A Journey in Signal Processing with Jupyter

JEAN-FRANÇOIS BERCHER

ESIEE-PARIS

Contents

1 A basic introduction to signals and systems

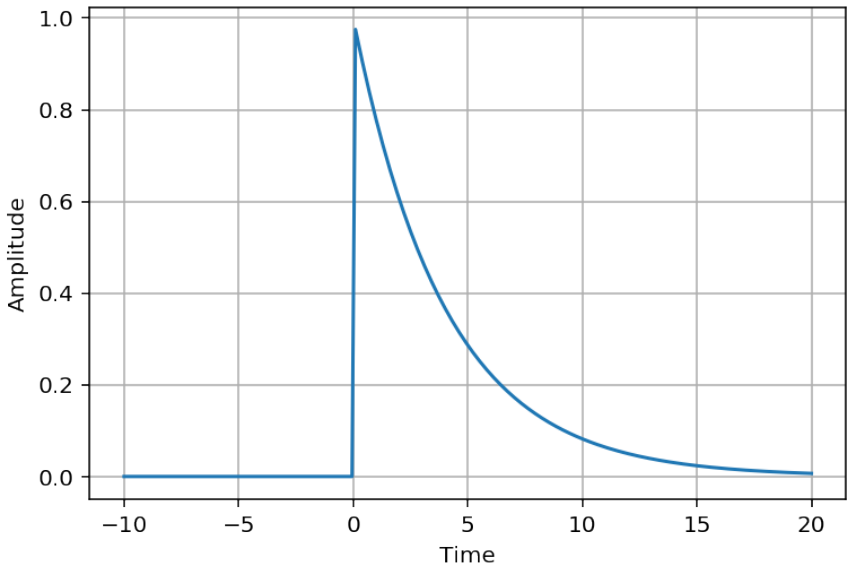
1.1 Effects of delays and scaling on signals

In this simple exercise, we recall the effect of delays and scaling on signals. It is important for students to experiments with that to ensure that they master these simple transformations.

Study the code below and experiment with the parameters

```
# Define a simple function
def f(t):
    return np.exp(-0.25*t) if t>0 else 0


T= np.linspace(-10,20,200)
L=len(T)
x=np.zeros(L)  # reserve some space for x

t0=0; a=1 # initial values

# Compute x as f(a*t+t0)
k=0
for t in T:
    x[k]=f(a*t+t0)
    k=k+1

# Plotting the signal
plt.plot(T,x)
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.grid(b=True)


# Experiment with several values of a and t0:
# a=1 t0=0
# a=1 t0=+5  (advance)
# a=1 t0=-5  (delay)
# a=-1 t0=0  (time reverse)
# a=-1 t0=5  (time reverse + advance)
# a=-1 t0=-5 (...)
```

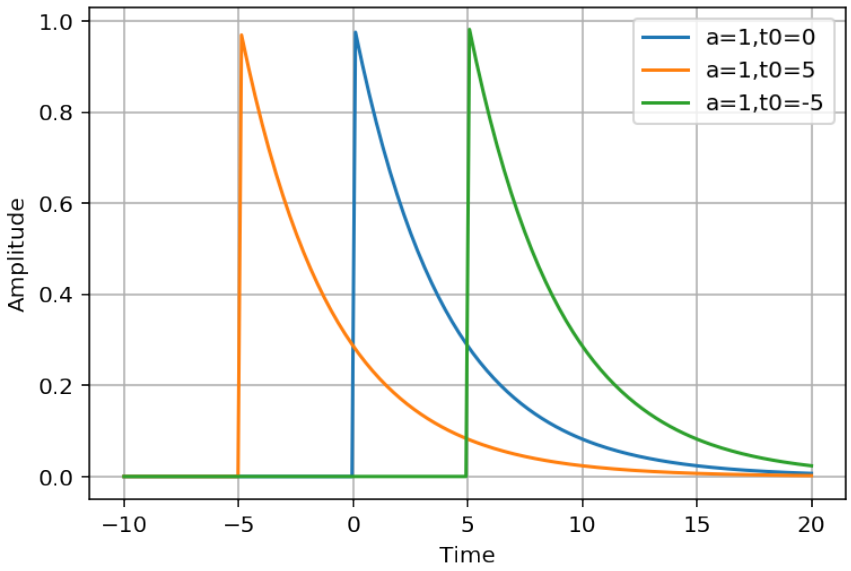
This to show that you do automatically several tests and plot the results all together.



