Lecture 6. **FIR and IIR filters**

Outline / overview

- **Section 1.** Signal smoothing
- **Section 2.** Amplitude spectra
- **Section 3.** Time and frequency domains
- **Section 4.** Filter design
- **Section 5.** Frequency response
- **Section 6.** Padding

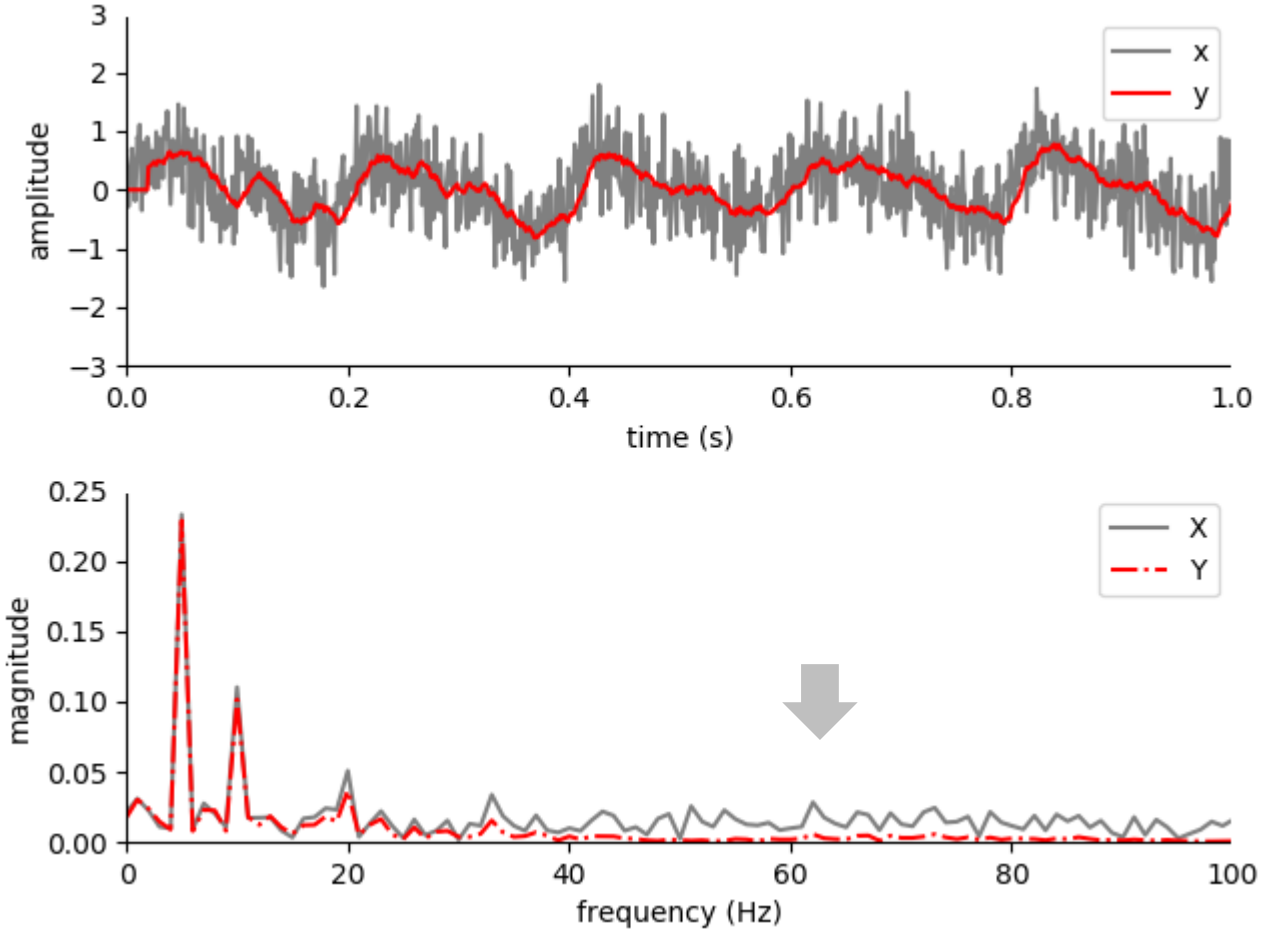
Section 1. Signal smoothing

Signal smoothing (1/4)

What is smoothing?

```
# signal
x = 0.5 * np.sin(2 * np.pi * 5 * t)  + \
    0.2 * np.sin(2 * np.pi * 10 * t) + \
    0.1 * np.sin(2 * np.pi * 20 * t) + \
    0.5 * np.random.randn(N)
X = np.abs(fft(x)) / N

# smoothing, M samples
M = 20
y = do_smoothing(x, M)
Y = np.abs(fft(y)) / N
```



See, “L06_smoothing.py”

Signal smoothing (2/4)

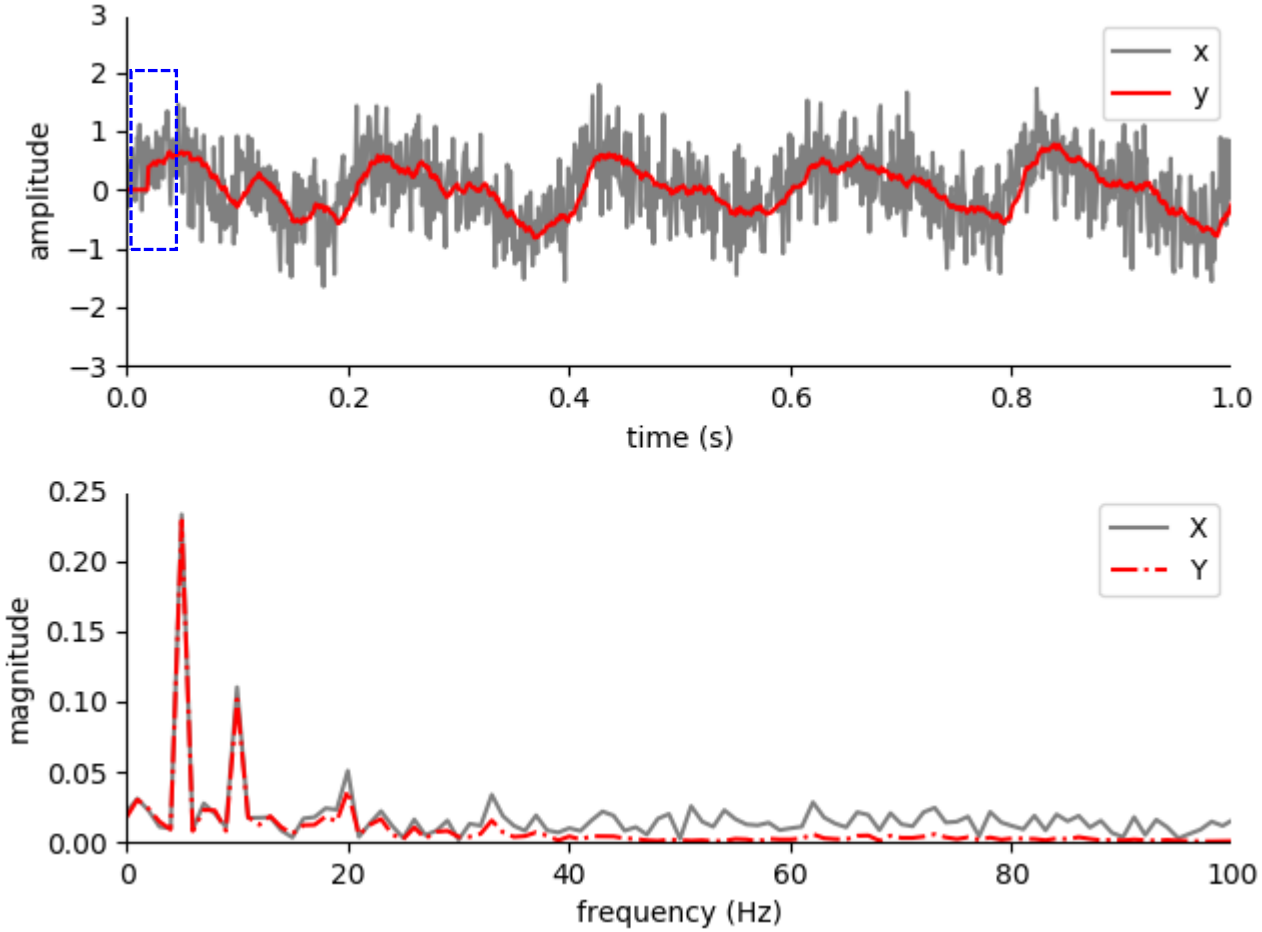
How does smoothing work?

```
def do_smoothing(x, M):

  N = len(x)
  y = np.zeros(N)

  # average
  for i in range(0, (N-M)):
    y[i+M] = np.sum(x[i:(i+M)]) / M

  return y
```



See, “L06_smoothing.py”

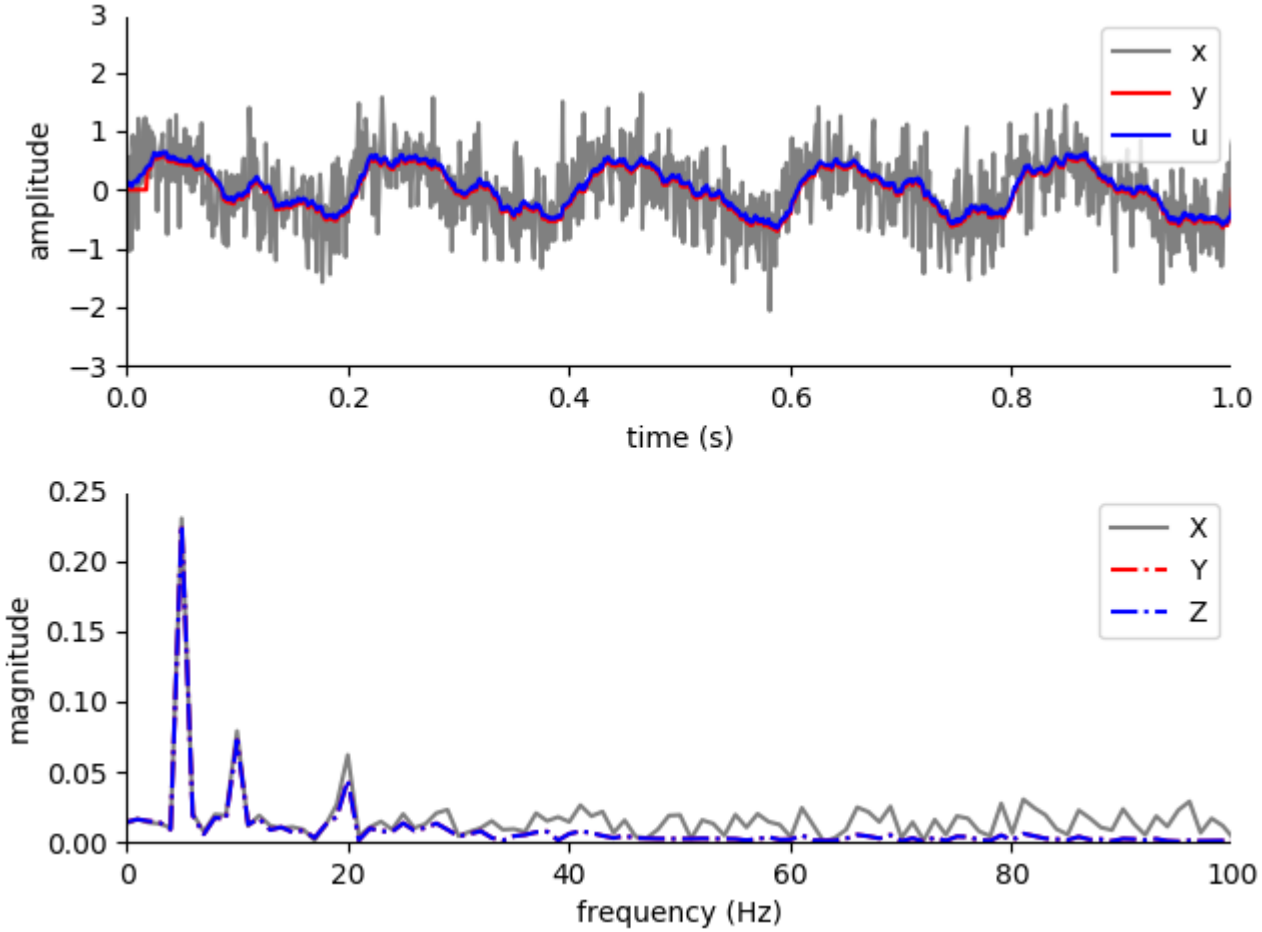
Signal smoothing (3/4)

