np.log10(np.abs(yQ)))
    plt.title('Level of output signal')
    plt.xlabel(r'$k$')
    plt.ylabel(r'$|y_Q[k]|$ in dB')
    plt.grid()

    plt.figure(figsize=(10, 3))
    k = np.arange(1000, 1050)
    plt.stem(k, yQ[k])
    plt.title('Output signal for zero input')
    plt.xlabel(r'$k$')
    plt.ylabel(r'$y_Q[k]$ ')
    #plt.axis([k[0], k[-1], -1.1, 1.1])
    plt.grid();
```



Exercise

- Determine the period of the large limit cycles. How is it related to the poles of the system?

Design of Digital Filters

8.1 Design of Non-Recursive Filters using the Window Method

The design of non-recursive filters with a finite impulse response (FIR) is a frequent task in practical applications. The designed filter should approximate a desired frequency response as close as possible. First, the design of causal filters is considered. For many applications the resulting filter should have a linear phase characteristic since this results in a constant (and thus frequency independent) group delay. We therefore specialize the design to causal linear-phase filters.

8.1.1 Causal Filters

Let's assume that the desired frequency characteristics of the filter are given by the frequency response (i.e. the DTFT spectrum) $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. The corresponding impulse response is computed by its inverse discrete-time Fourier transform (IDTFT)

$$h_\mathrm{d}[k] = \frac{1}{2\pi} \int_{-\pi}^{\pi} H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\, \mathrm{e}^{\,\mathrm{j}\,\Omega\, k}\, \mathrm{d}\Omega \tag{8.1}$$

In the general case, $h_\mathrm{d}[k]$ will not be a causal FIR. The Paley-Wiener theorem states, that a causal system $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ can only have zeros at single frequencies. This is not the case for idealized filters, like e.g. the ideal low-pass filter. The basic idea of the window method is to truncate the impulse response $h_\mathrm{d}[k]$ in order to derive a causal FIR filter. This can be achieved by applying a window $w[k]$ of finite length N to $h_\mathrm{d}[k]$

$$h[k] = h_\mathrm{d}[k] \cdot w[k] \tag{8.2}$$

where $h[k]$ denotes the impulse response of the designed filter. Its frequency response $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is given by the multiplication<->convolution theorem of the discrete-time Fourier transform (DTFT)

$$H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \frac{1}{2\pi}\, H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \circledast W(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \tag{8.3}$$

where $W(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ denotes the DTFT of the window function $w[k]$. The frequency response $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ of the filter is given as the periodic convolution of the desired frequency response $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and the frequency response of the window function $W(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. The frequency response $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is equal to the desired frequency response $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ only if $W(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = 2\pi \cdot \delta(\Omega)$. This would require that $w[k] = 1$ for $k = -\infty, \ldots, \infty$. Hence for a window $w[k]$ of finite length deviations from the desired frequency response are to be expected.

In order to investigate the effect of truncation on the frequency response $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$, a particular window is considered. A straightforward choice is the rectangular window $w[k] = \mathrm{rect}_N[k]$ of length N. Its DTFT is given as

$$W(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathrm{e}^{-\mathrm{j}\,\Omega\,\frac{N-1}{2}} \cdot \frac{\sin(\frac{N\,\Omega}{2})}{\sin(\frac{\Omega}{2})} \tag{8.4}$$

The frequency-domain properties of the rectangular window have already been discussed for the leakage effect. The rectangular window features a narrow main lobe at the cost of relative high sidelobe level. The main lobe

gets narrower with increasing length N. The convolution of the desired frequency response with the frequency response of the window function effectively results in smoothing and ringing. While the main lobe will smooth discontinuities of the desired transfer function, the sidelobes result in undesirable ringing effects. The latter can be alleviated by using other window functions. Note that typical window functions decay towards their ends and are symmetric with respect to their center. This may cause problems for desired impulse responses with large values towards their ends.

Example: Causal approximation of ideal low-pass

The design of an ideal low-pass filter using the window technique is illustrated in the following example. For $|\Omega| < \pi$ the transfer function of the ideal low-pass is given as

$$H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \begin{cases} 1 & \text{for } |\Omega| \leq \Omega_\mathrm{c} \\ 0 & \text{otherwise} \end{cases} \tag{8.5}$$

where Ω_c denotes the corner frequency of the low-pass. An inverse DTFT of the desired transfer function yields

$$h_\mathrm{d}[k] = \frac{\Omega_\mathrm{c}}{\pi} \cdot \mathrm{si}[\Omega_\mathrm{c}\, k] \tag{8.6}$$

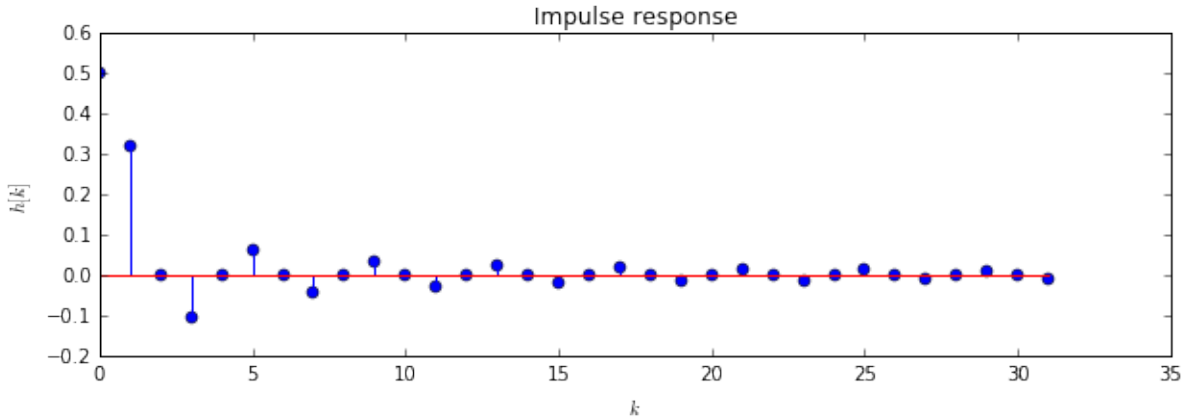
The impulse response $h_\mathrm{d}[k]$ is not causal nor FIR. In order to derive a causal and FIR approximation a rectangular window $w[k]$ of length N is applied

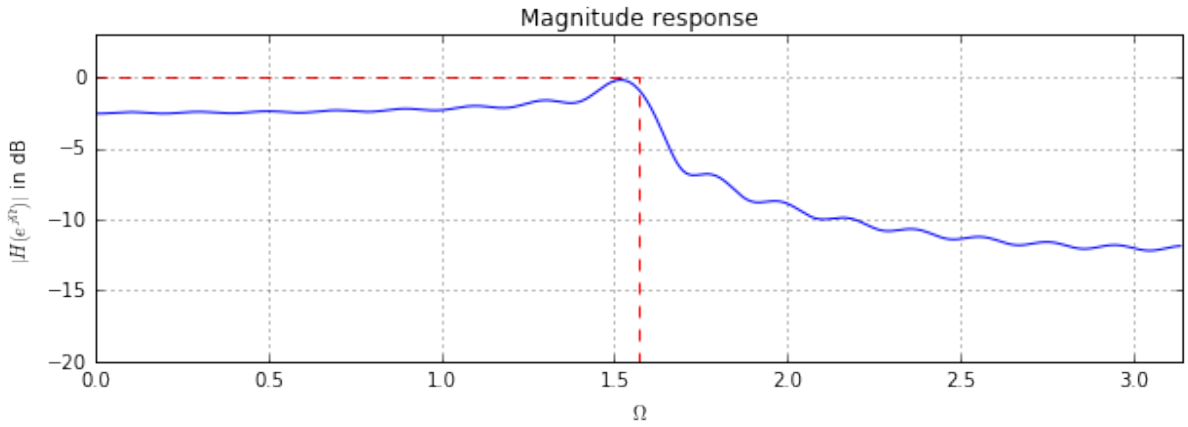
$$h[k] = h_\mathrm{d}[k] \cdot \mathrm{rect}_N[k] \tag{8.7}$$

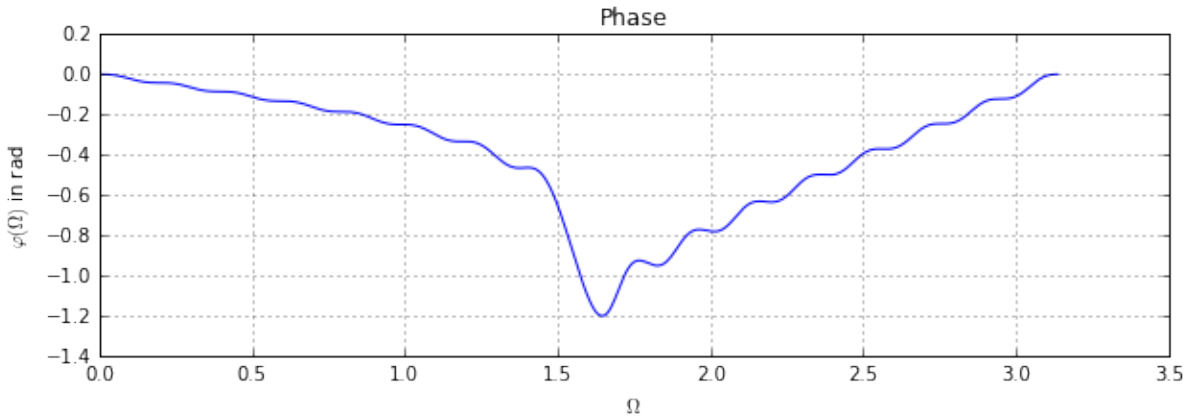
The resulting magnitude and phase response is computed numerically in the following.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig

        N = 32  # length of filter
        Omc = np.pi/2

        # compute impulse response
        k = np.arange(N)
        hd = Omc/np.pi * np.sinc(k*Omc/np.pi)
        # windowing
        w = np.ones(N)
        h = hd * w

        # frequency response
        Om, H = sig.freqz(h)

        # plot impulse response
        plt.figure(figsize=(10, 3))
        plt.stem(h)
        plt.title('Impulse response')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$h[k]$')
        # plot magnitude responses
        plt.figure(figsize=(10, 3))
        plt.plot([0, Omc, Omc], [0, 0, -100], 'r--')
        plt.plot(Om, 20 * np.log10(abs(H)))
        plt.title('Magnitude response')
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$|H(e^{j \Omega})|$ in dB')
        plt.axis([0, np.pi, -20, 3])
        plt.grid()
```

```
# plot phase responses
plt.figure(figsize=(10, 3))
plt.plot(Om, np.unwrap(np.angle(H)))
plt.title('Phase')
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$\varphi (\Omega)$ in rad')
plt.grid()
```







Exercises

- Does the resulting filter have the desired phase?
- Increase the length `N` of the filter. What changes?

8.1.2 Zero-Phase Filters

Above results show that an ideal-low pass cannot be realized very well with the window method. The reason is that an ideal-low pass has zero-phase, as most of the idealized filters.