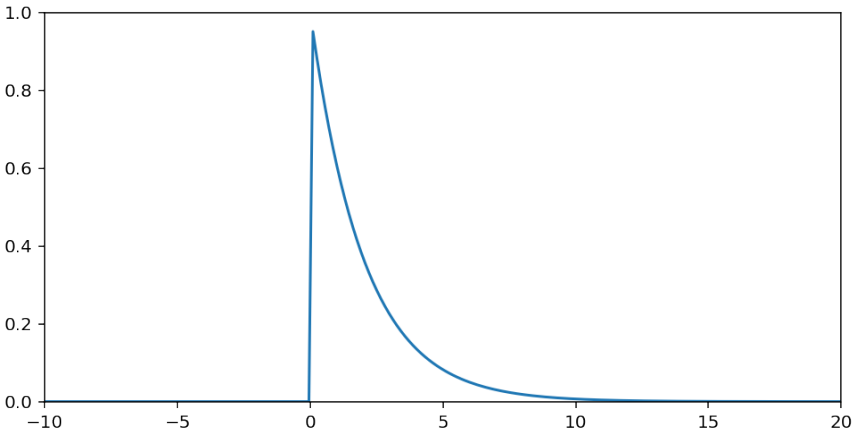

```
def compute_x(a,t0):
    k=0
    for t in T:
        x[k]=f(a*t+t0)
        k=k+1
    return x

list_tests=[(1,0),(1,5), (1,-5)]#,(-1,0),(-1,3), (-1,-3) ]
for (a,t0) in list_tests:
    x=compute_x(a,t0)
    plt.plot(T,x,label="a={},t0={}".format(a,t0))

plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.grid(b=True)
_=plt.legend()
```

And finally an interactive version

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (8,4)

def f(t):
    out=np.zeros(len(t))
    tpos=np.where(t>0)
    out[tpos]=np.exp(-0.25*t[tpos])
    return out

t= np.linspace(-10,20,200)
L=len(t)
x=np.zeros(L)

def compute_xx(t0,a):
        x=f(a*t+t0)
```




```
        plt.plot(t,x)
        plt.axis([-10, 20, 0, 1])

s_t0=widgets.FloatSlider(min=-20,max=20,step=1)
s_a=widgets.FloatSlider(min=0.1,max=5,step=0.1,value=2)
_=interact(compute_xx,t0=s_t0,a=s_a)
```



1.2 A basic introduction to filtering

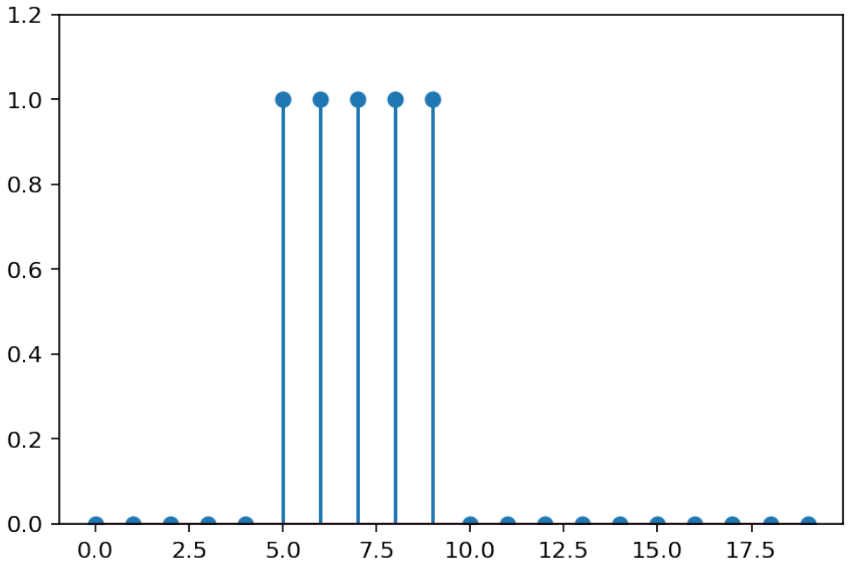
Through examples, we define several operations on signals and show how they transform them. Then we define what is a filter and the notion of impulse response.

- Section **??**
- Section 1.2.2
- Section **??**

1.2.1 Transformations of signals - Examples of difference equations

We begin by defining a test signal.