Averaging is equivalent to convolution with square window.

```
# signal
x = 0.5 * np.sin(2 * np.pi * 5 * t)  + \
    0.2 * np.sin(2 * np.pi * 10 * t) + \
    0.1 * np.sin(2 * np.pi * 20 * t) + \
    0.5 * np.random.randn(N)
X = np.abs(fft(x)) / N

# smoothing, M samples
M = 20
y = do_smoothing(x, M)
Y = np.abs(fft(y)) / N

# convolution
w = np.ones(M) / M
u = do_convolution(x, w)
u = u[0:N]
U = np.abs(fft(y))
```



See, “L06_convolution_square_window.py”

Signal smoothing (3/4)

How does convolution work?

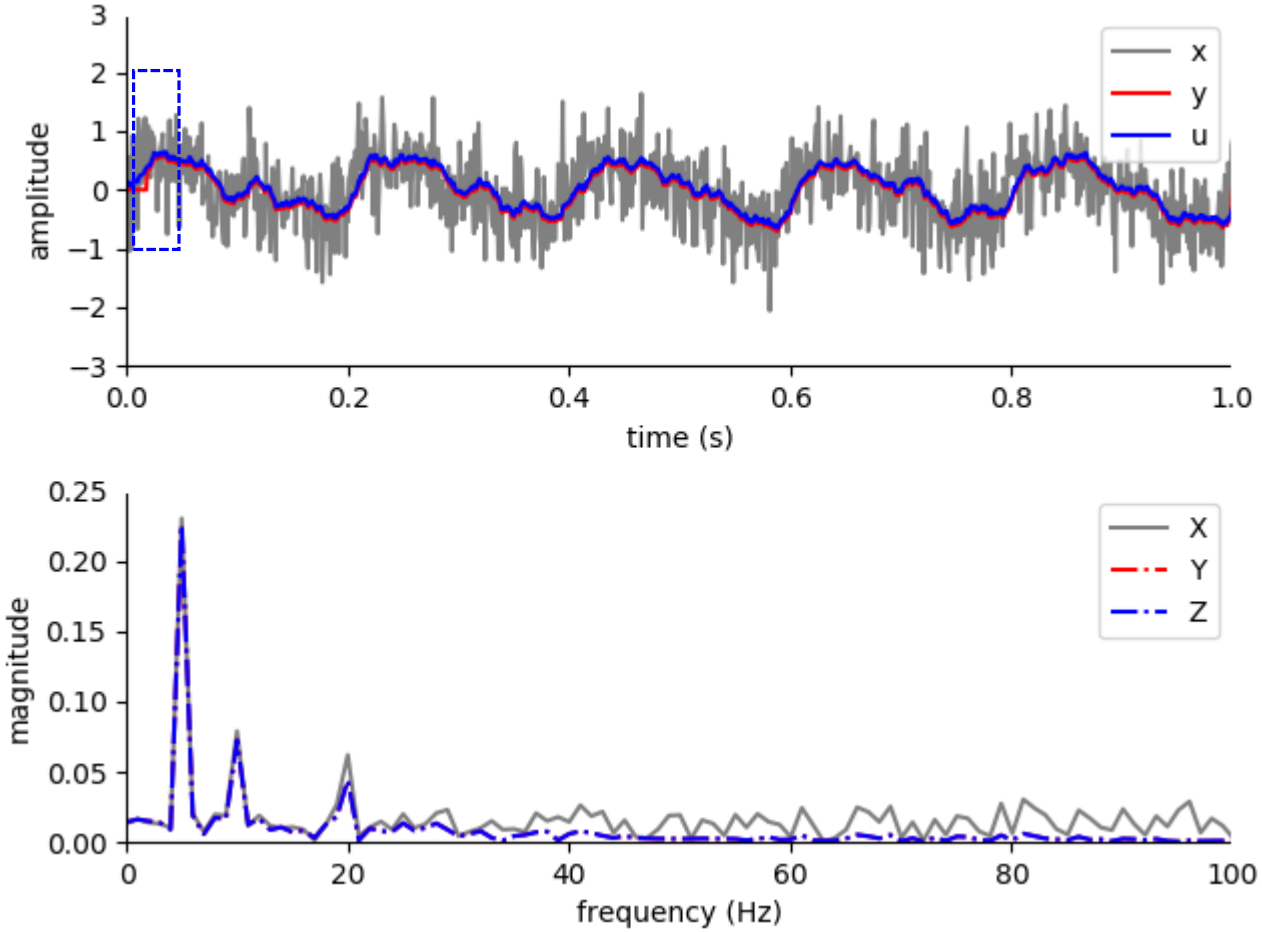
```
def do_convolution(x, w):

  # init
  N = len(x)
  M = len(w)

  # add zeros
  x = np.concatenate((np.zeros(M-1), x,
                      np.zeros(M-1)))
  u = np.zeros(N+M-1)

  # convolution
  for i in range(0, (N+M-1)):
    u[i] = np.sum(x[i:(i+M)] * w[::-1])

  return y
```



See, “L06_convolution_square_window.py”

Section 2. **Amplitude spectra**

Spectra (1/2)

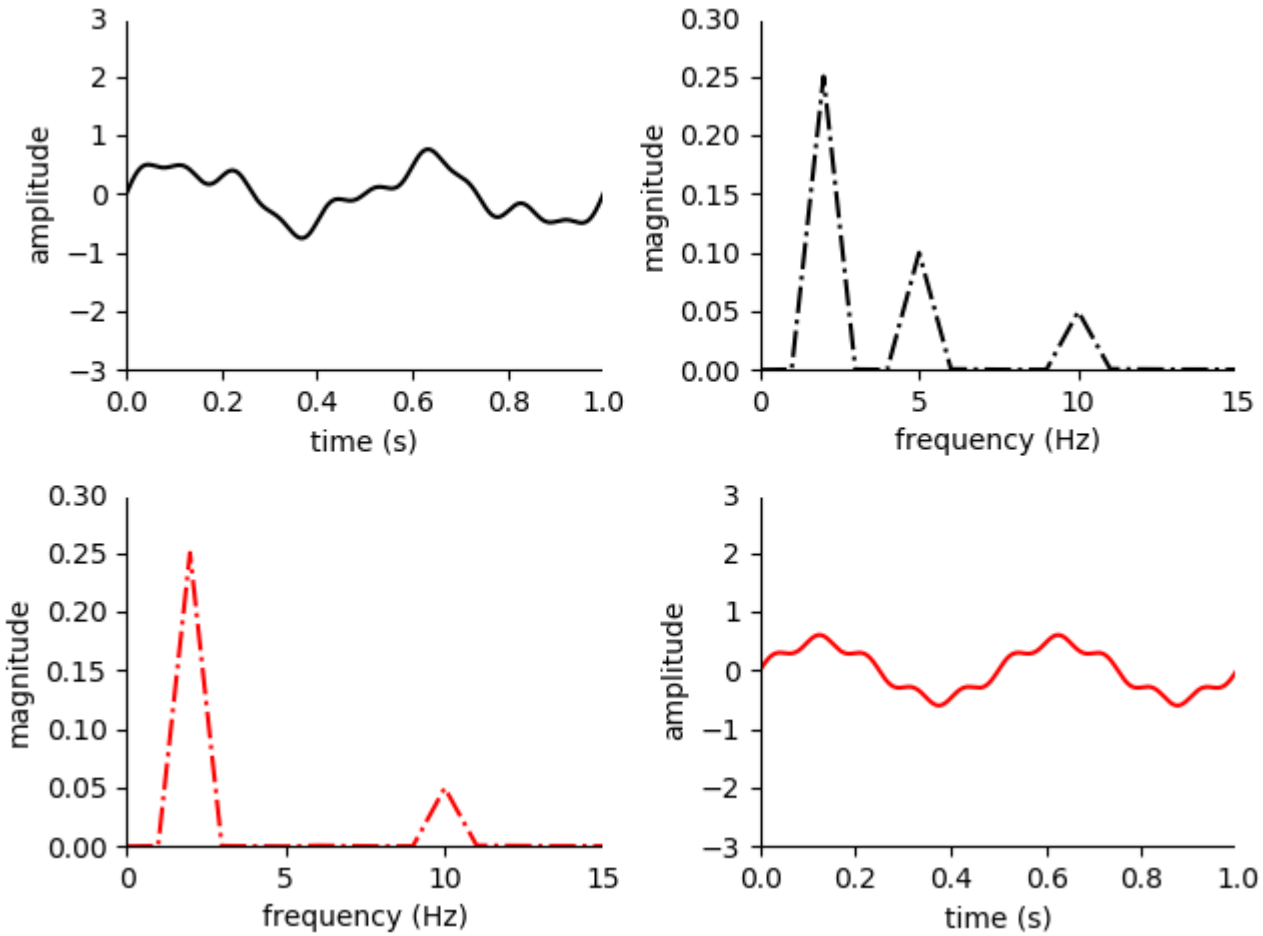
Could we supress certain frequencies in the signal?

```
# signal
x = 0.5 * np.sin(2 * np.pi * 2 * t)  + \
    0.2 * np.sin(2 * np.pi * 5 * t) + \
    0.1 * np.sin(2 * np.pi * 10 * t)
X = np.abs(fft(x)) / N


# fourier transform
fx = fft(x)

# remove 5 Hz
f0 = 5
fx[[f0, -f0]] = 0 # [f0, nFFT-f0]

# inverse fourier transform
y = np.real(ifft(fx))
Y = np.abs(fft(y)) / N
```



See, “L06_spectra_1.py”

Spectra (2/2)