Lets assume a general zero-phase filter with transfer function $H_d(\mathrm{e}^{\mathrm{j}\,\Omega}) = A(\mathrm{e}^{\mathrm{j}\,\Omega})$ with amplitude $A(\mathrm{e}^{\mathrm{j}\,\Omega}) \in \mathbb{R}$. Its impulse response $h_d[k] = \mathcal{F}_*^{-1}\{H_d(e^{j\Omega}\}$ is conjugate complex symmetric

$$h_d[k] = h_d^*[-k] \tag{8.8}$$

due to the symmetry relations of the DTFT. Hence, a transfer function with zero-phase cannot be realized by a causal non-recursive filter. This observation motivates to replace the zero-phase by a linear-phase in such situations. This is illustrated in the following.

8.1.3 Causal Linear-Phase Filters

The design of non-recursive FIR filters with a linear phase is often desired due to their constant group delay. Let's assume a system with generalized linear phase. For $|\Omega| < \pi$ its transfer function is given as

$$H_\mathrm{d}(\mathrm{e}^{\mathrm{j}\,\Omega}) = A(\mathrm{e}^{\mathrm{j}\,\Omega}) \cdot \mathrm{e}^{-\mathrm{j}\alpha\Omega + \mathrm{j}\beta} \tag{8.9}$$

where $A(\mathrm{e}^{\mathrm{j}\,\Omega}) \in \mathbb{R}$ denotes the amplitude of the filter, α its linear phase and β a constant phase offset. Such a system can be decomposed into two cascaded systems: a zero-phase system with transfer function $A(\mathrm{e}^{\mathrm{j}\,\Omega})$ and an all-pass with phase $\varphi(\Omega) = -\alpha\Omega + \beta$. The linear phase term $-\alpha\Omega$ results in the desired constant group delay $t_g(\Omega) = \alpha$.

The impulse response $h[k]$ of a linear-phase system shows a specific symmetry which can be deduced from the symmetry relations of the DTFT for odd/even symmetry of $H_\mathrm{d}(\mathrm{e}^{\mathrm{j}\,\Omega})$ as

$$h[k] = \pm h[N - 1 - k] \tag{8.10}$$

for $k = 0, 1, \ldots, N-1$ where $N \in \mathbb{N}$ denotes the length of the (finite) impulse response. The transfer function of a linear phase filter is given by its DTFT

$$H_\mathrm{d}(\mathrm{e}^{\mathrm{j}\,\Omega}) = \sum_{k=0}^{N-1} h[k]\, \mathrm{e}^{-\mathrm{j}\,\Omega\,k} \tag{8.11}$$

Introducing the symmetry relations of the impulse response $h[k]$ into the DTFT and comparing the result with above definition of a generalized linear phase system reveals four different types of linear-phase systems. These can be discriminated with respect to their phase and magnitude characteristics

Type	Length N	Impulse Response $h[k]$	Constant Group Delay α in Samples	Constant Phase β	Transfer Function $A(\mathrm{e}^{\mathrm{j}\,\Omega})$
1	odd	$h[k] = h[N-1-k]$	$\alpha = \frac{N-1}{2} \in \mathbb{N}$	$\beta = \{0, \pi\}$	$A(\mathrm{e}^{\mathrm{j}\,\Omega}) = A(\mathrm{e}^{-\mathrm{j}\,\Omega})$, all filter characteristics
2	even	$h[k] = h[N-1-k]$	$\alpha = \frac{N-1}{2} \notin \mathbb{N}$	$\beta = \{0, \pi\}$	$A(\mathrm{e}^{\mathrm{j}\,\Omega}) = A(\mathrm{e}^{-\mathrm{j}\,\Omega})$, $A(\mathrm{e}^{\mathrm{j}\,\pi}) = 0$, only lowpass or bandpass
3	odd	$h[k] = -h[N-1-k]$	$\alpha = \frac{N-1}{2} \in \mathbb{N}$	$\beta = \{\frac{\pi}{2}, \frac{3\pi}{2}\}$	$A(\mathrm{e}^{\mathrm{j}\,\Omega}) = -A(\mathrm{e}^{-\mathrm{j}\,\Omega})$, $A(\mathrm{e}^{\mathrm{j}\,0}) = A(\mathrm{e}^{\mathrm{j}\,\pi}) = 0$, only bandpass
4	even	$h[k] = -h[N-1-k]$	$\alpha = \frac{N-1}{2} \notin \mathbb{N}$	$\beta = \{\frac{\pi}{2}, \frac{3\pi}{2}\}$	$A(\mathrm{e}^{\mathrm{j}\,\Omega}) = -A(\mathrm{e}^{-\mathrm{j}\,\Omega})$, $A(\mathrm{e}^{\mathrm{j}\,0}) = 0$, only highpass or bandpass

These relations have to be considered in the design of a causal linear phase filter. Depending on the desired frequency characteristics $A(\mathrm{e}^{\mathrm{j}\,\Omega})$ the suitable type is chosen. The odd/even length N of the filter and the phase (or group delay) is chosen accordingly for the design of the filter. It is also abovious that a filter with zero-phase $\alpha = 0$, e.g. an ideal low-pass, would result in $N = 1$.

Example: Causal linear-phase approximation of ideal low-pass

We aim at the design of a causal linear-phase low-pass using the window technique. According to the previous example, the desired frequency response has an even symmetry $A(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = A(\mathrm{e}^{\,-\mathrm{j}\,\Omega})$ with $A(\mathrm{e}^{\,\mathrm{j}\,0}) = 1$. This could be realized by a filter of type 1 or 2. We choose type 1, since the resulting filter then exhibits an integer group delay of $t_g(\Omega) = \frac{N-1}{2}$ samples. Consequently the length of the filter N has to be odd.

The impulse response $h_\mathrm{d}[k]$ is given by the inverse DTFT of $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ as

$$h_\mathrm{d}[k] = \frac{\Omega_\mathrm{c}}{\pi} \cdot \mathrm{si}\left[\Omega_\mathrm{c}\left(k - \frac{N-1}{2}\right)\right] \tag{8.12}$$

The impulse response fulfills the desired symmetry for $k = 0, 1, \ldots, N-1$. A causal and FIR approximation is obtained by applying a window function of length N to the impulse response $h_\mathrm{d}[k]$

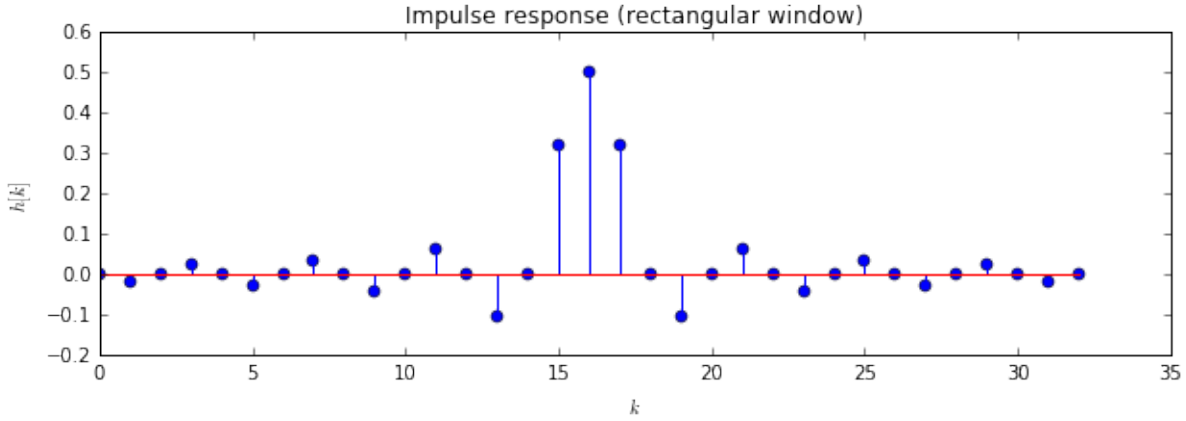
$$h[k] = h_\mathrm{d}[k] \cdot w[k] \tag{8.13}$$

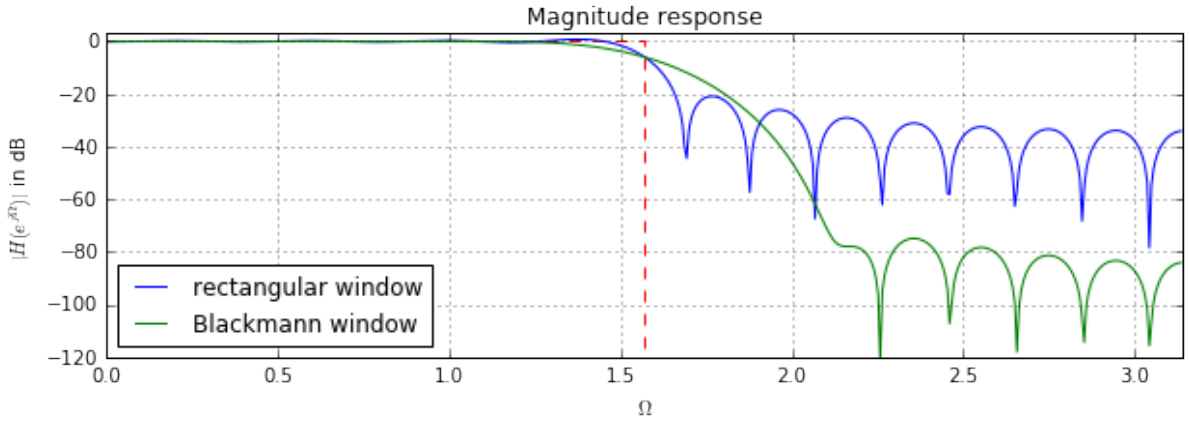
Note that the window function $w[k]$ also has to fulfill the desired symmetries.