```
# rectangular pulse
N=20; L=5; M=10
r=np.zeros(N)

r[L:M]=1
#
plt.stem(r)
_=plt.ylim([0, 1.2])
```

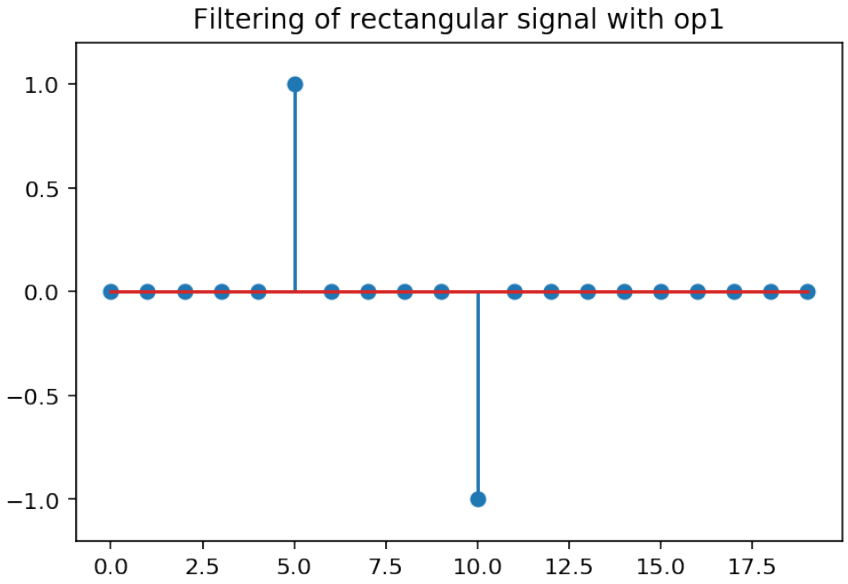


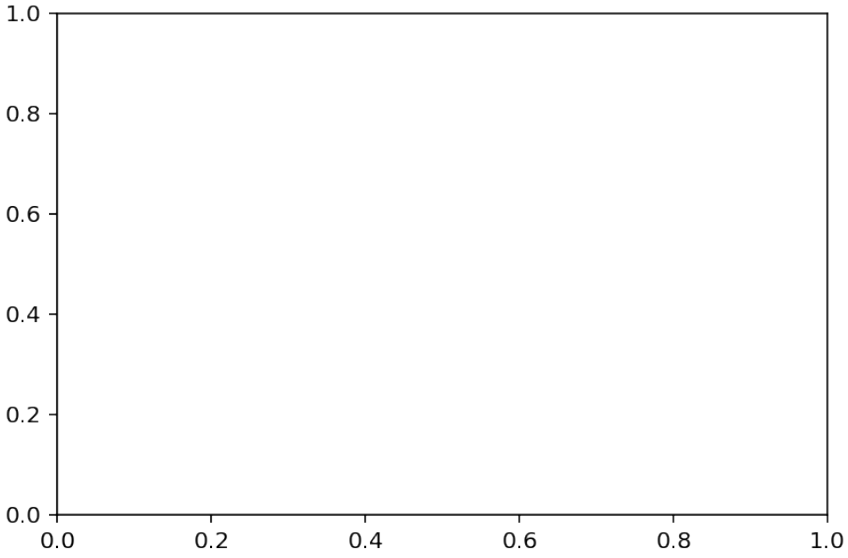