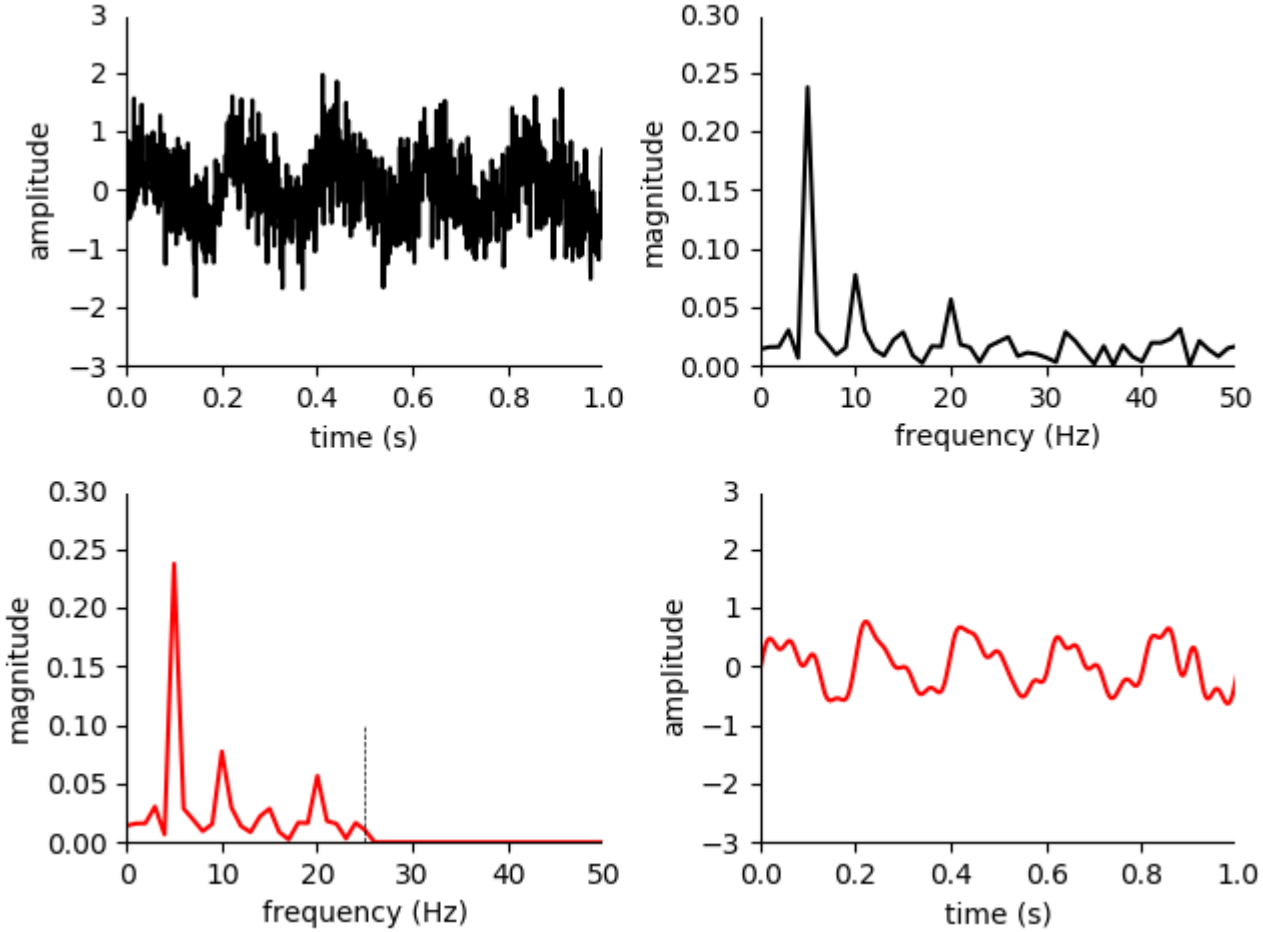
Could we supress range of frequencies in the signal?

```
# signal
x = 0.5 * np.sin(2 * np.pi * 5 * t)  + \
    0.2 * np.sin(2 * np.pi * 10 * t) + \
    0.1 * np.sin(2 * np.pi * 20 * t) + \
    0.5 * np.random.randn(N)
X = np.abs(fft(x)) / N

# fourier transform
fx = fft(x)

# remove above 25 Hz
f0 = 25
fx[np.arange(f0, nFFT-f0)] = 0

# inverse fourier transform
y = np.real(ifft(fx))
Y = np.abs(fft(y)) / N
```



See, “L06_spectra_2.py”

Section 3. Time and frequency domains

Time and frequency domains

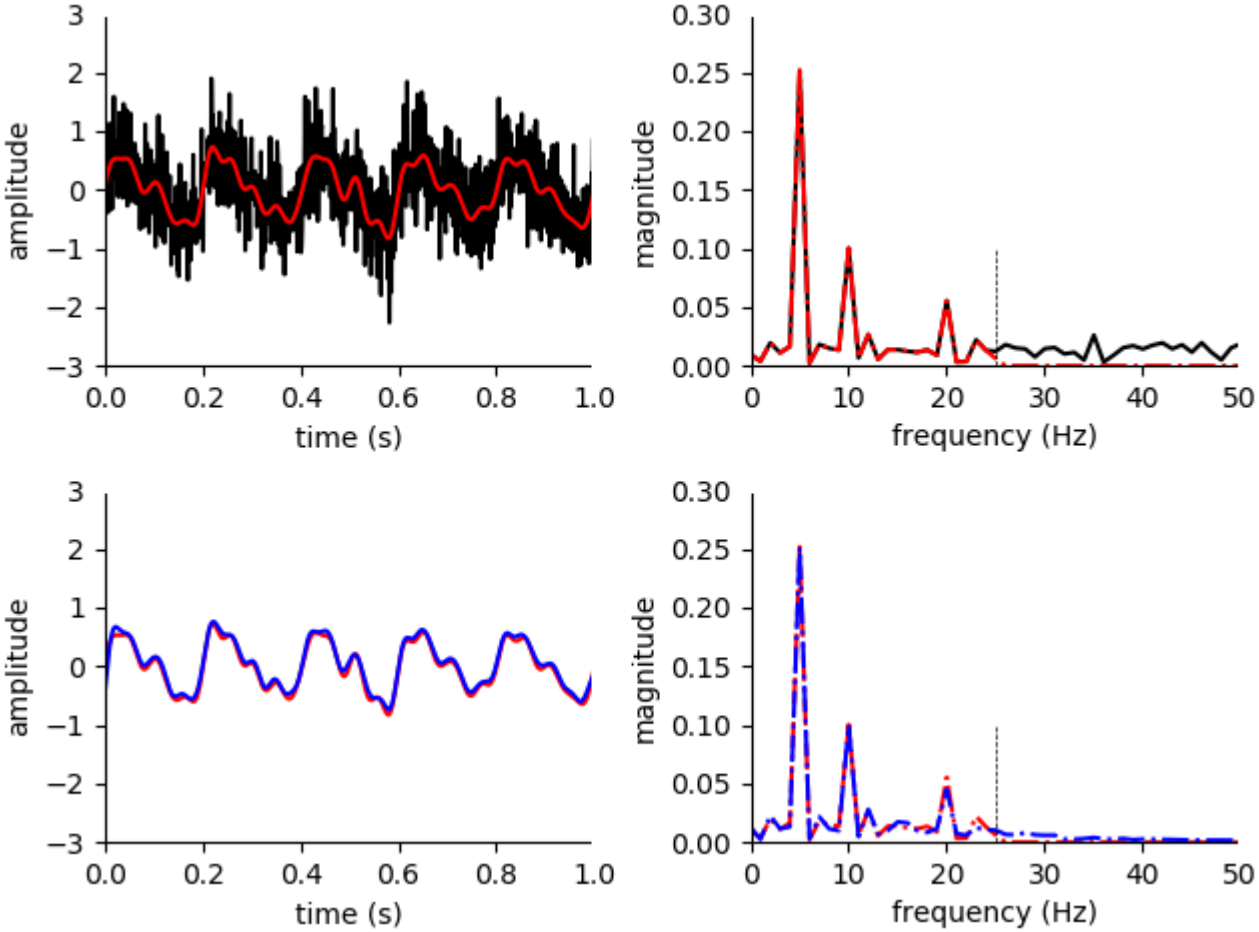
Filtering can be done in frequency and time domains.

Frequency domain

```
# fourier transform
fx = fft(x)
# remove frequencies above f0
fx[np.arange(f0, nFFT-f0)] = 0
# inverse fourier transform
y = ifft(fx)
```

Time domain

```
# design filter
[b, a] = signal.butter(4, f0 / (fs/2), 'low')
# apply filter
u = signal.filtfilt(b, a, x)
```



See, “L06_time_frequency_domains.py”

Applying filter

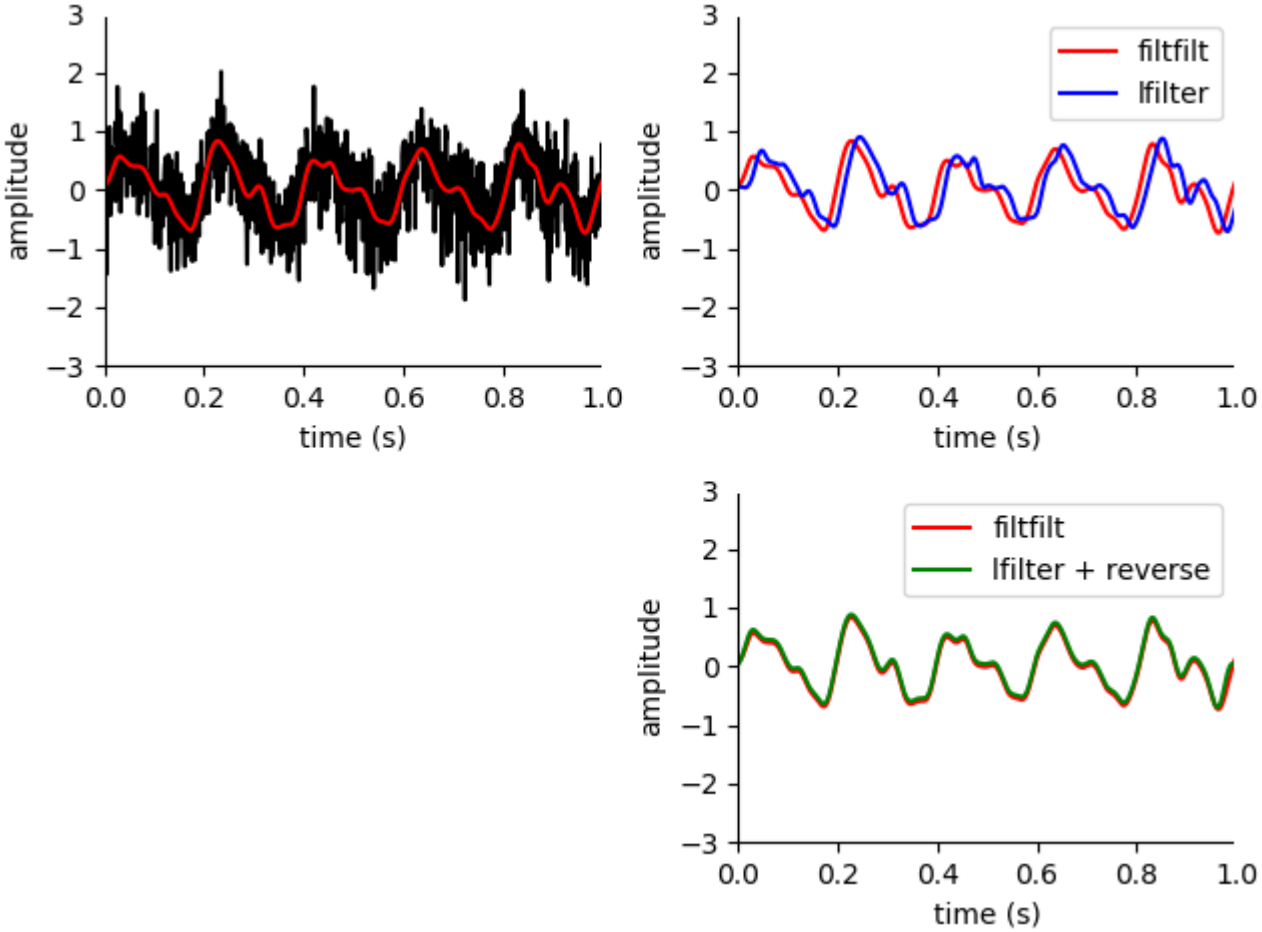
Normally, there is a lag between signal and its filtered copy which is equal to the length (number) of coefficients.

```
# zero-phase filtering
y = signal.filtfilt(b, a, x)

# tranditional filtering
u = signal.lfilter(b, a, x)


# zero-phase filtering using lfilter
z = lfilter(b, a, x)
z = z[::-1]
z = lfilter(b, a, z)
z = z[::-1]

# or in compact form
z = lfilter(b, a, lfilter(b, a, x)[::-1])[::-1]
```



See, “L06_apply_filter.py”

Filter coefficients and convolution

Convolution in time domain is equivalent to product in frequency domain.