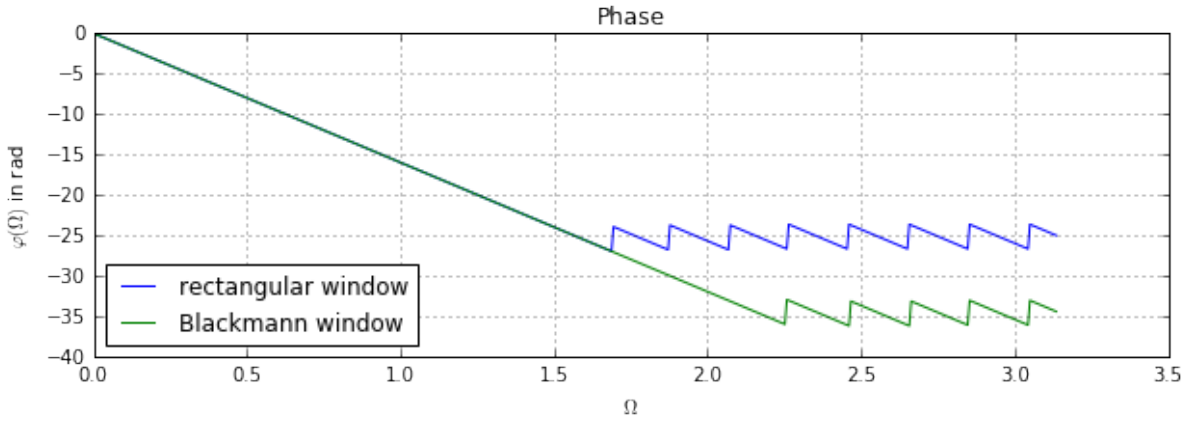
As already outlined, the chosen window determines the properties of the transfer function $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. The spectral properties of commonly applied windows have been discussed previously. The width of the main lobe will generally influence the smoothing of the desired transfer function $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$, while the sidelobes influence the typical ringing artifacts. This is illustrated in the following.

```
In [2]: N = 33  # length of filter
        Omc = np.pi/2

        # compute impulse response
        k = np.arange(N)
        hd = Omc/np.pi * np.sinc((k-(N-1)/2)*Omc/np.pi)
        # windowing
        w1 = np.ones(N)
        w2 = np.blackman(N)
        h1 = hd * w1
        h2 = hd * w2

        # frequency responses
        Om, H1 = sig.freqz(h1)
        Om, H2 = sig.freqz(h2)

        # plot impulse response
        plt.figure(figsize=(10, 3))
        plt.stem(h1)
        plt.title('Impulse response (rectangular window)')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$h[k]$')
        # plot magnitude responses
        plt.figure(figsize=(10, 3))
        plt.plot([0, Omc, Omc], [0, 0, -300], 'r--')
        plt.plot(Om, 20 * np.log10(abs(H1)), label='rectangular window')
        plt.plot(Om, 20 * np.log10(abs(H2)), label='Blackmann window')
        plt.title('Magnitude response')
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$|H(e^{j \Omega})|$ in dB')
        plt.axis([0, np.pi, -120, 3])
        plt.legend(loc=3)
        plt.grid()
        # plot phase responses
        plt.figure(figsize=(10, 3))
        plt.plot(Om, np.unwrap(np.angle(H1)), label='rectangular window')
        plt.plot(Om, np.unwrap(np.angle(H2)), label='Blackmann window')
```

```
plt.title('Phase')
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$\varphi (\Omega)$ in rad')
plt.legend(loc=3)
plt.grid()
```







Exercises

- Does the impulse response fulfill the required symmetries for a type 1 filter?
- Can you explain the differences between the magnitude responses $|H(\mathrm{e}^{\mathrm{j}\,\Omega})|$ for the different window functions?
- What happens if you increase the length `N` of the filter?

8.2 Design of Non-Recursive Filters using the Frequency Sampling Method

For some applications, the desired frequency response is not given at all frequencies but rather at a number of discrete frequencies. For this case, the frequency sampling method provides a solution for the design of non-recursive filters.

8.2.1 The Frequency Sampling Method

Let's assume that the desired transfer function $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is specified at a set of N equally spaced frequencies $\Omega_\mu = \frac{2\pi}{N}\mu$

$$H_\mathrm{d}[\mu] = H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\frac{2\pi}{N}\mu}) \tag{8.14}$$

for $\mu = 0, 1, \ldots, N-1$. The coefficients of a non-recursive filter with finite impulse response (FIR) can then be computed by inverse discrete Fourier transformation (DFT) of $H_\mathrm{d}[\mu]$

$$h[k] = \mathrm{DFT}_N^{-1}\{H_\mathrm{d}[\mu]\} = \frac{1}{N}\sum_{\mu=0}^{N-1} H_\mathrm{d}[\mu]\ \mathrm{e}^{\,\mathrm{j}\,\frac{2\pi}{N}\mu\,k} \tag{8.15}$$

for $k = 0, 1, \ldots, N-1$.

In order to investigate the properties of the designed filter, its transfer function $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is computed. It is given by discrete-time Fourier transformation (DTFT) of its impulse response $h[k]$

$$H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{k=0}^{N-1} h[k]\ \mathrm{e}^{\,-\mathrm{j}\,\Omega\,k} = \sum_{\mu=0}^{N-1} H_\mathrm{d}[\mu] \cdot \frac{1}{N}\sum_{k=0}^{N-1} \mathrm{e}^{\,-\mathrm{j}\,k\,(\Omega - \frac{2\pi}{N}\,\mu)} \tag{8.16}$$

When comparing this result with the *interpolation of a DFT*, it can be concluded that $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is yielded by interpolation of the desired transfer function $H_\mathrm{d}[\mu]$

$$H(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \sum_{\mu=0}^{N-1} H_\mathrm{d}[\mu] \cdot \mathrm{e}^{\,-\mathrm{j}\,\frac{(\Omega - \frac{2\pi}{N}\mu)(N-1)}{2}} \cdot \mathrm{psinc}_N(\Omega - \frac{2\pi}{N}\mu) \tag{8.17}$$

where $\mathrm{psinc}_N(\cdot)$ denotes the N-th order periodic sinc function.

Both the transfer function of the filter $H(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and the desired transfer function $H_\mathrm{d}[\mu]$ are equal at the specified frequencies $\Omega_\mu = \frac{2\pi}{N}\mu$. Values in between adjacent Ω_μ are interpolated by the periodic sinc function. This is illustrated in the following.

Example: Approximation of an ideal low-pass

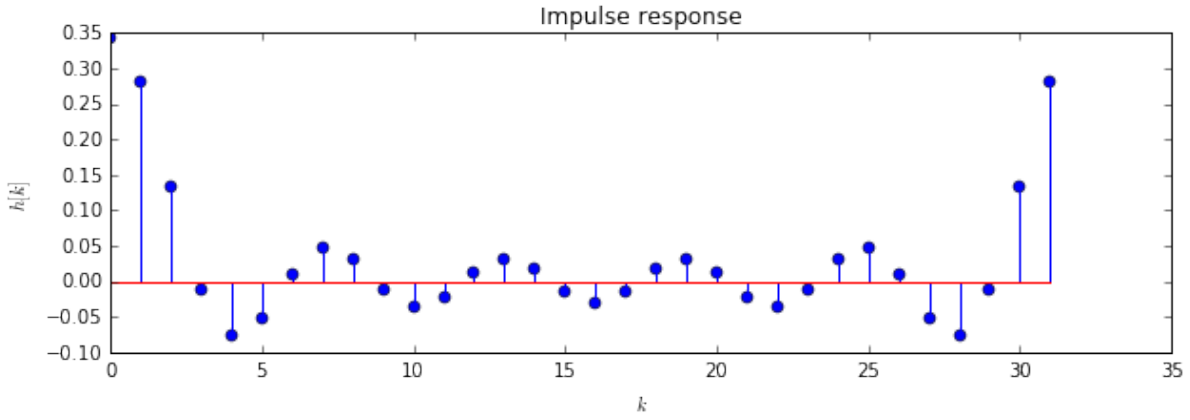
The design of an ideal low-pass filter using the frequency sampling method is considered. For $|\Omega| < \pi$ the transfer function of the ideal low-pass is given as

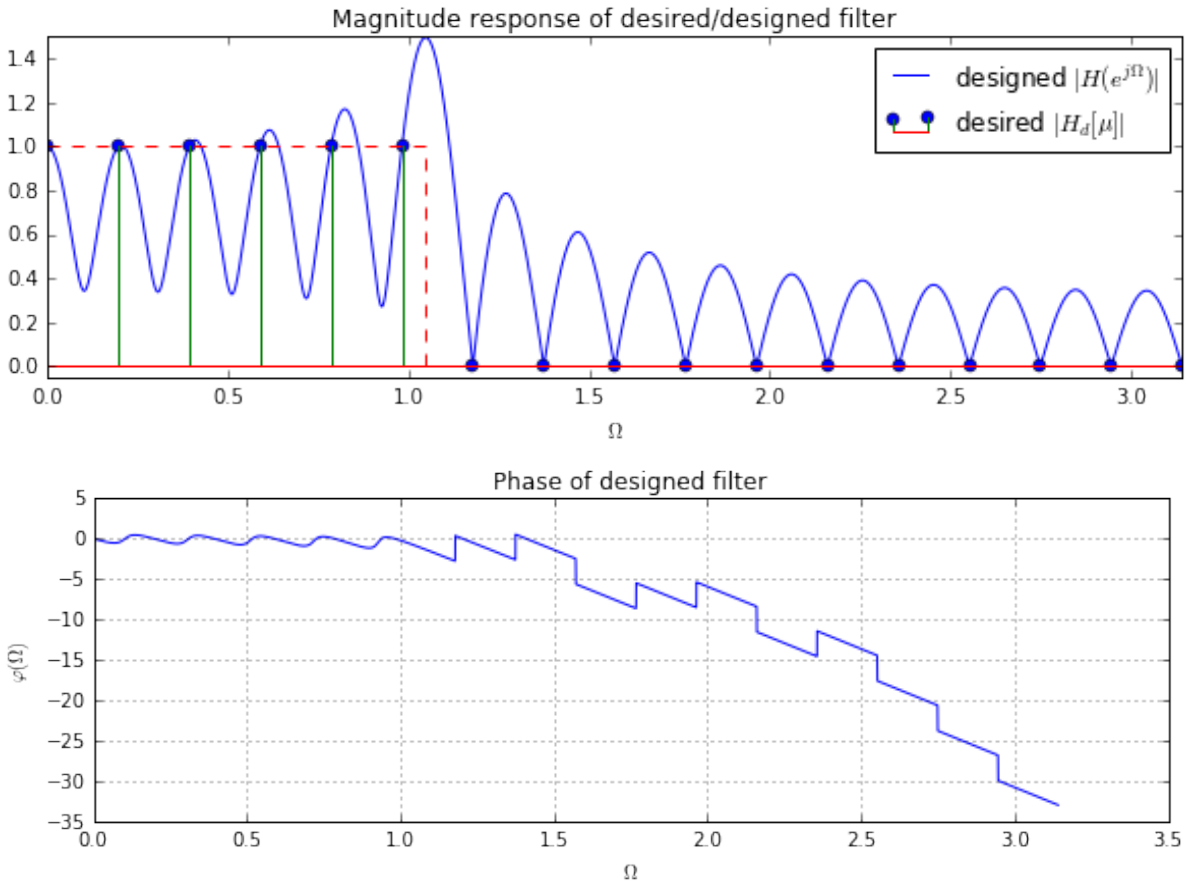
$$H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \begin{cases} 1 & \text{for } |\Omega| \leq \Omega_\mathrm{c} \\ 0 & \text{otherwise} \end{cases} \tag{8.18}$$

where Ω_c denotes its corner frequency. The desired transfer function $H_\mathrm{d}[\mu]$ for the frequency sampling method is derived by sampling $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$. Note that for sampling on the unit circle with $0 \leq \Omega < 2\pi$, the periodicity $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}(\Omega + n2\pi)})$ for $n \in \mathbb{Z}$ has to be considered.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig
```

```
N = 32  # length of filter
Omc = np.pi/3  # corner frequency of low-pass

# specify desired frequency response
Ommu = 2*np.pi/N*np.arange(N)
Hd = np.zeros(N)
Hd[Ommu <= Omc] = 1
Hd[Ommu >= (2*np.pi-Omc)] = 1

# compute impulse response of filter
h = np.fft.ifft(Hd)
h = np.real(h)  # due to round-off errors
# compute frequency response of filter
Om, H = sig.freqz(h, worN=8192)