```
def op1(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]=signal[t]-signal[t-1]
    return transformed_signal

def op2(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]=0.5*signal[t]+0.5*signal[t-1]
    return transformed_signal
```

```
plt.figure()
plt.stem(op1(r))
_=plt.ylim([-1.2, 1.2])
plt.title("Filtering of rectangular signal with op1")
plt.figure()
plt.stem(op2(r),'r')
_=plt.ylim([-0.2, 1.2])
plt.title("Filtering of rectangular signal with op2")
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-4-e2dbae2047ad> in <module>()
      4 plt.title("Filtering of rectangular signal with op1")
      5 plt.figure()
----> 6 plt.stem(op2(r),'r')
      7 _=plt.ylim([-0.2, 1.2])
      8 plt.title("Filtering of rectangular signal with op2")

/usr/local/lib/python3.5/site-packages/matplotlib/pyplot.py in stem(linefmt
   2924         *args, linefmt=linefmt, markerfmt=markerfmt, basefmt=basefmt,
   2925         bottom=bottom, label=label, **({"data": data} if data is not
-> 2926         None else {}))
   2927
   2928


/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py in inner(ax,
   1808                         "the Matplotlib list!)" % (label_namer, func.__
   1809                         RuntimeWarning, stacklevel=2)
-> 1810             return func(ax, *args, **kwargs)
   1811
   1812         inner.__doc__ = _add_data_doc(inner.__doc__,


/usr/local/lib/python3.5/site-packages/matplotlib/axes/_axes.py in stem(sel
   2625         else:
   2626             x = y
-> 2627             y = np.asarray(args[0], dtype=float)
   2628             args = args[1:]
   2629


/usr/local/lib/python3.5/site-packages/numpy/core/numeric.py in asarray(a,
   499
   500     """
```

```
--> 501     return array(a, dtype, copy=False, order=order)
    502
    503


    ValueError: could not convert string to float: 'r'
```

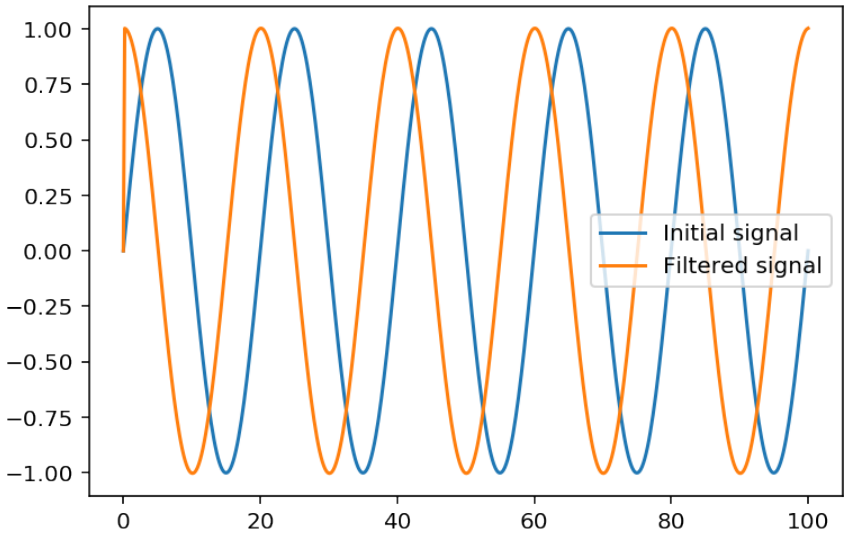




We define a sine wave and check that the operation implemented by "op1" seems to be a derivative...