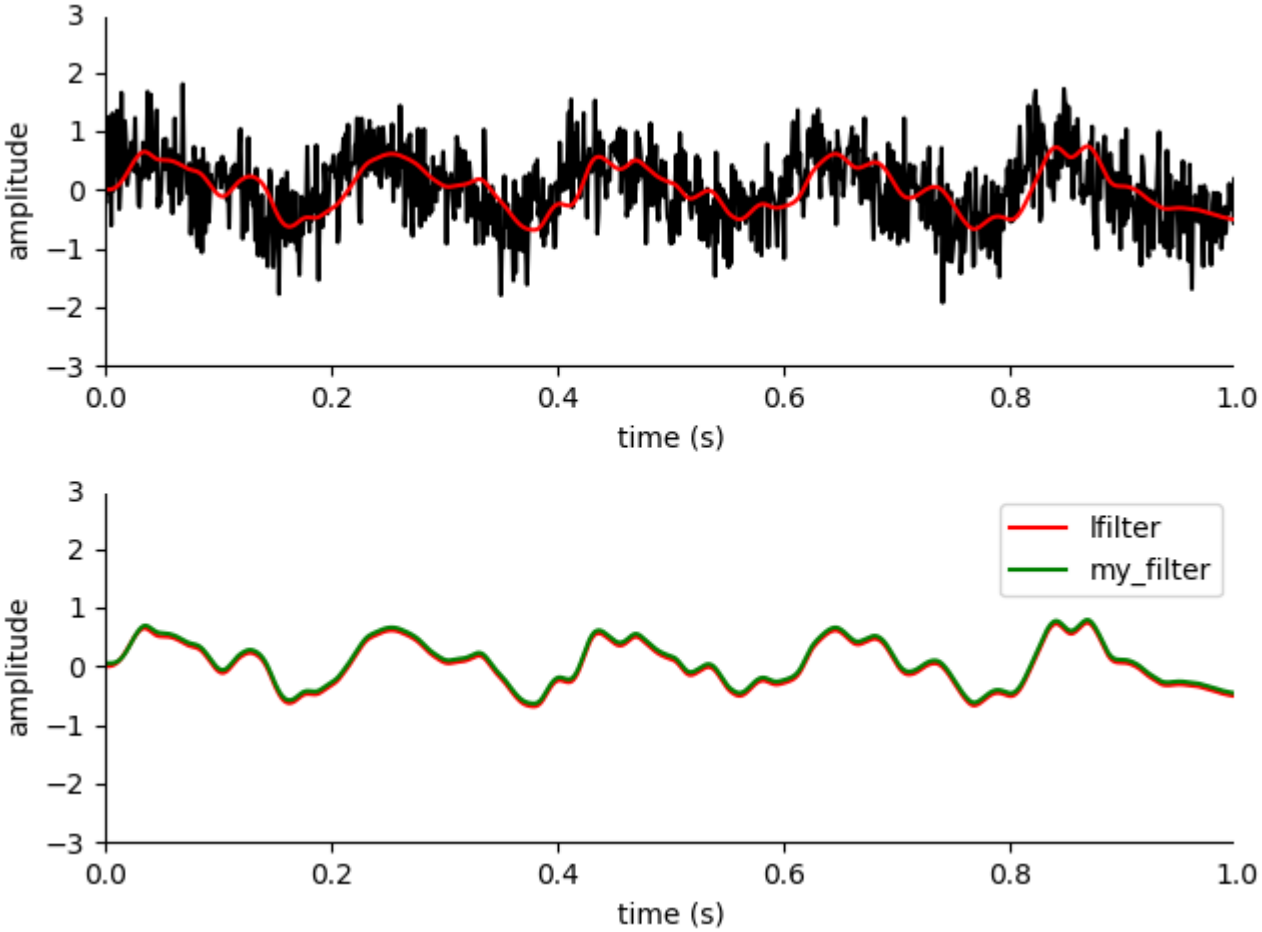
```
def my_filter(b, a, x):

  # get length
  N = len(x)
  M = max(len(b), len(a))

  # init
  y = np.zeros(N)

  # convolution
  for i in range(M, N):
    y[i] = np.sum(b * x[i:(i-M):-1]) -
           np.sum(a[1:] * y[(i-1):(i-M):-1])

  return y
```



See, “L06_apply_filter.py”

Filter parameters

What are the filter parameters?

```
# design filter
[b, a] = signal.butter(4, f0 / (fs/2), 'low')
```

Input parameters

- “butter” – Butterworth filter design (way to design filter)
- “4” – filter order
- “f0” – cutoff frequency
- “fs/2” – sampling rate
- “low” – low-pass filter (type of filter)

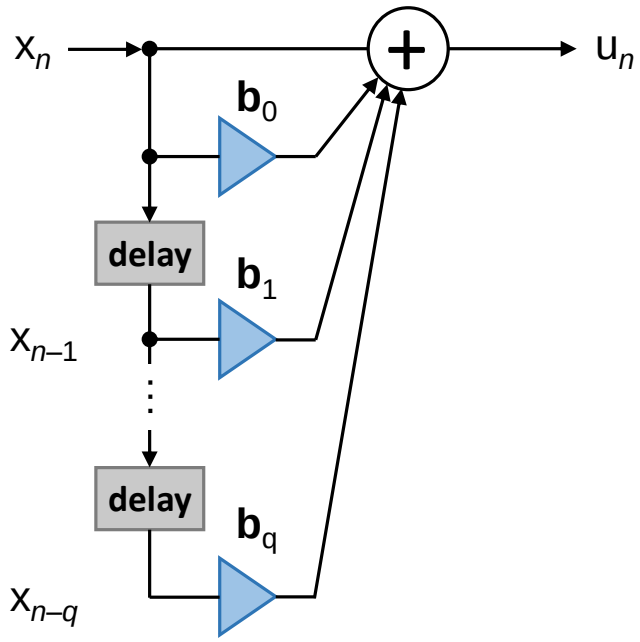
Output parameters

- “b, a” – filter coefficients

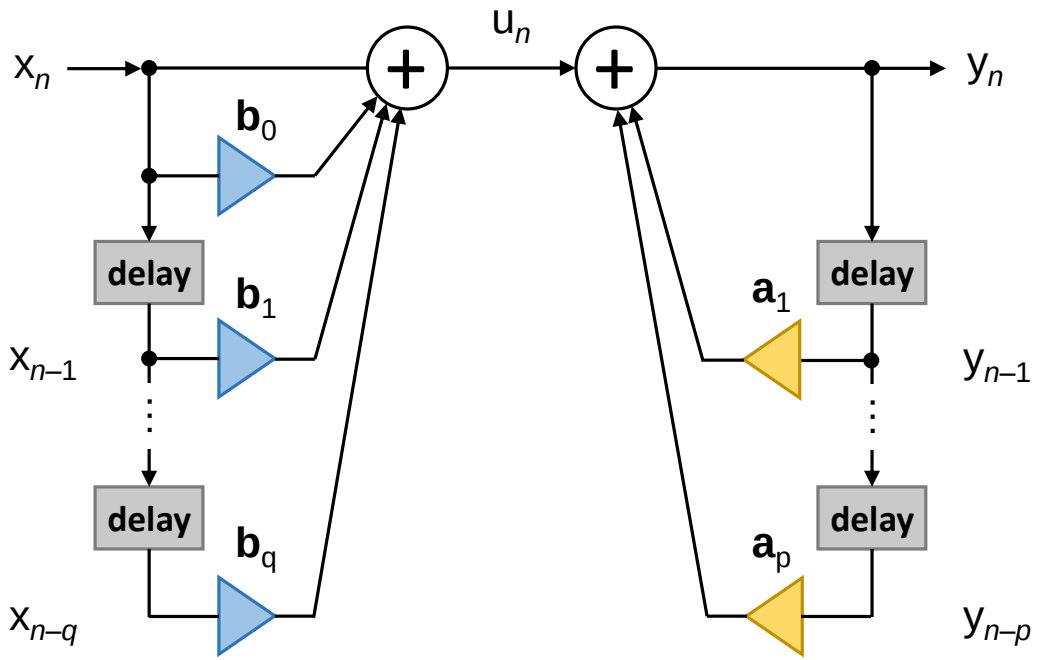
Section 4. **Filter design**

Impulse response (Finite IR / Infinite IR)