# plot impulse response
plt.figure(figsize = (10,3))
plt.stem(h)
plt.title('Impulse response')
plt.xlabel(r'$k$')
plt.ylabel(r'$h[k]$')
# plot transfer functions
plt.figure(figsize = (10,3))
plt.plot(Om, np.abs(H), 'b-', label=r'designed $|H(e^{j \Omega})|$')
plt.stem(Ommu, np.abs(Hd), 'g', label=r'desired $|H_d[\mu]|$')
plt.plot([0, Omc, Omc], [1, 1, 0], 'r--')
plt.title('Magnitude response of desired/designed filter')
plt.xlabel(r'$\Omega$')
plt.legend()
plt.axis([0, np.pi, -0.05, 1.5])
# plot phase
plt.figure(figsize = (10,3))
plt.plot(Om, np.unwrap(np.angle(H)))
plt.title('Phase of designed filter')
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$\varphi(\Omega)$')
plt.grid()
```





Exercises

- What phase behavior does the designed filter have?
- Increase the length `N` of the filter. Does the attenuation in the stop-band improve?

The reason for the poor perfomance of the designed filter is the zero-phase of the desired transfer function which cannot be realized by a causal non-recursive system. This was *already discussed for the window method*. In comparison to the window method, the freqeuncy sampling method suffers from addtional time-domain aliasing due to the periodicy of the DFT. Again a linear-phase design is better in such situations.

8.2.2 Design of Linear-Phase Filters

The design of non-recursive FIR filters with a linear phase is often desired due to their constant group delay. As for the window method, the design of a digital filter with a generalized linear phase is considered in the following. For $|\Omega| < \pi$ its transfer function is given as

$$H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = A(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \cdot \mathrm{e}^{\,-\mathrm{j}\alpha\Omega + \mathrm{j}\beta} \tag{8.19}$$

where $A(\mathrm{e}^{\,\mathrm{j}\,\Omega}) \in \mathbb{R}$ denotes the amplitude of the filter, $-\alpha\,\Omega$ its linear phase and β a constant phase offset. The impulse response $h[k]$ of a linear-phase filter shows specific symmetries which have already been discussed for the *design of linear-phase filters using the window method*. For the resulting four types of linear-phase FIR filters, the properties of $A(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and the values of α and β have to be chosen accordingly for the formulation of $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ and $H_d[\mu]$, respectively. This is illustrated in the following for the design of a low-pass filter.

Example: Linear-phase approximation of an ideal low-pass

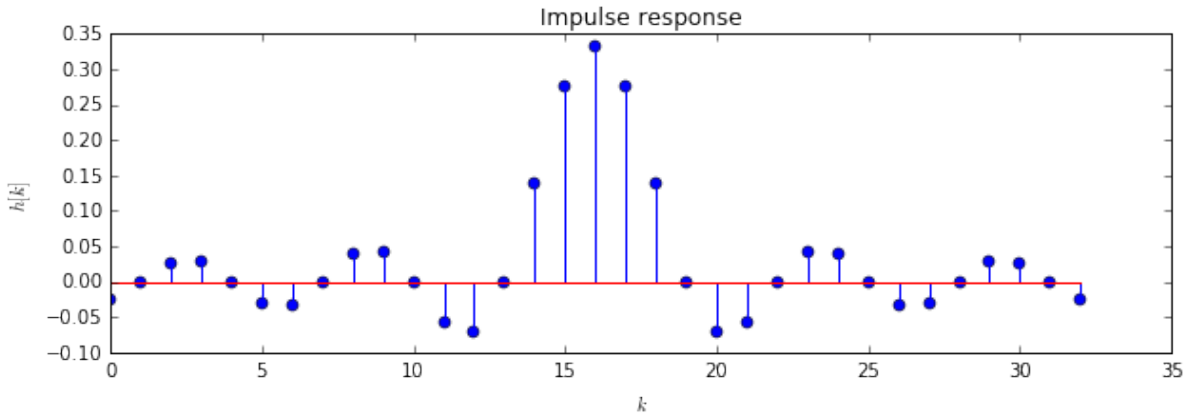
We aim at the approximation of an ideal low-pass as a linear-phase non-recursive FIR filter. For the sake of comparison with a similar *example for the window method*, we choose a type 1 filter with odd filter length N,

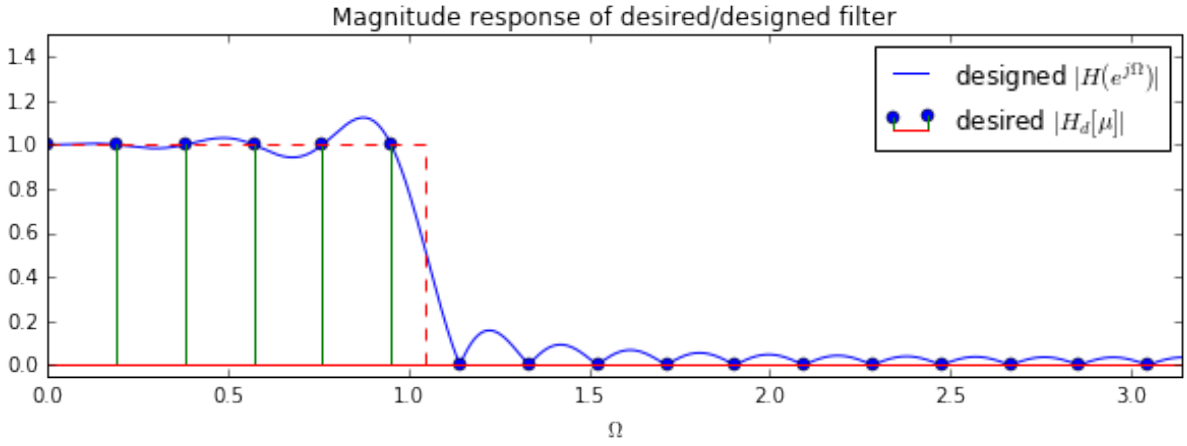
$\alpha = \frac{N-1}{2}$ and $\beta = 0$. The desired frequency response $H_\mathrm{d}[\mu]$ is given by sampling

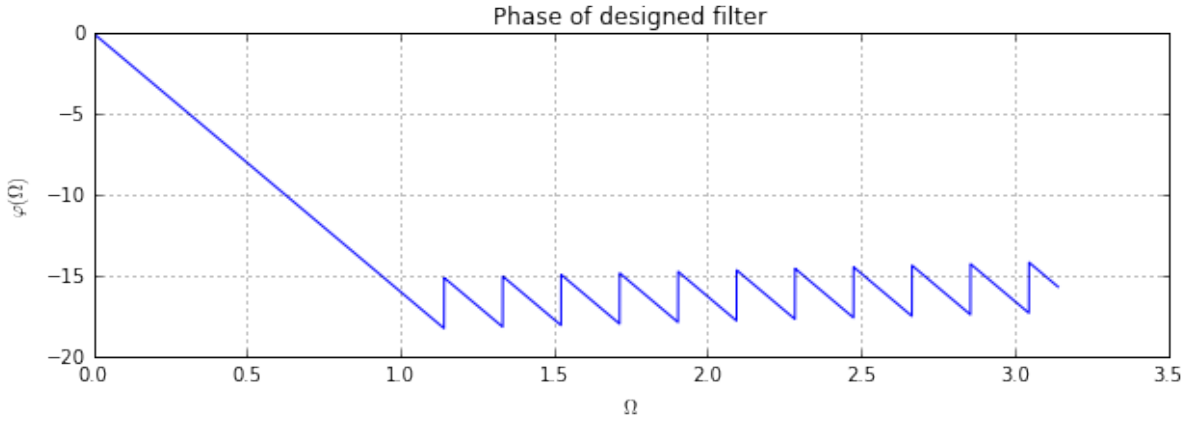
$$H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = \mathrm{e}^{\,-\mathrm{j}\,\frac{N-1}{2}\,\Omega} \cdot \begin{cases} 1 & \text{for } |\Omega| \leq \Omega_\mathrm{c} \\ 0 & \text{otherwise} \end{cases} \tag{8.20}$$

which is defined for $|\Omega| < \pi$. Note that for sampling on the unit circle with $0 \leq \Omega < 2\pi$, the periodicity $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}(\Omega + n 2\pi)})$ for $n \in \mathbb{Z}$ has to be considered.

```
In [2]: N = 33  # length of filter
        Omc = np.pi/3  # corner frequency of low-pass

        # specify desired frequency response
        Ommu = 2*np.pi/N*np.arange(N)
        Hd = np.zeros(N)
        Hd[Ommu <= Omc] = 1
        Hd[Ommu >= (2*np.pi-Omc)] = 1
        Hd = Hd * np.exp(-1j*Ommu*(N-1)/2)

        # compute impulse response of filter
        h = np.fft.ifft(Hd)
        h = np.real(h)  # due to round-off errors
        # compute frequency response of filter
        Om, H = sig.freqz(h, worN=8192)