```
t=np.linspace(0,100,500)
```

```
sig=np.sin(2*pi*0.05*t)
plt.plot(t,sig, label="Initial signal")
plt.plot(t,5/(2*pi*0.05)*op1(sig), label="Filtered signal")
plt.legend()
```

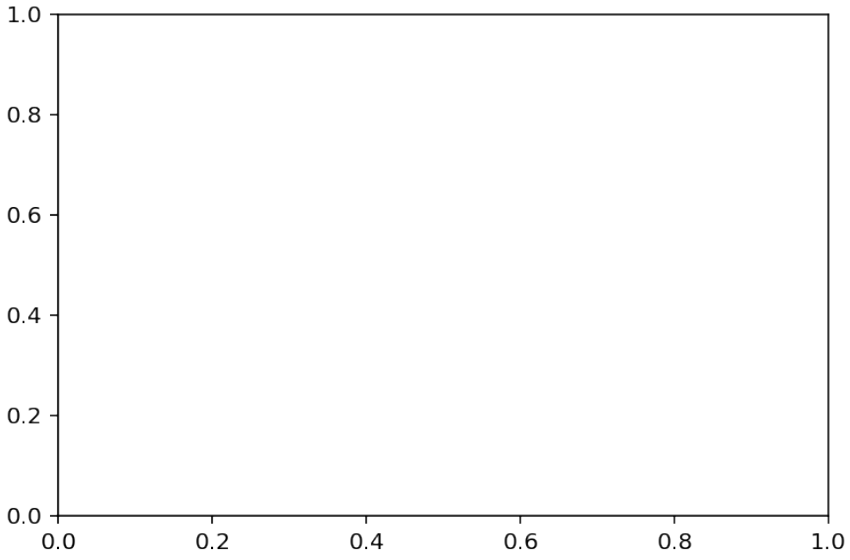


Composition of operations:

```
plt.stem(op1(op2(r)),'r')
_=plt.ylim([-1.2, 1.2])
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-6-96c17cc23118> in <module>()
----> 1 plt.stem(op1(op2(r)),'r')
      2 _=plt.ylim([-1.2, 1.2])


/usr/local/lib/python3.5/site-packages/matplotlib/pyplot.py in stem(linefmt
   2924         *args, linefmt=linefmt, markerfmt=markerfmt, basefmt=basefmt,
   2925         bottom=bottom, label=label, **({"data": data} if data is not
-> 2926         None else {}))
   2927
   2928


/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py in inner(ax,
   1808                         "the Matplotlib list!)" % (label_namer, func.__
   1809                         RuntimeWarning, stacklevel=2)
-> 1810             return func(ax, *args, **kwargs)
```

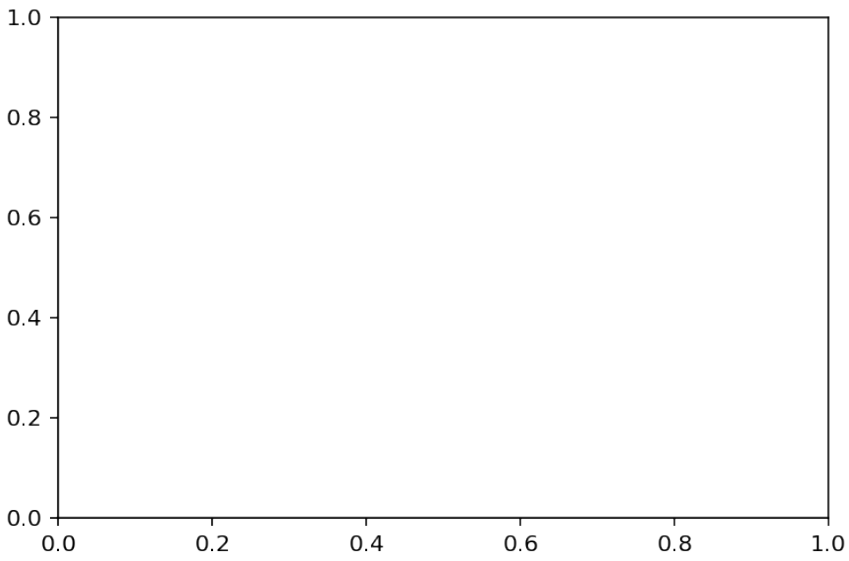
```
   1811
   1812         inner.__doc__ = _add_data_doc(inner.__doc__,