FIR / Direct form I



IIR / Direct form I



https://se.mathworks.com/help/signal/ref/dfilt.html

FIR filters design

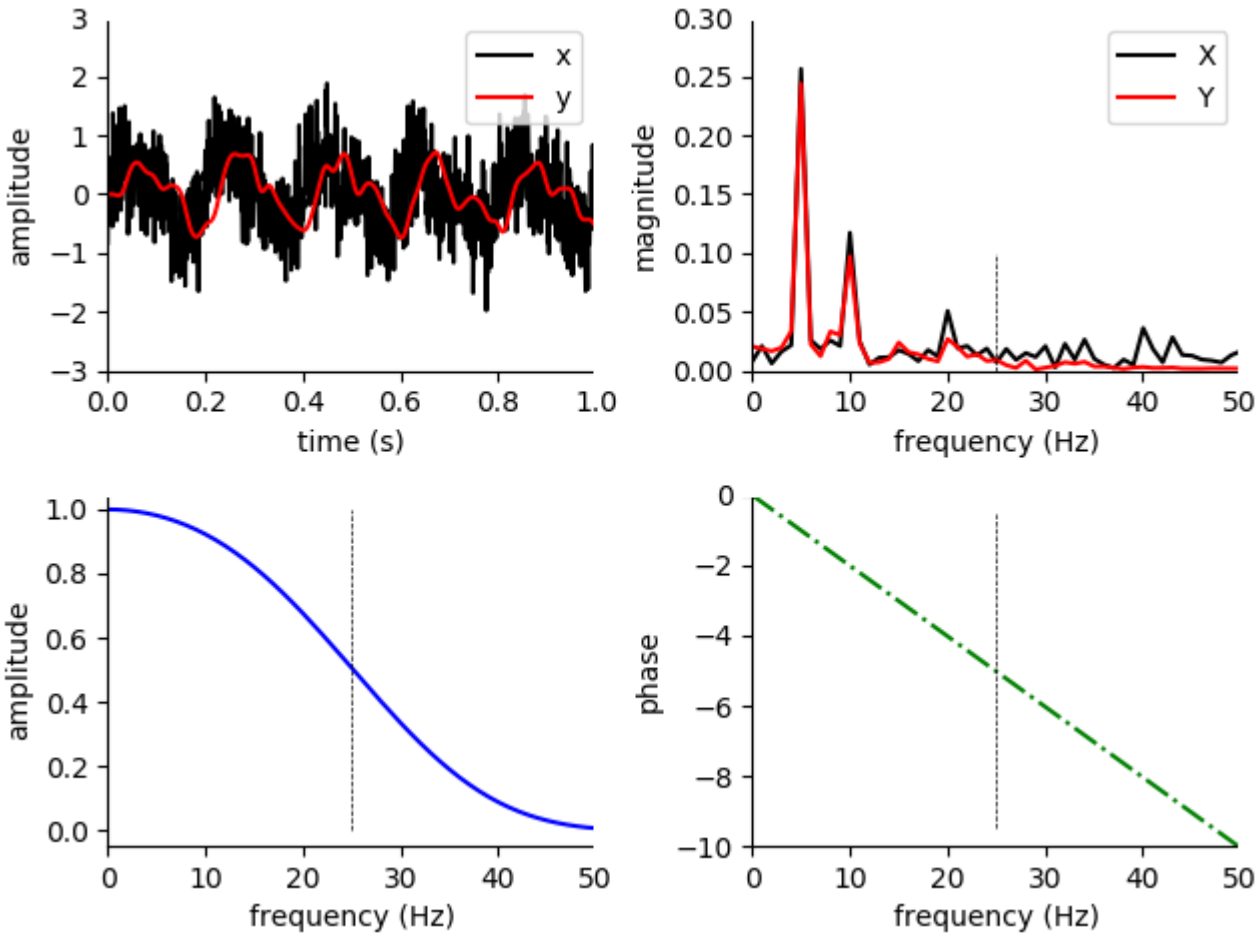
```
# design filter in time domain
f0 = 25
fc = f0 / (fs/2) # cutoff frequency
n = 65           # filter order

# low-pass filter
a = 1
b = signal.firwin(numtaps=n, cutoff=fc)
y = signal.lfilter(b, a, x)


# frequency response
w, h = signal.freqz(b, a, (fs//2))

# amplitude
ampl = np.abs(h)

# phase
phase = np.unwrap(np.angle(h))
```



See, “L06_fir_filter.py”

IIR filters design

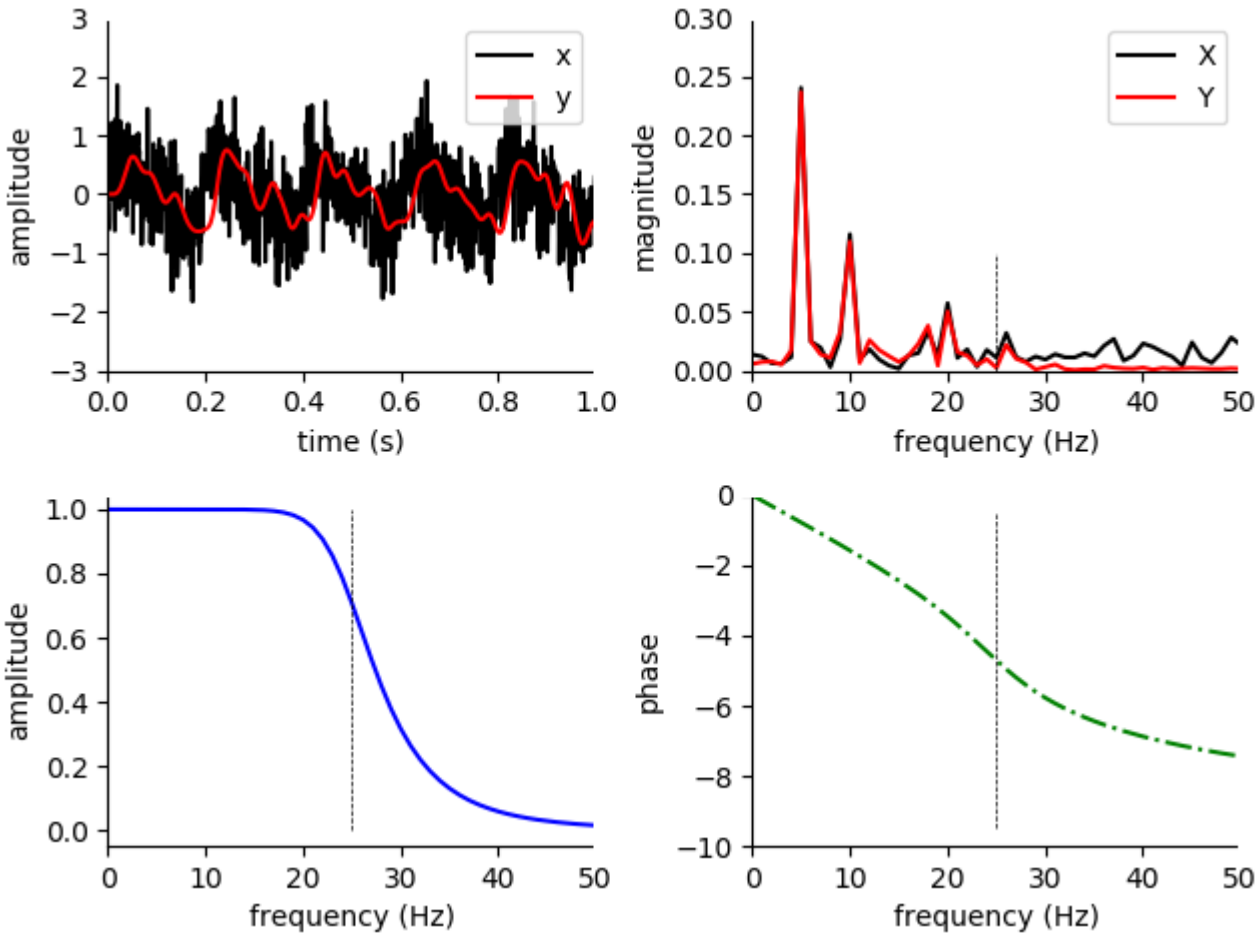
```
# design filter in time domain
f0 = 25
fc = f0 / (fs/2) # cutoff frequency
n = 6            # filter order

# low-pass filter
[b, a] = signal.butter(n, fc, 'low')
y = signal.lfilter(b, a, x)
```

```
# frequency response
w, h = signal.freqz(b, a, (fs//2))

# amplitude
ampl = np.abs(h)

# phase
phase = np.unwrap(np.angle(h))
```



See, “L06_iir_filter.py”

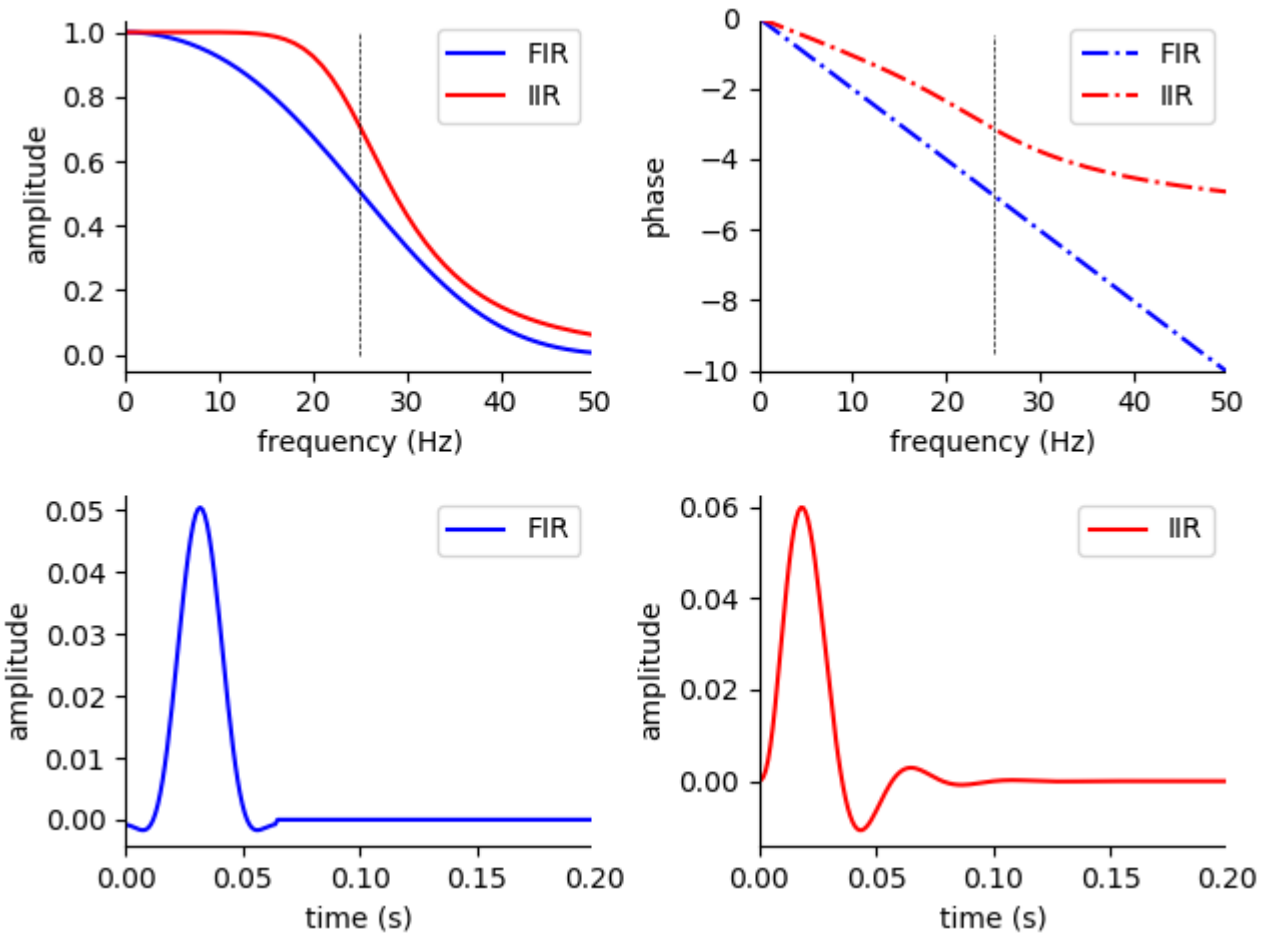
FIR vs IIR filters

```
# design filter
n1 = 65 # FIR filter order
n2 = 4  # IIR filter order

# low-pass FIR filter
a1 = 1
b1 = signal.firwin(numtaps=n1, cutoff=fc)

# low-pass IIR filter
[b2, a2] = signal.butter(n2, fc, 'low')


# impulse response
p = np.zeros(200)
p[0] = 1
resp = signal.lfilter(b, a, p)
```



See, "L06_fir_vs_iir_filter.py"