        # plot impulse response
        plt.figure(figsize = (10,3))
        plt.stem(h)
        plt.title('Impulse response')
        plt.xlabel(r'$k$')
        plt.ylabel(r'$h[k]$')
        # plot frequency response
        plt.figure(figsize = (10,3))
        plt.plot(Om, np.abs(H), 'b-', label=r'designed $|H(e^{j \Omega})|$')
        plt.stem(Ommu, np.abs(Hd), 'g', label=r'desired $|H_d[\mu]|$')
        plt.plot([0, Omc, Omc], [1, 1, 0], 'r--')
        plt.title('Magnitude response of desired/designed filter')
        plt.xlabel(r'$\Omega$')
        plt.legend()
        plt.axis([0, np.pi, -0.05, 1.5])
        # plot phase
        plt.figure(figsize = (10,3))
        plt.plot(Om, np.unwrap(np.angle(H)))
        plt.title('Phase of designed filter')
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$\varphi(\Omega)$')
        plt.grid()
```







Exercises

- Does the designed filter have the desired linear phase?
- Increase the length `N` of the filter. What is different to the previous example?
- How could the method be modified to change the properties of the frequency response?

8.2.3 Comparison to Window Method

For a comparison of the frequency sampling to the window method it is assumed that the desired frequency response $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is given. For both methods, the coefficients $h[k]$ of an FIR approximation are computed as follows

1. Window Method

$$h_\mathrm{d}[k] = \mathcal{F}_*^{-1}\{H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})\} \tag{8.21}$$
$$h[k] = h_\mathrm{d}[k] \cdot w[k] \tag{8.22}$$

2. Frequency Sampling Method

$$H_\mathrm{d}[\mu] = H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\frac{2\pi}{N}\mu}) \tag{8.23}$$
$$h[k] = \mathrm{DFT}_N^{-1}\{H_\mathrm{d}[\mu]\} \tag{8.24}$$

For finite lengths N, the difference between both methods is related to the periodicity of the DFT. For a desired frequency response $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ which does not result in a FIR $h_\mathrm{d}[k]$ of length N, the inverse DFT in the frequency sampling method will suffer from time-domain aliasing. In the general case, filter coefficients computed by the window and frequency sampling method will hence differ.

However, for a rectangular window $w[k]$ and $N \to \infty$ both methods will become equivalent. This reasoning motivates an oversampled frequency sampling method, where $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ is sampled at $M \gg N$ points in order to derive an approximation of $h_\mathrm{d}[k]$ which is then windowed to the target length N. The method is beneficial in cases where a closed-form inverse DTFT of $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$, as required for the window method, cannot be found.

Example: Oversampled frequency sampling method

We consider the design of a linear-phase approximation of an ideal low-pass filter using the oversampled frequency sampling method. For the sake of comparison, the parameters have been chosen in accordance to a *similar example using the window method*. Using $H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ from the previous example in this section, the filter is computed by

1. (Over)-Sampling the desired response at M frequencies

$$H_\mathrm{d}[\mu] = H_\mathrm{d}(\mathrm{e}^{\,\mathrm{j}\,\frac{2\pi}{M}\mu}) \tag{8.25}$$

2. Inverse DFT of length M

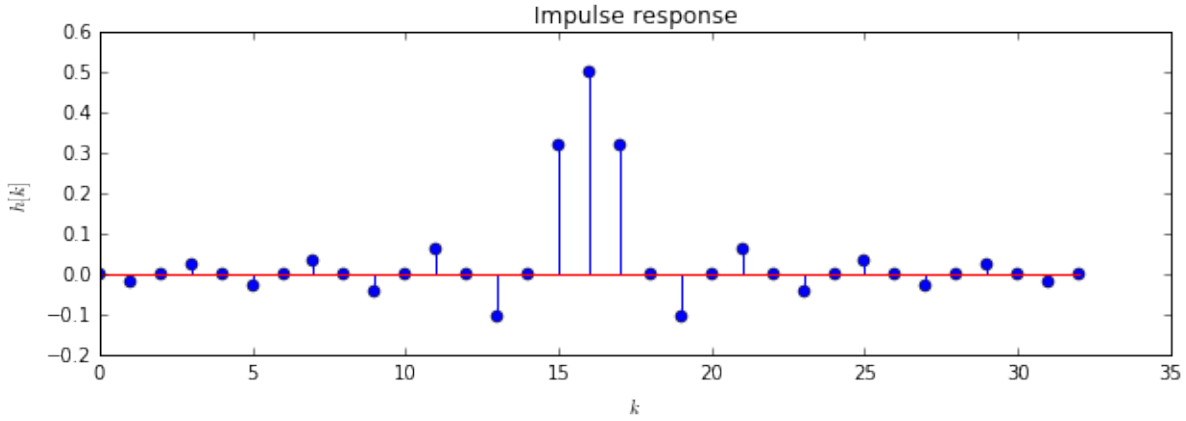
$$h[k] = \mathrm{DFT}_M^{-1}\{H_\mathrm{d}[\mu]\} \tag{8.26}$$

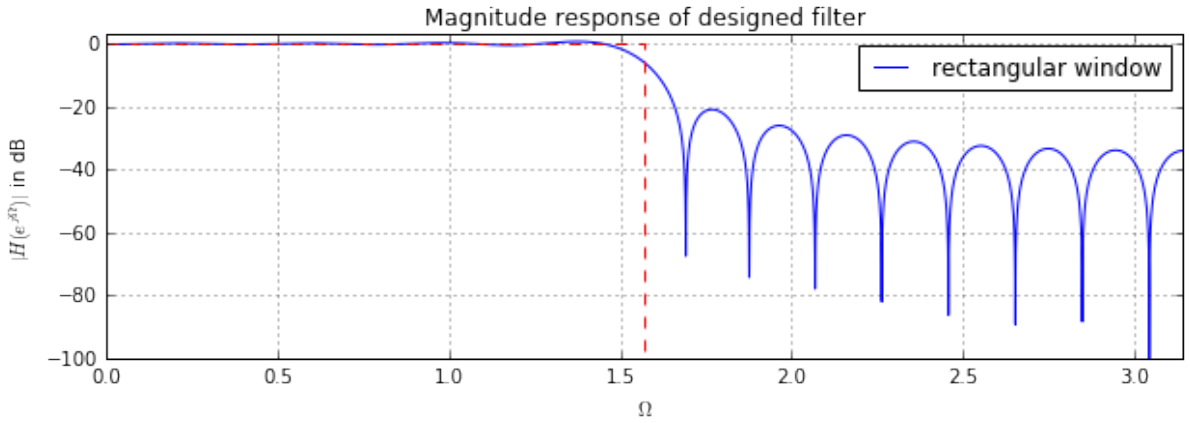
3. Windowing to desired length N

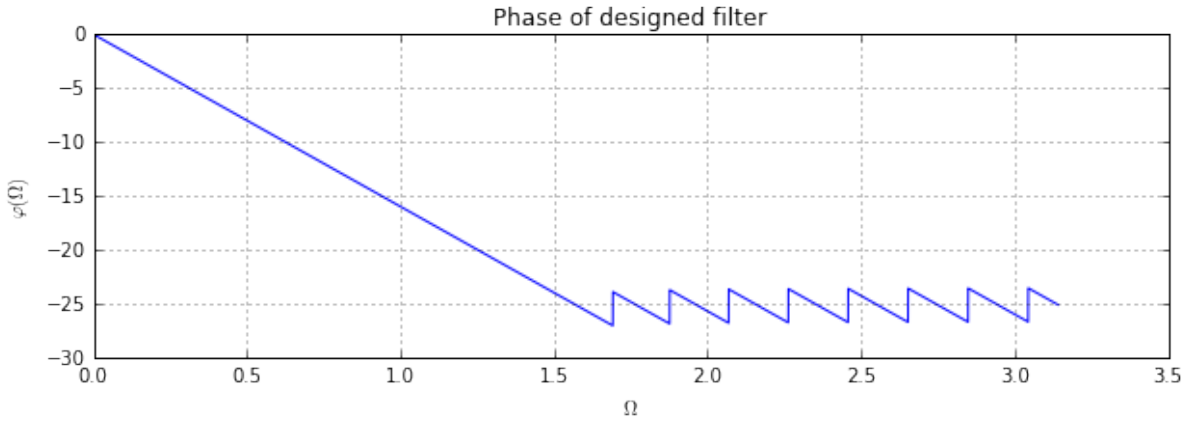
$$h[k] = h_\mathrm{d}[k] \cdot w[k] \tag{8.27}$$

```
In [3]: N = 33  # length of filter
        M = 8192  #
        Omc = np.pi/2  # corner frequency of low-pass

        # specify desired frequency response
        Ommu = 2*np.pi/M*np.arange(M)
        Hd = np.zeros(M)
        Hd[Ommu <= Omc] = 1
        Hd[Ommu >= (2*np.pi-Omc)] = 1
        Hd = Hd * np.exp(-1j*Ommu*(N-1)/2)

        # compute impulse response of filter
        h = np.fft.ifft(Hd)
        h = np.real(h)  # due to round-off errors
        h = h[0:N]  # rectangular window
        # compute frequency response of filter
        Om, H = sig.freqz(h, worN=8192)

        # plot impulse response
        plt.figure(figsize = (10,3))
        plt.stem(h)
        plt.title('Impulse response')
```

```
plt.xlabel(r'$k$')
plt.ylabel(r'$h[k]$')
# plot frequency response
plt.figure(figsize = (10,3))
plt.plot(Om, 20 * np.log10(abs(H)), label='rectangular window')
plt.plot([0, Omc, Omc], [0, 0, -100], 'r--')
plt.title('Magnitude response of designed filter')
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$|H(e^{j \Omega})|$ in dB')
plt.axis([0, np.pi, -100, 3])
plt.legend()
plt.grid()
# plot phase
plt.figure(figsize = (10,3))
plt.plot(Om, np.unwrap(np.angle(H)))
plt.title('Phase of designed filter')
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$\varphi(\Omega)$')
plt.grid()
```







Exercises

- Compare the designed filter and its properties to the *same design using the window method*
- Change the number of samples `M` used for sampling the desired response. What changes if you increase/decrease `M`?

8.3 Design of Recursive Filters by the Bilinear Transform

Various techniques have been developed to derive digital realizations of analog systems. For instance the impulse invariance method, the matched Z-transform and the bilinear transform. The following section introduces the bilinear transform and its application to the digital realization of analog systems and recursive filter design.

8.3.1 The Bilinear Transform

The bilinear transform is used to map the transfer function $H(s) = \mathcal{L}\{h(t)\}$ of a continous system to the transfer function $H(z) = \mathcal{Z}\{h[k]\}$ of a discrete system approximating the continuous system. The transform is designed such that if $H(s)$ is a rational function in s then $H(z)$ is a rational function in z^{-1}. The coefficients of the powers of z^{-1} are then the coefficients of the digital system.

Assuming ideal sampling with sampling interval T, the frequency variable s of the Laplace transformation is linked to the frequency variable z of the z-transform by

$$z = \mathrm{e}^{sT} \tag{8.28}$$

For sampled signals the resulting mapping from the s-plane into the z-plane is shown in the following illustration:

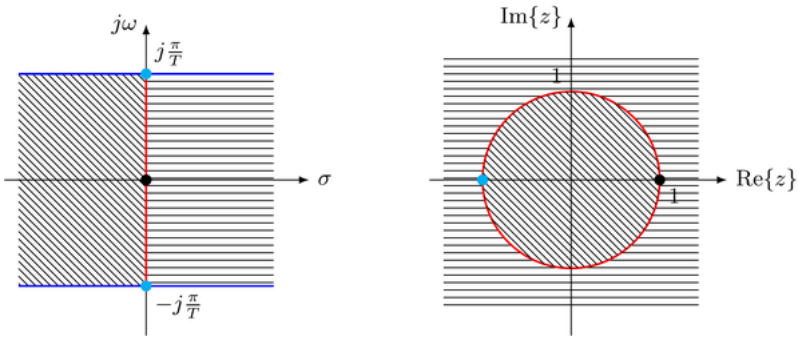


Fig. 8.1: Mapping of s-plane onto z-plane for sampled signals

The shading indicates how the different areas are mapped. The imaginary axis $s = \mathrm{j}\,\omega$ is mapped onto the unit circle $z = \mathrm{e}^{\,\mathrm{j}\,\Omega}$, representing the frequency response of the continuous and discrete system. The left half-plane of the s-plane is mapped into the unit circle of the z-plane.

For the desired mapping of $H(s)$ to $H(z)$ we need the inverse of above equation. It is given as

$$s = \frac{1}{T} \cdot \ln(z) \tag{8.29}$$

However, when introduced into a rational transfer function $H(s)$ this non-linear mapping would not result in the desired rational transfer function $H(z)$. In order to achieve the desired mapping, $\ln(z)$ is expanded into the power series

$$\ln(z) = 2\left(\frac{z-1}{z+1} + \frac{(z-1)^3}{3(z+1)^3} + \frac{(z-1)^5}{5(z+1)^5} + \ldots\right) \tag{8.30}$$

Using only the linear term as approximation of $\ln(z)$ yields the bilinear transform

$$s = \frac{2}{T} \cdot \frac{z-1}{z+1} \tag{8.31}$$

and its inverse

$$z = \frac{2+sT}{2-sT} \tag{8.32}$$

It worthwhile noting that this mapping rule is a special case of a conformal map.

Let's consider the mapping of the frequency response $H(\mathrm{j}\,\omega) = H(s)\big|_{s=\mathrm{j}\,\omega}$ of a continuous system to the frequency response $H_d(\mathrm{e}^{\,\mathrm{j}\,\omega T}) = H_d(z)\big|_{z=\mathrm{e}^{\,\mathrm{j}\,\Omega}}$ of a discrete system. Introducing the bilinear transform into the continuous system to yield its discrete counterpart results in

$$H_d(\mathrm{e}^{\,\mathrm{j}\,\Omega}) = H\left(\frac{2}{T} \cdot \frac{z-1}{z+1}\right)\Bigg|_{z=\mathrm{e}^{\,\mathrm{j}\,\Omega}} = H\Big(\mathrm{j}\,\underbrace{\frac{2}{T} \cdot \tan(\frac{\Omega}{2})}_{\omega}\Big) \tag{8.33}$$

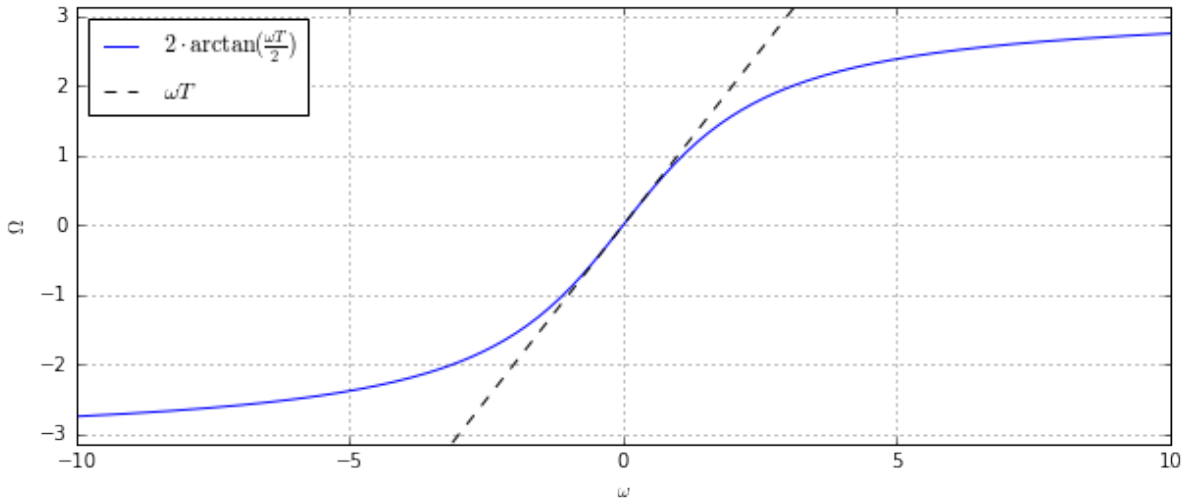
The imaginary axis $s = \mathrm{j}\,\omega$ of the s-plane is mapped onto the unit circle $\mathrm{e}^{\,\mathrm{j}\,\Omega}$ of the z-plane. Note, that for sampled signals the mapping between the continuous frequency axis ω and the frequency axis Ω of the discrete system is $\Omega = \omega T$. However, for the bilinear transform the mapping is non-linear

$$\omega = \frac{2}{T} \cdot \tan(\frac{\Omega}{2}) \tag{8.34}$$

$$\Omega = 2\arctan(\frac{\omega T}{2}) \tag{8.35}$$

In the following, this is illustrated for $T = 1$

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        om = np.linspace(-10, 10, 200)
        Om = 2*np.arctan(om*1/2)

        plt.figure(figsize=(10,4))
        plt.plot(om, Om, label=r'$2 \cdot \arctan(\frac{\omega T}{2})$')
        plt.plot(om, om, 'k--', label=r'$\omega T$')
        plt.xlabel(r'$\omega$')
        plt.ylabel(r'$\Omega$')
        plt.axis([-10, 10, -np.pi, np.pi])
        plt.legend(loc=2)
        plt.grid()
```