    /usr/local/lib/python3.5/site-packages/matplotlib/axes/_axes.py in stem(sel
   2625         else:
   2626             x = y
-> 2627             y = np.asarray(args[0], dtype=float)
   2628             args = args[1:]
   2629


    /usr/local/lib/python3.5/site-packages/numpy/core/numeric.py in asarray(a,
    499
    500     """
--> 501     return array(a, dtype, copy=False, order=order)
    502
    503


    ValueError: could not convert string to float: 'r'
```




```
def op3(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]= 0.7*transformed_signal[t-1]+signal[t]
    return transformed_signal

plt.stem(op3(r),'r')
plt.title("Filtering of rectangular signal with op3")
```

```
_=plt.ylim([-0.2, 3.2])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-7-3cf0c745df68> in <module>()
      5     return transformed_signal
      6
----> 7 plt.stem(op3(r),'r')
      8 plt.title("Filtering of rectangular signal with op3")
      9 _=plt.ylim([-0.2, 3.2])


/usr/local/lib/python3.5/site-packages/matplotlib/pyplot.py in stem(linefmt
   2924         *args, linefmt=linefmt, markerfmt=markerfmt, basefmt=basefmt,
   2925         bottom=bottom, label=label, **({"data": data} if data is not
-> 2926         None else {}))
   2927
   2928


/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py in inner(ax,
   1808                         "the Matplotlib list!)" % (label_namer, func.__
   1809                         RuntimeWarning, stacklevel=2)
-> 1810             return func(ax, *args, **kwargs)
   1811
   1812         inner.__doc__ = _add_data_doc(inner.__doc__,


/usr/local/lib/python3.5/site-packages/matplotlib/axes/_axes.py in stem(sel
   2625         else:
   2626             x = y
-> 2627             y = np.asarray(args[0], dtype=float)
   2628             args = args[1:]
   2629


/usr/local/lib/python3.5/site-packages/numpy/core/numeric.py in asarray(a,
    499
    500     """
--> 501     return array(a, dtype, copy=False, order=order)
    502
    503


ValueError: could not convert string to float: 'r'
```



A curiosity

```
def op4(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]= 1*transformed_signal[t-1]+signal[t]
    return transformed_signal

plt.stem(op4(r),'r')
plt.title("Filtering of rectangular signal with op4")
_=plt.ylim([-0.2, 5.6])
# And then..
plt.figure()
plt.stem(op1(op4(r)),'r')
plt.title("Filtering of rectangular signal with op1(op4)")
_=plt.ylim([-0.2, 1.2])
```

```
    ---------------------------------------------------------------------------

    ValueError                                Traceback (most recent call last)

    <ipython-input-8-46cc0d896a24> in <module>()
      5     return transformed_signal
      6
----> 7 plt.stem(op4(r),'r')
      8 plt.title("Filtering of rectangular signal with op4")
      9 _=plt.ylim([-0.2, 5.6])
```

```
    /usr/local/lib/python3.5/site-packages/matplotlib/pyplot.py in stem(linefmt
   2924         *args, linefmt=linefmt, markerfmt=markerfmt, basefmt=basefmt,
   2925         bottom=bottom, label=label, **({"data": data} if data is not
-> 2926         None else {}))
   2927
   2928


    /usr/local/lib/python3.5/site-packages/matplotlib/__init__.py in inner(ax,
   1808                         "the Matplotlib list!)" % (label_namer, func.__
   1809                         RuntimeWarning, stacklevel=2)
-> 1810             return func(ax, *args, **kwargs)
   1811
   1812         inner.__doc__ = _add_data_doc(inner.__doc__,


    /usr/local/lib/python3.5/site-packages/matplotlib/axes/_axes.py in stem(sel
   2625         else:
   2626             x = y
-> 2627             y = np.asarray(args[0], dtype=float)
   2628             args = args[1:]
   2629