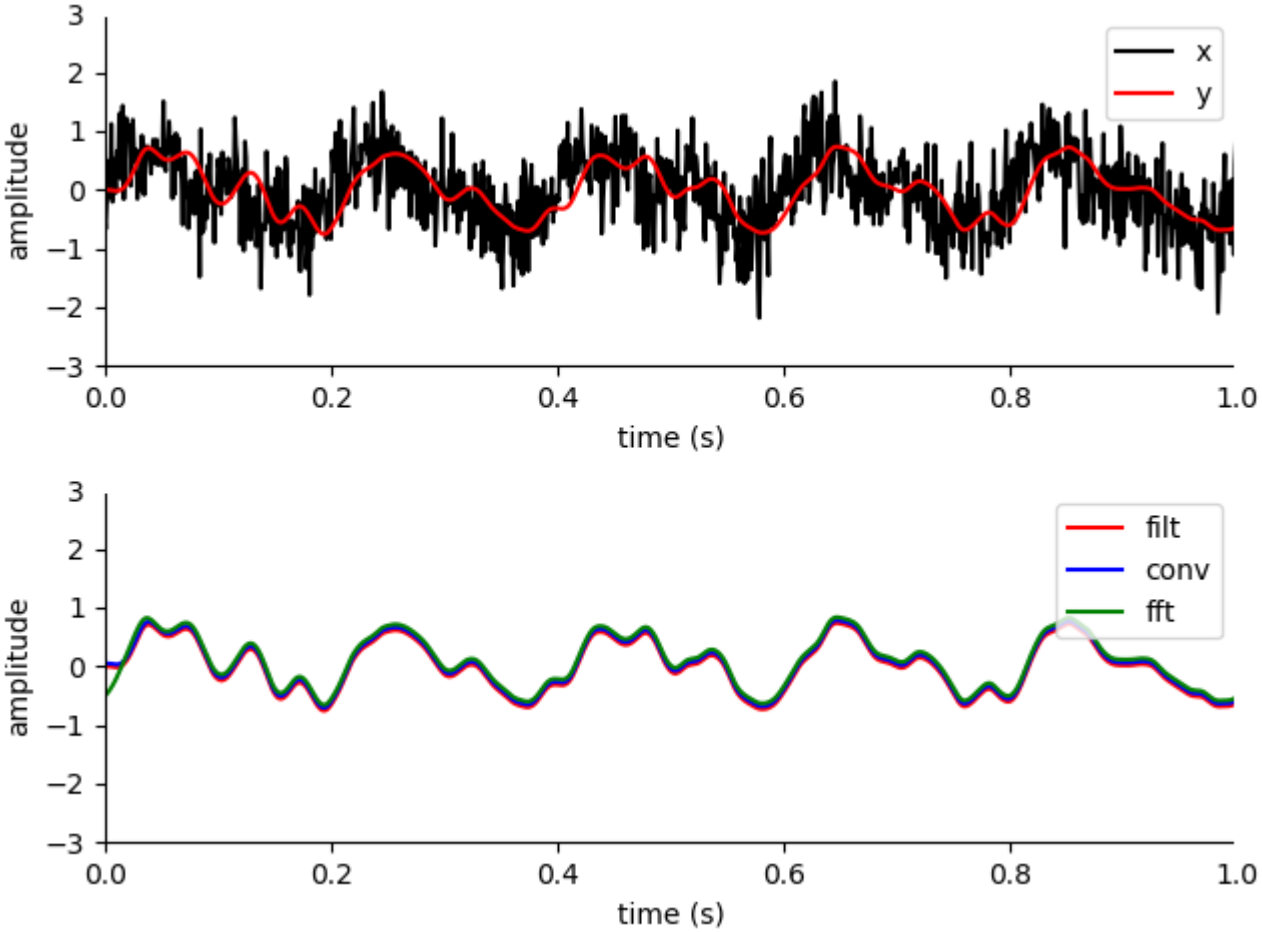
Impulse response

```
# low-pass IIR filter
n = 4
[b, a] = signal.butter(n, fc, 'low')
y = signal.lfilter(b, a, x)

# impulse response
p = np.zeros(200)
p[0] = 1
h = signal.lfilter(b, a, p)


# convolution
u = signal.convolve(x, h)
u = u[0:N]

# inverse fourier transform
z = ifft(fft(x, nFFT) * fft(h, nFFT))
```



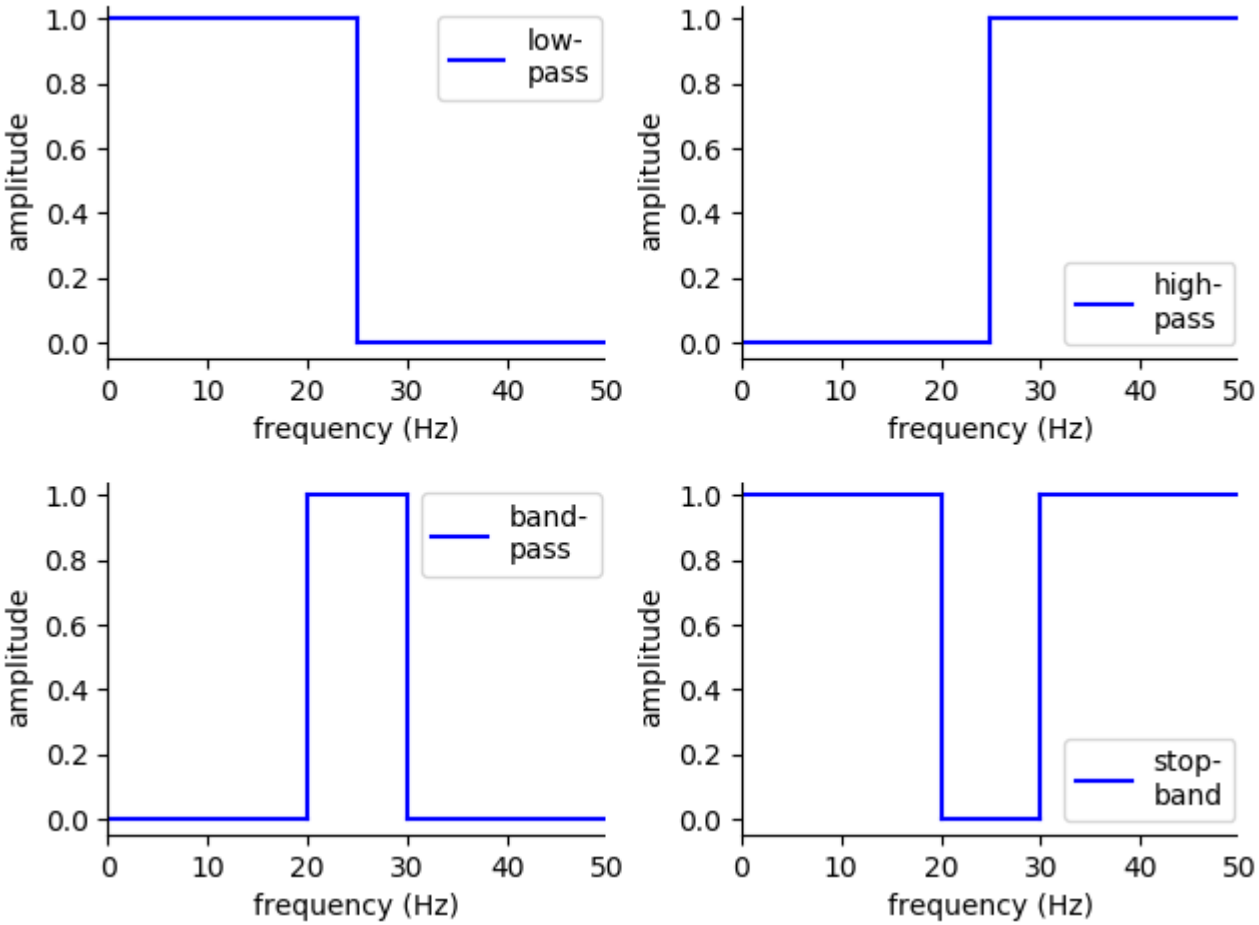
See, “L06_impulse_response.py”

Section 5. **Frequency response**

Frequency response (1/2)

There are four basic types of frequency response:

- low-pass – passes frequencies below given
- high-pass – passes frequencies above given
- band-pass – passes frequencies between two given
- stop-band – passes frequencies outside of two given

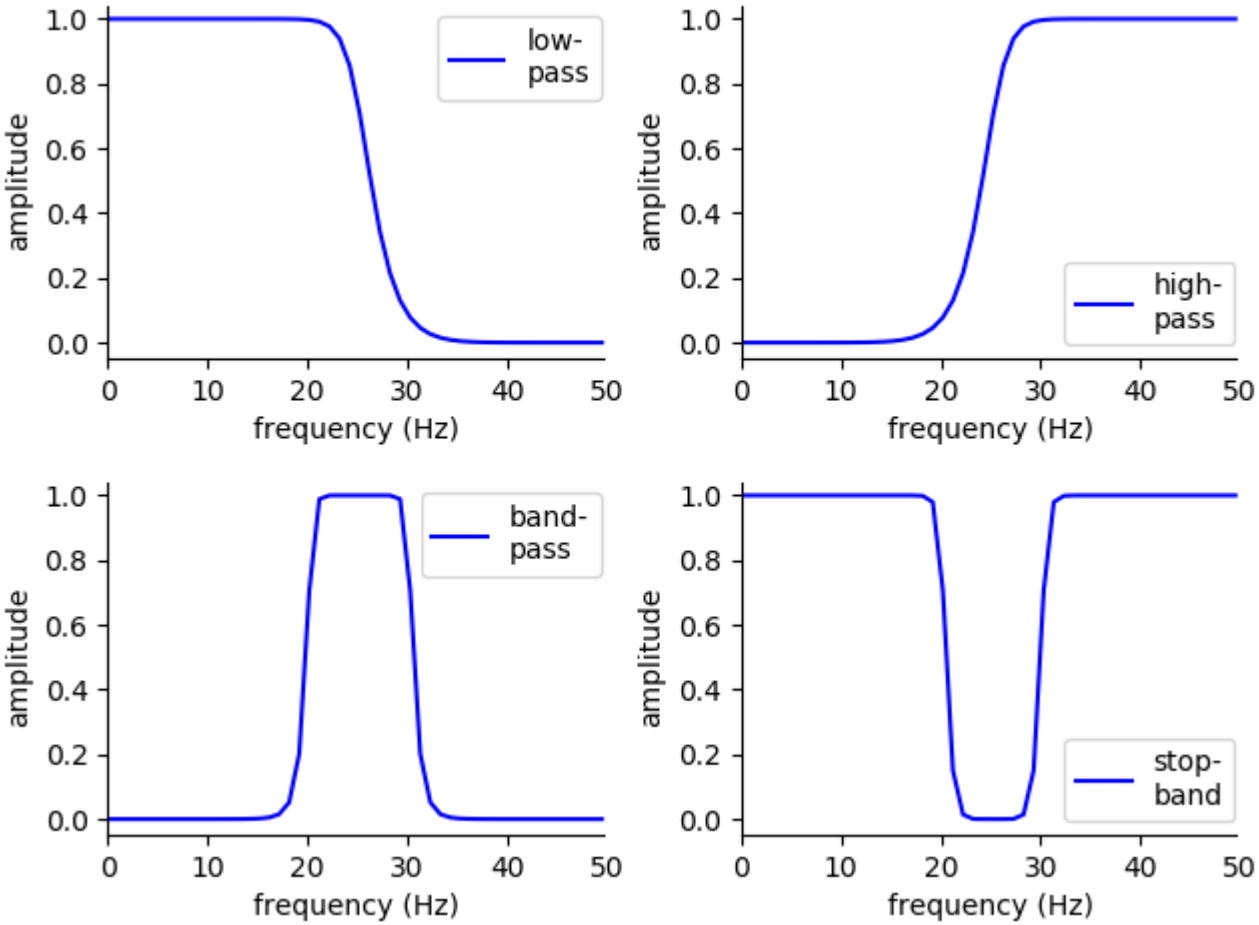


See, "L06_frequency_response_ideal.py"