It is evident that the frequency axis deviates from the linear mapping $\Omega = \omega T$, especially for high frequencies. The frequency response of the digital filter $H_d(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ therefore deviates from the desired continuous frequency response $H(\mathrm{j}\,\omega)$. This is due to the first-order approximation of the mapping from the s-plane to the z-plane. The effect is known as *frequency warping*. It can be considered explicitly in the filter design stage, as shown in the examples.

Besides this drawback, the bilinear transform has a number of benefits:

- stability and minimum-phase of the continuous filter is preserved. This is due to mapping of the left half-space of the s-plane into the unit-circle of the z-plane.
- the order of the continuous filter is preserved. This is due to the linear mapping rule.
- no aliasing distortion as observed for the impulse invariance method

The application of the bilinear transform to the design of digital filters is discussed in the following.

8.3.2 Design of Digital Filter

We aim at designing a digital filter $H_d(z)$ that approximates a given continuous prototype $H(s)$ using the bilinear transform. For instance, the transfer function $H(s)$ may result from the analysis of an analog circuit or filter design technique. The transfer function $H_d(z)$ of the digital filter is then given by

$$H_d(z) = H(s)\Big|_{s=\frac{2}{T}\cdot\frac{z-1}{z+1}} \tag{8.36}$$

The coefficients of the digital filter are derived by representing the numerator and denominator of $H_d(z)$ as polynomials with respect to z^{-1}. For instance, for a continuous system of second order (second order section)

$$H(s) = \frac{\beta_0 + \beta_1 s + \beta_2 s^2}{\alpha_0 + \alpha_1 s + \alpha_2 s^2} \tag{8.37}$$

the bilinear transform results in

$$H_d(z) = \frac{(\beta_2 K^2 - \beta_1 K + \beta_0)\, z^{-2} + (2\beta_0 - 2\beta_2 K^2)\, z^{-1} + (\beta_2 K^2 + \beta_1 K + \beta_0)}{(\alpha_2 K^2 - \alpha_1 K + \alpha_0)\, z^{-2} + (2\alpha_0 - 2\alpha_2 K^2)\, z^{-1} + (\alpha_2 K^2 + \alpha_1 K + \alpha_0)} \tag{8.38}$$

where $K = \frac{2}{T}$.

As outlined in the previous section, the frequency response of the digital filter $H_d(\mathrm{e}^{\,\mathrm{j}\,\Omega})$ will differ for high frequencies from the desired analog frequency response $H(\mathrm{j}\,\omega)$. For the design of a digital filter from an analog prototype, this can be coped for by replacing corner frequencies with

$$\omega_{\mathrm{cw}} = \frac{2}{T} \cdot \tan\left(\frac{\omega_{\mathrm{c}} T}{2}\right) \tag{8.39}$$

where ω_{cw} denotes a *warped* corner frequency ω_{c}. This techniques is known as *pre-warping*.

8.3.3 Examples

The following two examples illustrate the digital realization of an analog system and the design of a recursive filter, respectively.

Digital realization of an analog system

A second-order lowpass filter can be realized by the following passive circuit

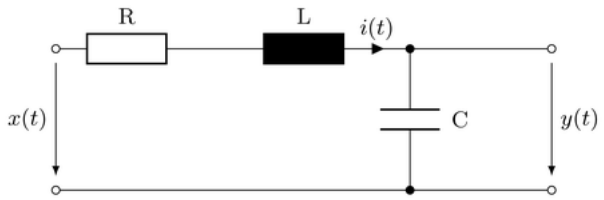


Fig. 8.2: Analog second-order lowpass

where $x(t)$ denotes the input signal and $y(t)$ the output signal. Analysis of the circuit reveals its transfer function as

$$H(s) = \frac{Y(s)}{X(s)} = \frac{1}{LC\,s^2 + RC\,s + 1} \tag{8.40}$$

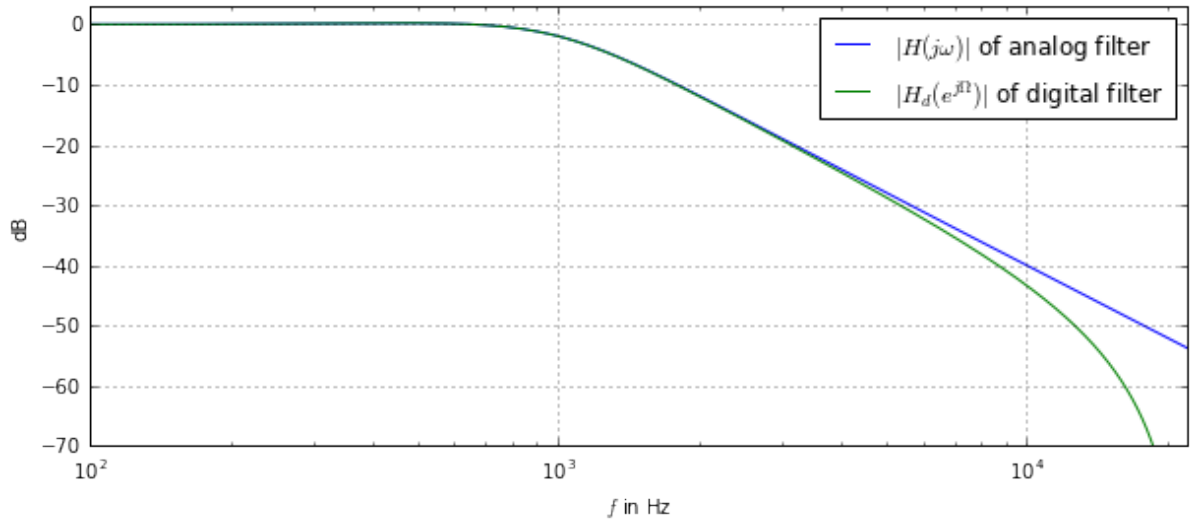
Introducing this into the bilinear transform of a second order section (SOS) given in the previous section yields

$$H_d(z) = \frac{T^2\,z^{-2} + 2T^2\,z^{-1} + T^2}{(4LC - 2TRC + T^2)\,z^{-2} + (-8LC + 2T^2)\,z^{-1} + (4LC - 2TRC + T^2)} \tag{8.41}$$

In the following, the frequency response of the analog filter and its digital realization is compared numerically.

```
In [2]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig


        fs = 44100  # sampling frequency
        fc = 1000  # corner frequency of the lowpass

        # coefficients of analog lowpass filter
        Qinf = 0.8
        sinf = 2*np.pi*fc
        C = 1e-6
        L = 1/(sinf**2*C)
        R = sinf*L/Qinf

        B = [0, 0, 1]
        A = [L*C, R*C, 1]

        # cofficients of digital filter
        T = 1/fs
        b = [T**2, 2*T**2, T**2]
        a = [(4*L*C+2*T*R*C+T**2), (-8*L*C+2*T**2), (4*L*C-2*T*R*C+T**2)]
```

```
# compute frequency responses
Om, Hd = sig.freqz(b, a, worN=1024)
tmp, H = sig.freqs(B, A, worN=fs*Om)

# plot results
f = Om*fs/(2*np.pi)
plt.figure(figsize = (10, 4))
plt.semilogx(f, 20*np.log10(np.abs(H)), label=r'$|H(j \omega)|$ of analog filter
plt.semilogx(f, 20*np.log10(np.abs(Hd)), label=r'$|H_d(e^{j \Omega})|$ of digita
plt.xlabel(r'$f$ in Hz')
plt.ylabel(r'dB')
plt.axis([100, fs/2, -70, 3])
plt.legend(loc=1)
plt.grid()
```



Exercise

- Increase the corner frequency `fc` of the analog filter. What effect does this have on the deviations between the analog filter and its digital representation?