    /usr/local/lib/python3.5/site-packages/numpy/core/numeric.py in asarray(a,
    499
    500     """
--> 501     return array(a, dtype, copy=False, order=order)
    502
    503


    ValueError: could not convert string to float: 'r'
```

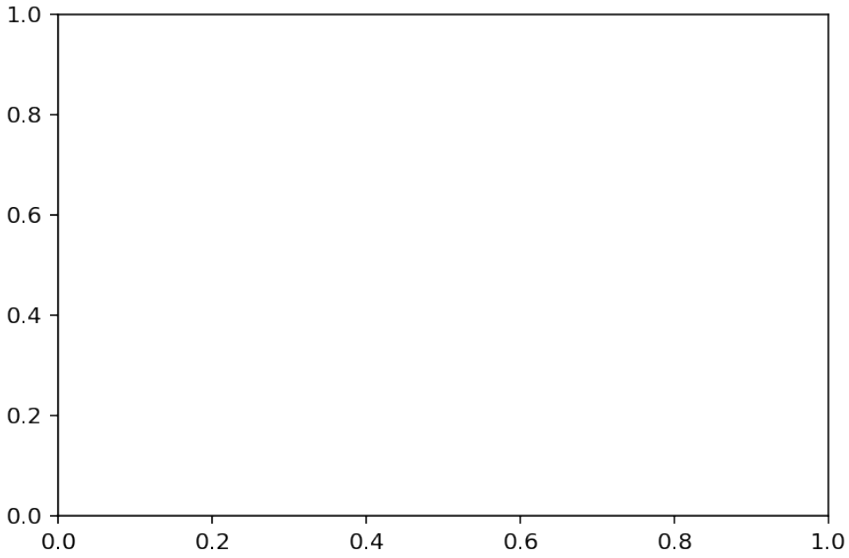
1.2.2 Filters

Definition A filter is a time-invariant linear system.

- Time invariance means that if $y(n)$ is the response associated with an input $x(n)$, then $y(n-n_0)$ is the response associated with the input $x(n-n_0)$.
- Linearity means that if $y_1(n)$ and $y_2(n)$ are the outputs associated with $x_1(n)$ and $x_2(n)$, then the output associated with $a_1x_1(n)+a_2x_2(n)$ is $a_1y_1(n)+a_2y_2(n)$ (superposition principle)

Exercise 1. *Check whether the following systems are filters or not.*

- $x(n) \to 2x(n)$



- $x(n) \to 2x(n)+1$
- $x(n) \to 2x(n)+x(n-1)$
- $x(n) \to x(n)^2$

Notion of impulse response

Definition 1. *A Dirac impulse (or impulse for short) is defined by*

$$\delta(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{elsewhere} \end{cases}$$

Definition 2. *The impulse response of a system is nothing but the output of the system excited by a Dirac impulse. It is often denoted $h(h)$.*

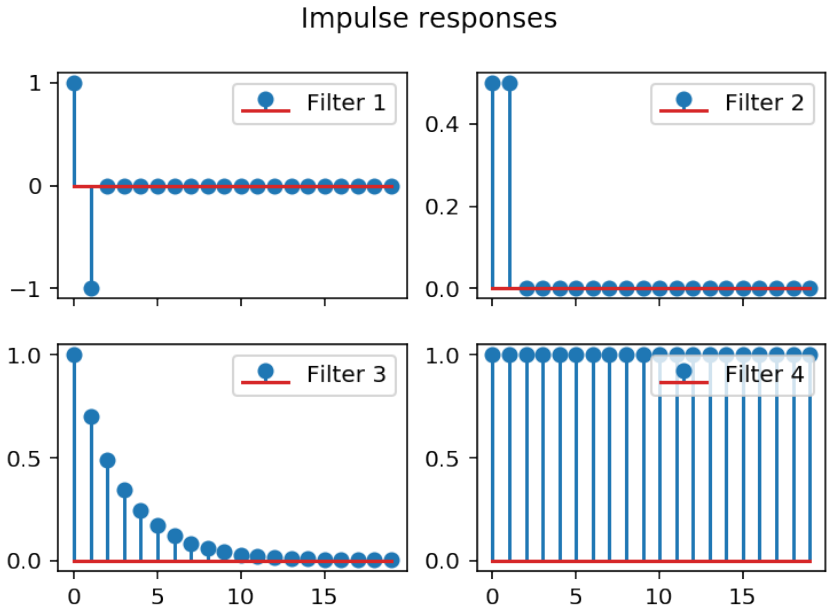
$$\delta(n) \to System \to h(n)$$