Frequency response (2/2)

There are four basic types of frequency response:

- low-pass – passes frequencies below given
- high-pass – passes frequencies above given
- band-pass – passes frequencies between two given
- stop-band – passes frequencies outside of two given



See, "L06_frequency_response_real.py"

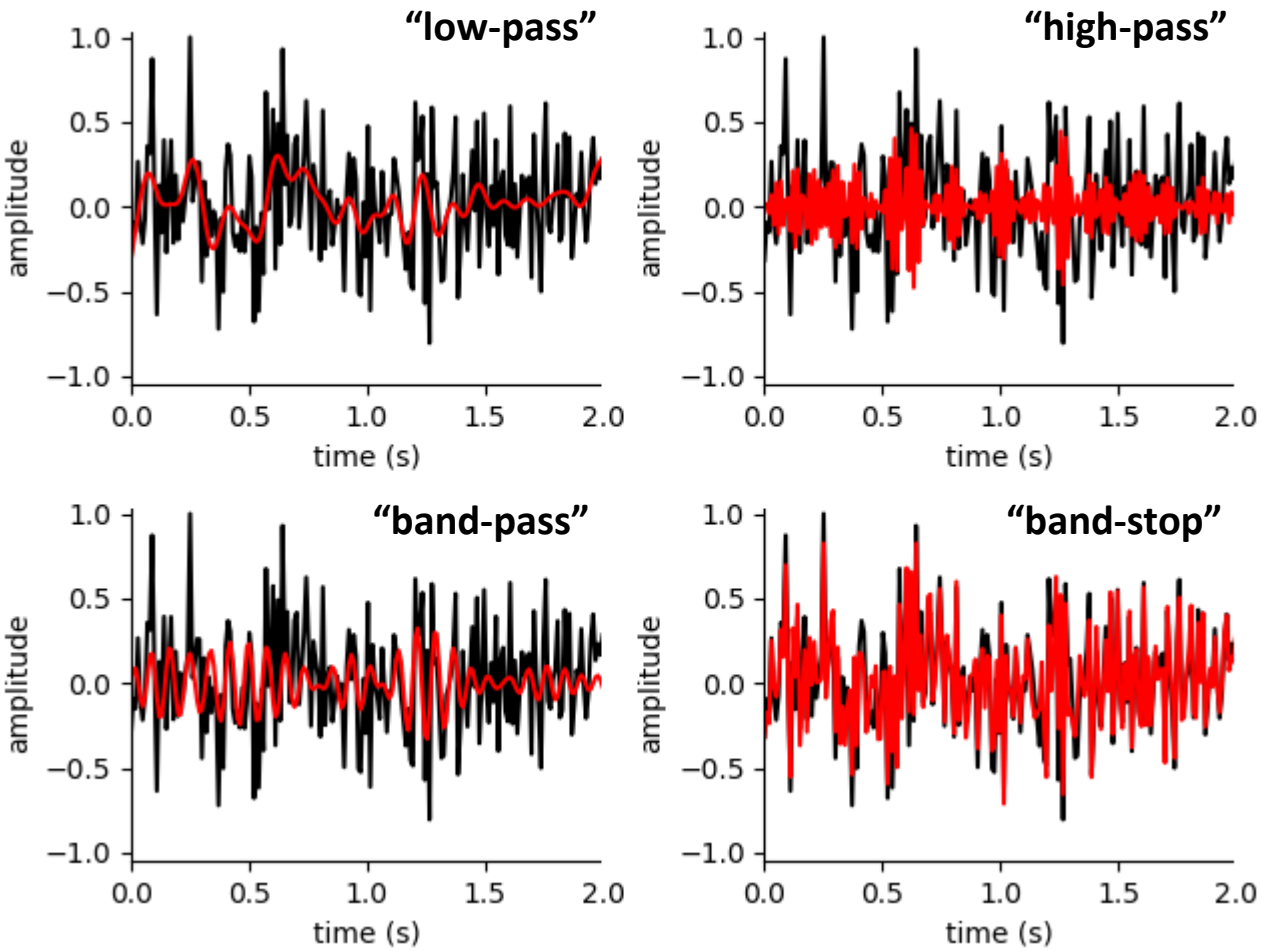
Filtered time series

```
# design low-pass IIR filter
[b, a] = signal.butter(8, 10.0/(fs/2),
         'lowpass')
y1 = signal.filtfilt(b, a, x)

# design high-pass IIR filter
[b, a] = signal.butter(8, 40.0/(fs/2),
         'highpass')
y2 = signal.filtfilt(b, a, x)

# design band-pass IIR filter
[b, a] = signal.butter(8, [10.0/(fs/2),
         15.0/(fs/2)], 'bandpass')
y3 = signal.filtfilt(b, a, x)

# design stop-band IIR filter
[b, a] = signal.butter(8, [10.0/(fs/2),
         15.0/(fs/2)], 'bandstop')
y4 = signal.filtfilt(b, a, x)
```



See, “L06_frequency_response_data.py”

Narrowband filter (1/2)

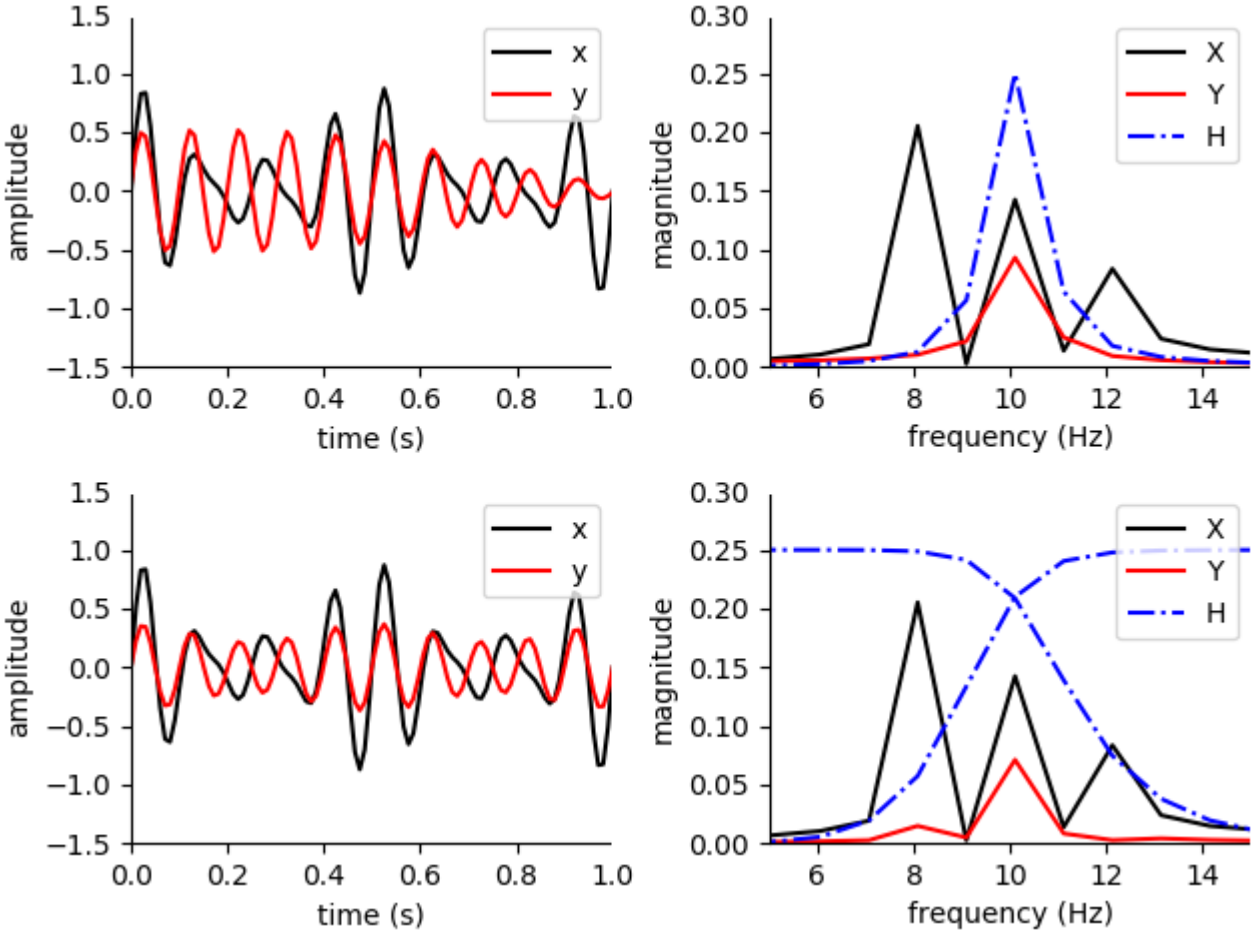
How to extract narrowband signal?

```
# signal
x = 0.4 * np.sin(2 * np.pi * 8 * t)  + \
    0.3 * np.sin(2 * np.pi * 10 * t) + \
    0.2 * np.sin(2 * np.pi * 12 * t)

# band-pass filter
n = 2
[b, a] = signal.butter(n,
         [9.5/(fs/2), 10.5/(fs/2)], 'bandpass')


# low-pass and high-pass filters
n = 8
[bl, al] = signal.butter(n, 10.5/(fs/2),
           'lowpass')

[bh, ah] = signal.butter(n, 9.5/(fs/2),
           'highpass')
```



See, "L06_narrowband_filter.py"

Narrowband filter (2/2)