Design of digital filters from analog prototypes

The design of analog filters is a topic with a long lasting history. Consequently, many analog filter designs are known. For instance

- Butterworth filters
- Chebyshev_filters
- Cauer filters
- Bessel filters

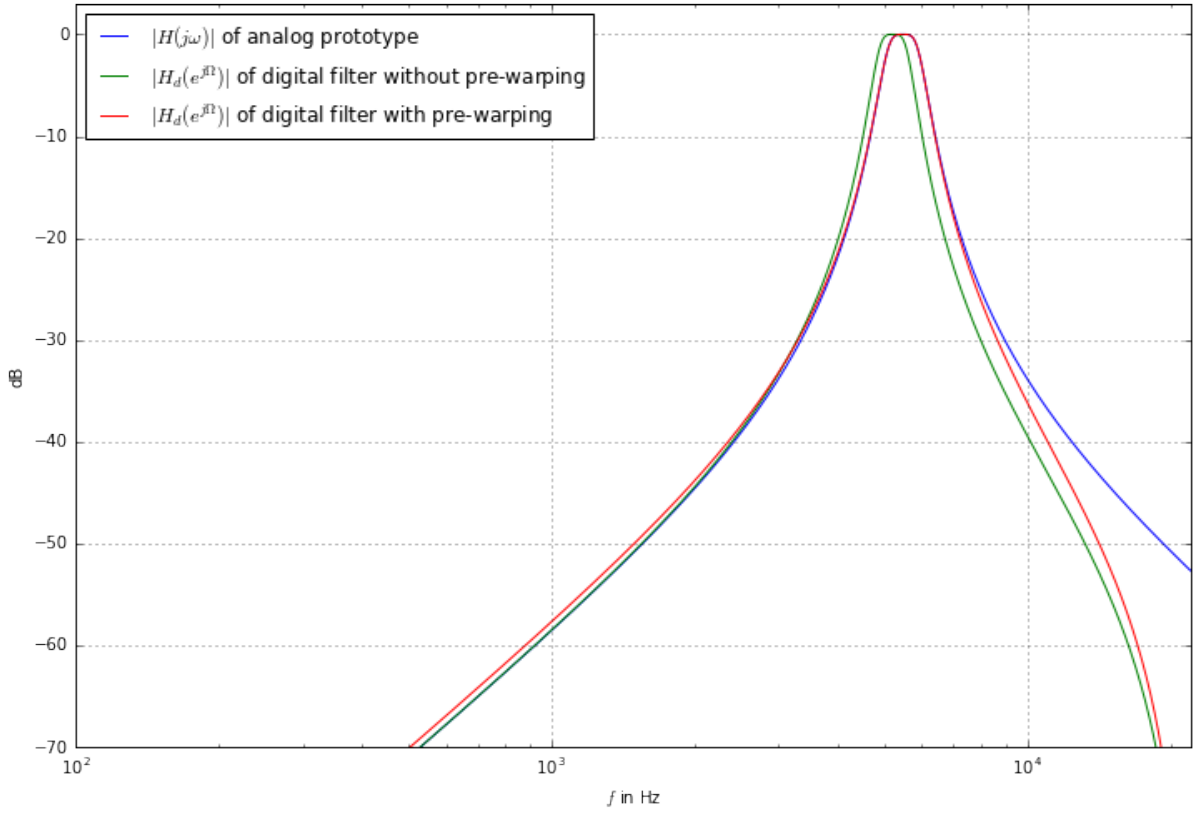
The properties of these designs are well documented and therefore digital realizations are of interest. These can be achieved by the bilinear transform. In Python, the `'scipy.signal <http://docs.scipy.org/doc/scipy/reference/signal.html>'__` package provides implementations of various analog filter design techniques, as well as an implementation of the bilinear transform.

The design of a Butterworth bandpass using pre-warping is illustrated in the following.

```
In [3]: omc = 2*np.pi*np.array([5000, 6000])  # corner frequencies of bandpass
        N = 2  # order of filter

        # pre-warping of corner frequencies
        omcp = 2*fs*np.tan(omc/(2*fs))
```

```
# design of analog filters with and without pre-warping
B, A = sig.butter(N, omc, btype='bandpass', analog=True)
Bp, Ap = sig.butter(N, omcp, btype='bandpass', analog=True)

# bilinear transform of analog prototypes
b, a = sig.bilinear(B, A, fs)
bp,ap = sig.bilinear(Bp, Ap, fs)

# compute frequency responses
Om, Hdp = sig.freqz(bp, ap, worN=1024)
Om, Hd = sig.freqz(b, a, worN=1024)
tmp, H = sig.freqs(B, A, worN=fs*Om)

# plot results
f = Om*fs/(2*np.pi)
plt.figure(figsize = (12, 8))
plt.semilogx(f, 20*np.log10(np.abs(H)), label=r'$|H(j \omega)|$ of analog protot
plt.semilogx(f, 20*np.log10(np.abs(Hd)), label=r'$|H_d(e^{j \Omega})|$ of digita
plt.semilogx(f, 20*np.log10(np.abs(Hdp)), label=r'$|H_d(e^{j \Omega})|$ of digit
plt.xlabel(r'$f$ in Hz')
plt.ylabel(r'dB')
plt.axis([100, fs/2, -70, 3])
plt.legend(loc=2)
plt.grid()
```

```
/opt/local/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
/opt/local/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
```



Exercise

- What is improved by pre-warping?
- Change the corner frequencies `omc` of the analog prototype and examine the deviations from the analog

prototype. When is pre-warping reasonable and when not?

8.4 Example: Non-Recursive versus Recursive Filter

In the following example, the characteristics and computational complexity of a non-recursive and a recursive filter are compared for a particular design. Quantization is not considered. In order to design the filters we need to specify the requirements. This is typically done by a *tolerance scheme*. The scheme states the desired frequency response and allowed deviations. This is explained at an example.

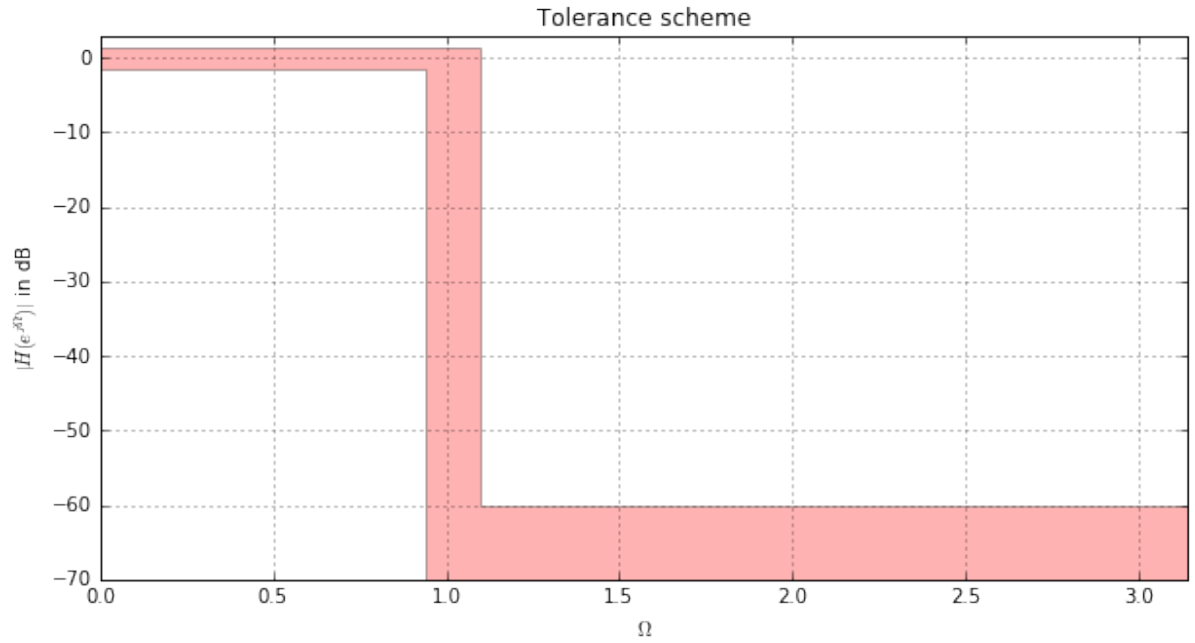
We aim at the design of a low-pass filter with

1. unit amplitude with an allowable symmetric deviation of δ_p for $|\Omega| < \Omega_\mathrm{p}$
2. an attenuation of a_s for $|\Omega| > \Omega_\mathrm{s}$

where the indices p and s denote the pass- and stop-band, respectively. The region between the pass-band Ω_p and the stop-band Ω_s is known as *transition-band*. The phase of the filter is not specified.

The resulting tolerance scheme is illustrated for the design parameters $\Omega_\mathrm{p} = \frac{\pi}{3}$, $\Omega_\mathrm{s} = \frac{\pi}{3} + 0.05$, $\delta_\mathrm{p} = 1.5$ dB and $a_\mathrm{s} = -60$ dB.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.patches as mpatches
        import scipy.signal as sig

        def plot_tolerance_scheme(Omp, Oms, d_p, a_s):
            Omp = Omp * np.pi
            Oms = Oms * np.pi

            p = [[0, -d_p], [Omp, -d_p], [Omp, -300], [np.pi, -300], [np.pi, a_s], [Oms,
            polygon = mpatches.Polygon(p, closed=True, facecolor='r', alpha=0.3)
            plt.gca().add_patch(polygon)


        Omp = .3  # normalized corner frequency of pass-band
        Oms = .3 + 0.05  # normalized corner frequency of stop-band
        d_p = 1.5  # one-sided pass-band ripple in dB
        a_s = -60  # stop-band attenuation in dB

        plt.figure(figsize = (10, 5))
        plot_tolerance_scheme(Omp, Oms, d_p, a_s)
        plt.title('Tolerance scheme')
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$|H(e^{j \Omega})|$ in dB')
        plt.axis([0, np.pi, -70, 3])
        plt.grid();
```



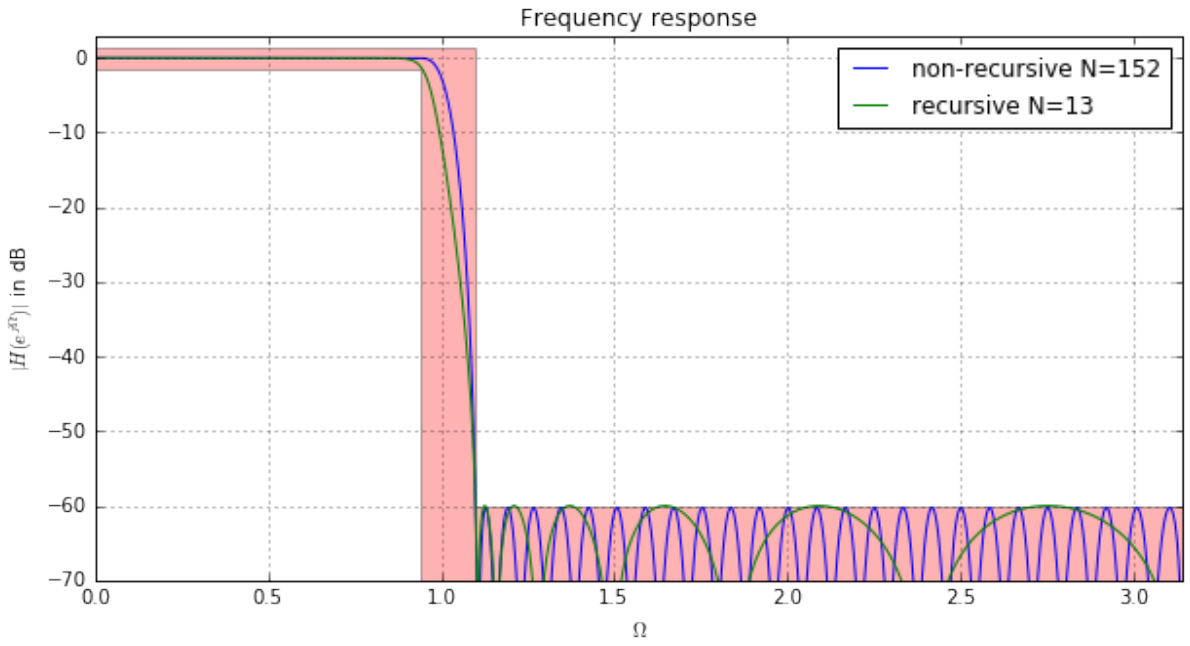
Exercise

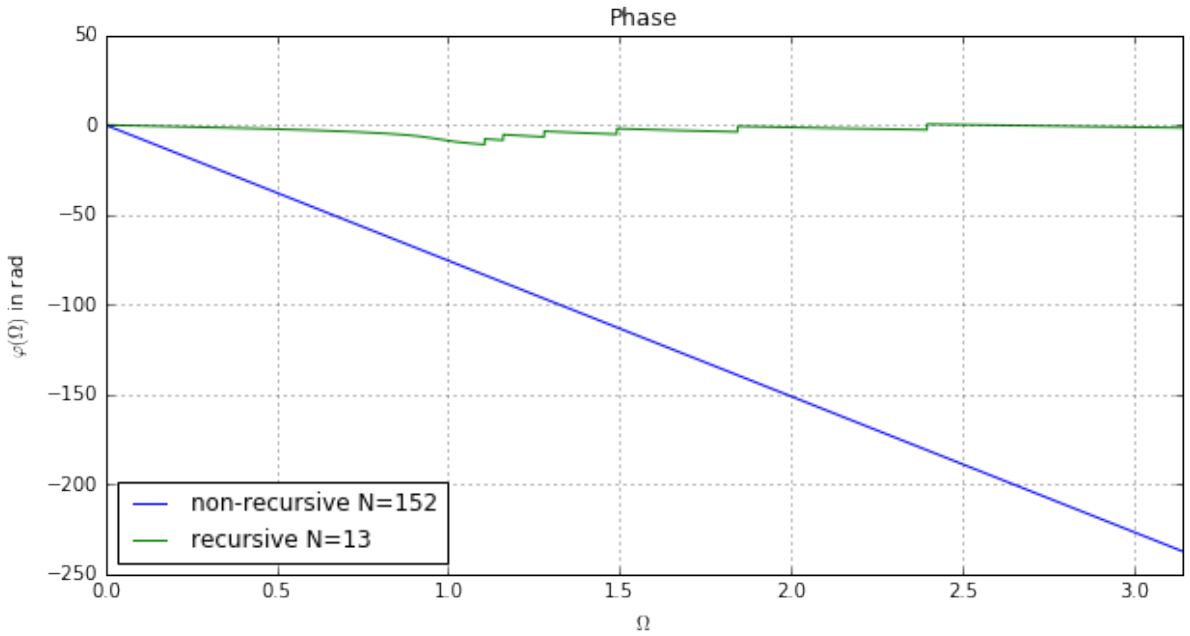
- What corner frequencies f_p and f_s result for a sampling frequency of $f_\mathrm{s} = 48$ kHz?

The comparison of non-recursive and recursive filters depends heavily on the chosen filter design algorithm. For the design of the non-recursive filter a technique is used which bases on numerical optimization of the filter coefficients with respect to the desired response. The Remez algorithm, as implemented in `scipy.signal.remez`, is used for this purpose. The parameters for the algorithm are the corner frequencies of the pass- and stop-band, as well as the desired attenuation in the stop-band. For the recursive filter, a Chebyshev type II design is used. Here the parameters are the corner frequency and attenuation of the stop-band. The order of both filters has been chosen manually to fit the given tolerance scheme.

```
In [2]: N = 152  # length of non-recursive filter
        M = 13  # order of recursive filter

        # design of non-recursive filter
        h = sig.remez(N, [0, Omp/2, Oms/2, 1/2], [1, 10**((a_s-5)/20)], weight=[1, 1])

        # design of recursive filter
        b, a = sig.cheby2(M, -a_s, Oms)

        # compute frequency response of filter
        Om, Hn = sig.freqz(h, worN=8192)
        Om, Hr = sig.freqz(b, a, worN=8192)

        # plot frequency response
        plt.figure(figsize = (10,5))
        plt.plot(Om, 20*np.log10(np.abs(Hn)), 'b-', label=r'non-recursive N=%d'%N)
        plt.plot(Om, 20*np.log10(np.abs(Hr)), 'g-', label=r'recursive N=%d'%M)
        plot_tolerance_scheme(Omp, Oms, d_p, a_s)
        plt.title('Frequency response')
        plt.xlabel(r'$\Omega$')
        plt.ylabel(r'$|H(e^{j \Omega})|$ in dB')
        plt.legend()
        plt.axis([0, np.pi, -70, 3])
        plt.grid()
        # plot phase
        plt.figure(figsize = (10,5))
```

```
plt.plot(Om, np.unwrap(np.angle(Hn)), label=r'non-recursive N=%d'%N)
plt.plot(Om, np.unwrap(np.angle(Hr)), label=r'recursive N=%d'%M)
plt.title('Phase')
plt.xlabel(r'$\Omega$')
plt.ylabel(r'$\varphi(\Omega)$ in rad')
plt.legend(loc=3)
plt.xlim([0, np.pi])
plt.grid()
```



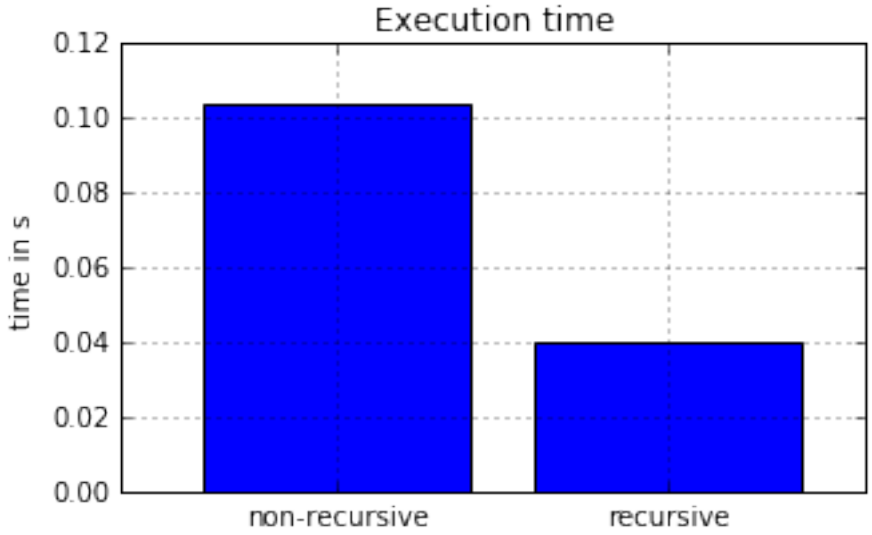


Exercises

- How do both designs differ in terms of their magnitude and phase responses?
- Calculate the number of multiplications and additions required to realize the non-recursive filter
- Calculate the number of multiplications and additions required to realize the recursive filter in *transposed direct form II*
- Decrease the corner frequencies and adapt the order of the filters to match the tolerance scheme

In order to evaluate the computational complexity of both filters, the execution time is measured when filtering a signal $x[k]$ of length $L = 10^5$ samples. The non-recursive filter is realized by direct convolution, the recursive filter in transposed direct form II using the respective Python functions.

```
In [3]: import timeit

        reps = 20  # number of repetitions for timeit

        # setup environment for timeit
        tsetup = 'import numpy as np; import scipy.signal as sig; from __main__ import h
        # non-recursive filter
        tn = timeit.timeit('np.convolve(x, h, "full")', setup=tsetup, number=reps)
        # recursive filter
        tr = timeit.timeit('sig.lfilter(b, a, x)' , setup=tsetup, number=reps)

        # show the results
        plt.figure(figsize = (5, 3))
        plt.bar(1, tn)
        plt.bar(2, tr)
        plt.title('Execution time')
        plt.xticks([1.4, 2.4], ('non-recursive', 'recursive'))
        plt.ylabel('time in s')
        plt.xlim([.75, 3])
        plt.grid()
```



Exercises

- Do the execution times correspond with the number of algorithmic operations calculated in the previous exercise?
- Estimate the computational load for the filtering of a signal with a sampling rate of 48 kHz
- How could the execution time of the non-recursive filter be decreased?
- Finally, would you prefer the non-recursive or the recursive design for a practical implementation? Consider the numerical complexity, as well as numerical aspects in your decision.

CHAPTER 9

Getting Started

The jupyter notebooks are available

- Online as static web pages
- Online for interactive use
- Local (offline) use on your computer

For local use on your computer, e.g. for the exercises, you have to download the notebooks from Github. Use Git to download the files and then start the Juyter notebook:

```
git clone https://github.com/spatialaudio/digital-signal-processing-lecture.git
cd digital-signal-processing-lecture
jupyter-notebook
```

This will open a new view in your web browser with a list of notebooks. Click on index.ipynb (or any of the other available notebooks). Alternatively, you can also download individual notebook files (with the extension `.ipynb`) and open them in Jupyter. Note that some notebooks make use of additional files (audio files etc.) which you'll then also have to download manually.

CHAPTER 10

Literature

The lecture notes base on the following literature:

- Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing*, Pearson Prentice Hall, 2010.
- John G. Proakis and Dimitris K. Manolakis, *Digital Signal Processing*, Pearson, 2006.
- Petre Stoica and Randolph Moses, *Spectral Analysis of Signals*, Prentice Hall, 2005.
- Udo Zölzer, *Digitale Audiosignalverarbeitung*, Teubner, 2005.
- Peter Vary, Ulrich Heute and Wolfgang Hess, *Digitale Sprachsignalverarbeitung*, Teubner, 1998.
- Bernd Girod, Rudolf Rabenstein, Alexander Stenger, *Einführung in die Systemtheorie*, B.G. Teubner Verlag, 2007.

CHAPTER 11

Contributors

- Sascha Spors (Author)
- Vera Erbes (Proofreading)
- Matthias Geier (Technical advice)
- Frank Schultz (Proofreading)