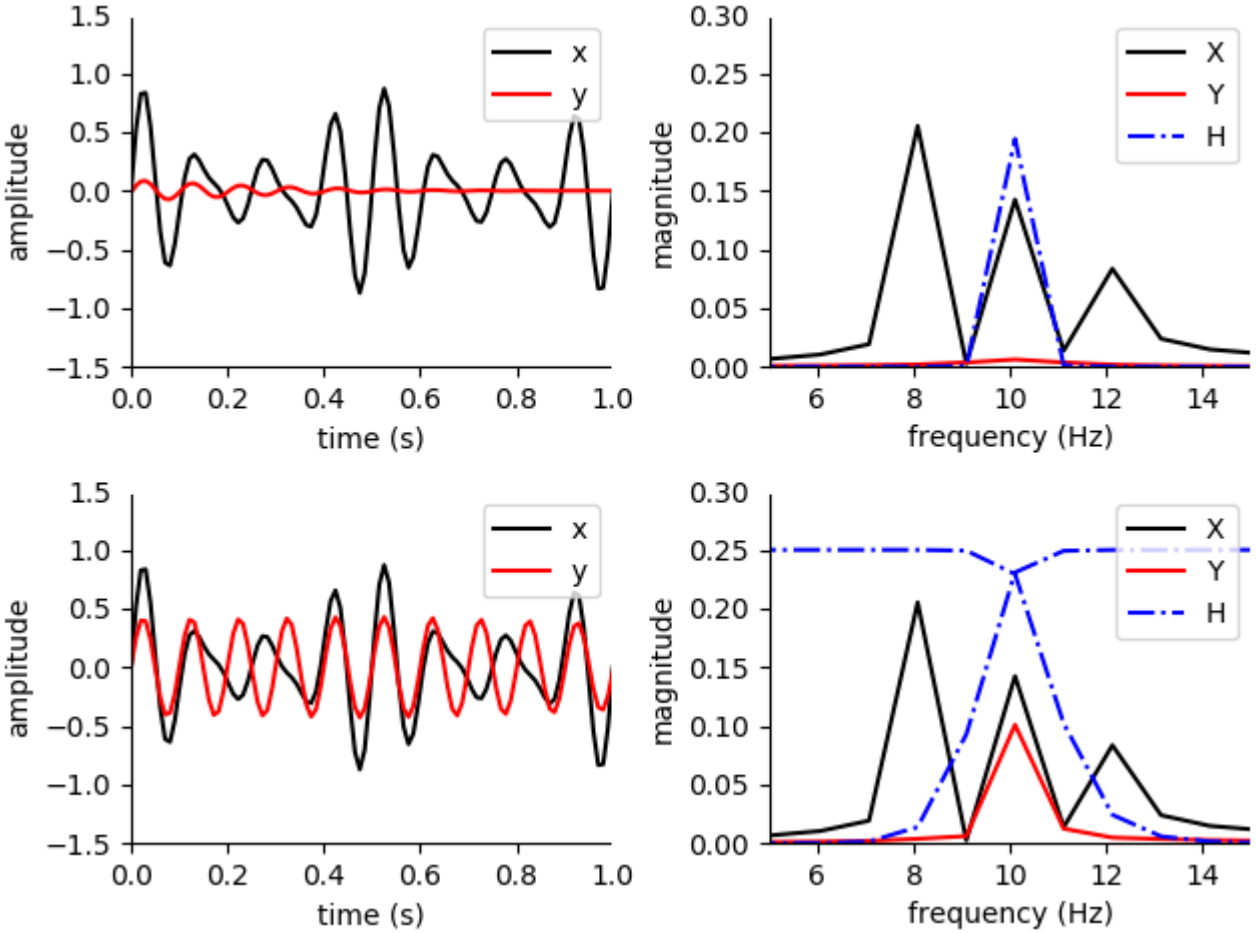
What if we increase the filter order?

```
# signal
x = 0.4 * np.sin(2 * np.pi * 8 * t)  + \
    0.3 * np.sin(2 * np.pi * 10 * t) + \
    0.2 * np.sin(2 * np.pi * 12 * t)

# band-pass filter
n = 8
[b, a] = signal.butter(n,
         [9.5/(fs/2), 10.5/(fs/2)], 'bandpass')


# low-pass and high-pass filters
n = 16
[bl, al] = signal.butter(n, 10.5/(fs/2),
           'lowpass')

[bh, ah] = signal.butter(n, 9.5/(fs/2),
           'highpass')
```



See, “L06_narrowband_filter.py”

Section 6. **Padding**

Padding (1/2)

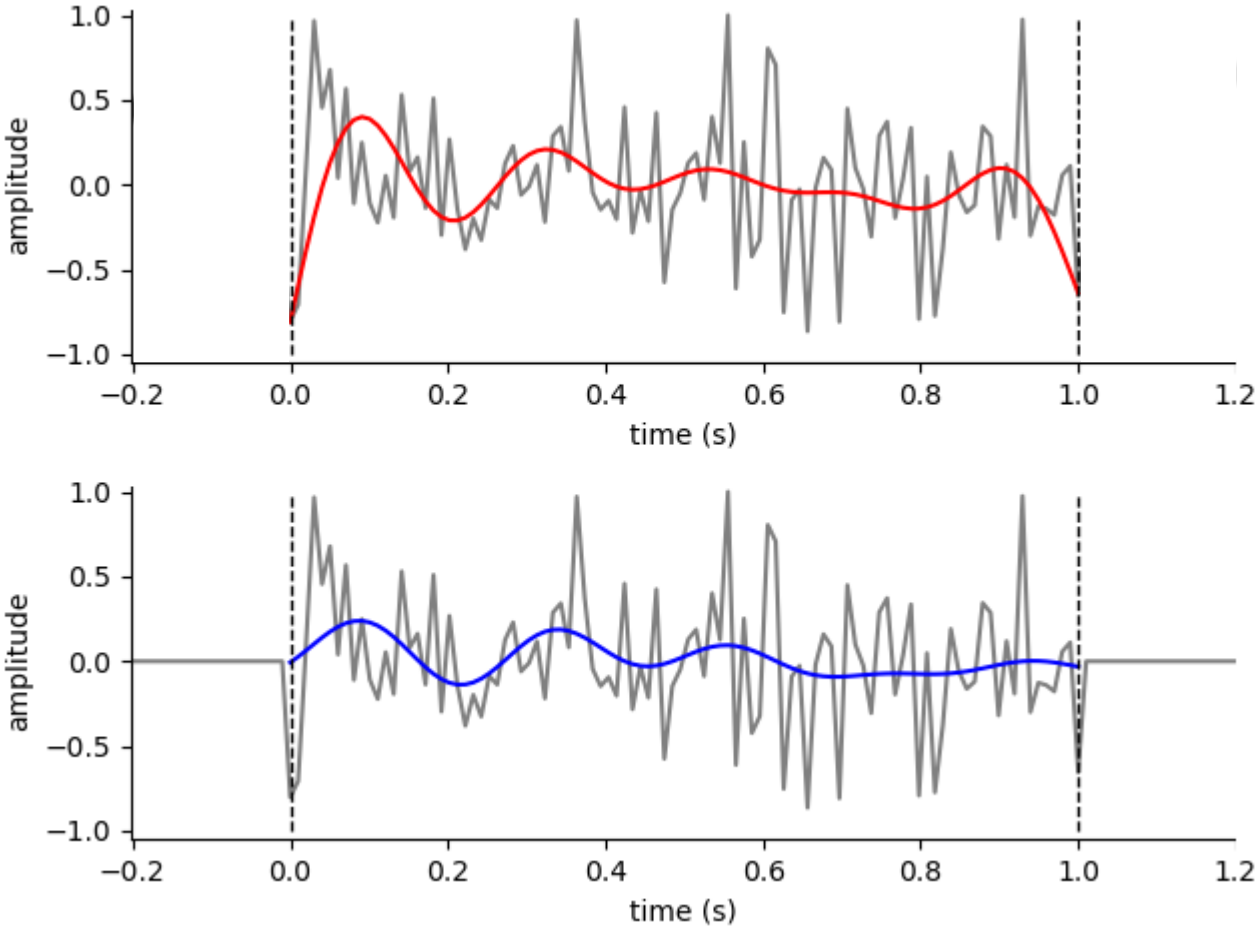
How to supress the “border” effects?

```
# design filter
fc = f0 / (fs/2) # cutoff frequency

# low-pass filter
n = 16
[b, a] = signal.butter(n, fc, 'lowpass')
y = signal.filtfilt(b, a, x)


# padding length
L = fs // f0

# zeros-padding
u = np.concatenate((np.zeros(L), x,
                    np.zeros(L)))
u = signal.filtfilt(b, a, u)
u = u[L:(N+L)]
```



See, “L06_padding.py”

Padding (2/2)

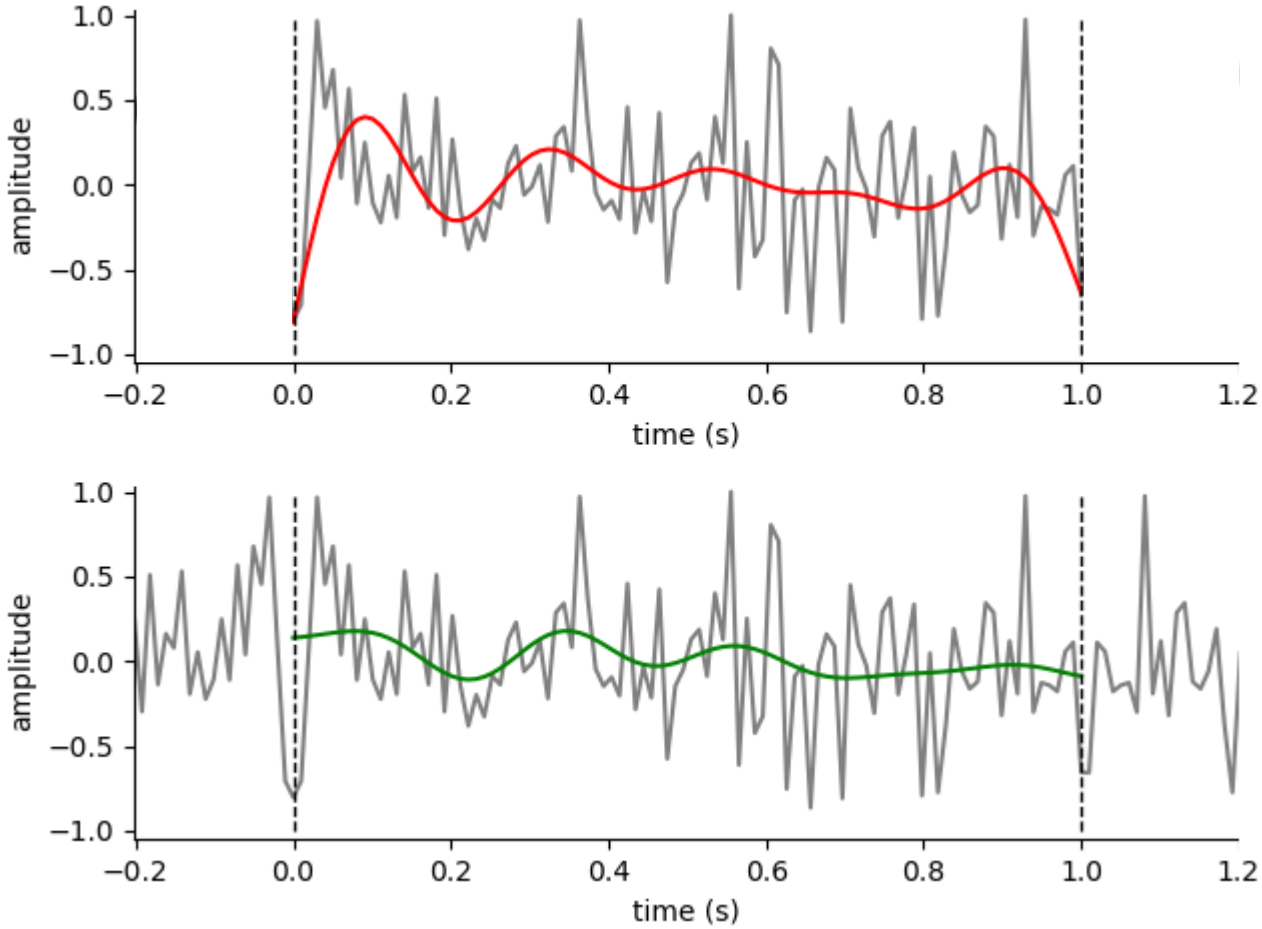
How to supress the “border” effects?

```
# design filter
fc = f0 / (fs/2) # cutoff frequency

# low-pass filter
n = 16
[b, a] = signal.butter(n, fc, 'lowpass')
y = signal.filtfilt(b, a, x)


# padding length
L = fs // f0

# signal-mirroring
u = np.concatenate((x[L:0:-1], x,
                    x[N:(N-L-1):-1]))
u = signal.filtfilt(b, a, u)
u = u[L:(N+L)]
```



See, “L06_padding.py”

Literature

- **Python programming language**

- http://www.scipy-lectures.org/, see “materials/L02_ScipyLectures.pdf”

- **Data analysis**

- Downey A., “**Think DSP: Digital Signal Processing in Python**”, see materials/L05_thinkdsp.pdf