```
def dirac(n):
# dirac function
    return 1 if n==0 else 0
def dirac_vector(N):
    out = np.zeros(N)
    out[0]=1
    return out
```

```
d=dirac_vector(20)
fig,ax=plt.subplots(2,2,sharex=True)

ax[0][0].stem(op1(d),  label="Filter 1")
ax[0][0].legend()
ax[0][1].stem(op2(d),  label="Filter 2")
ax[0][1].legend()
ax[1][0].stem(op3(d), label="Filter 3")
ax[1][0].legend()
```

```
ax[1][1].stem(op4(d), label="Filter 4")
ax[1][1].legend()
plt.suptitle("Impulse responses")
```

```
Text(0.5, 0.98, 'Impulse responses')
```

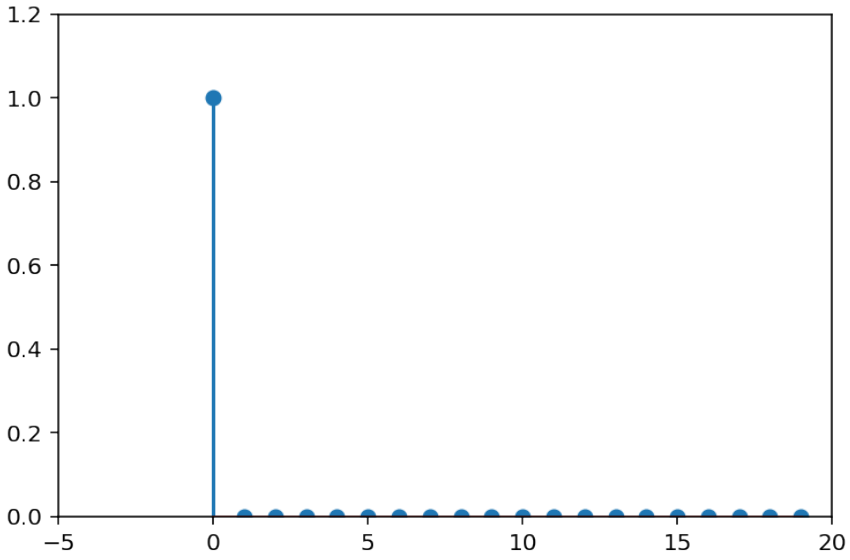


Curiosity (continued)

The impulse response of op4(op1) is given by

```
h=op4(op1(dirac_vector(20)))
plt.stem(h, label="Filter 4(1)")
_=plt.axis([-5, 20, 0, 1.2])
```

This is nothing but a Dirac impulse! We already observed that op4(op1(signal))=signal; that is the filter is an identity transformation. In other words, op4 acts as the "inverse" of op1. Finally, we note that the impulse response of the indentity filter is a Dirac impulse.


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−5
0
5
10
15
20

2

Introduction to the Fourier representation

We begin by a simple example which shows that the addition of some sine waves, with special coefficients, converges constructively. We then explain that any periodic signal can be expressed as a sum of sine waves. This is the notion of Fourier series. After an illustration (denoising of a corrupted signal) which introduces a notion of filtering in the frequency domain, we show how the Fourier representation can be extended to aperiodic signals.

- Section **??**
- Section **??**
- Section **??**
- Section **??**
- Section **??**
- Section **??**

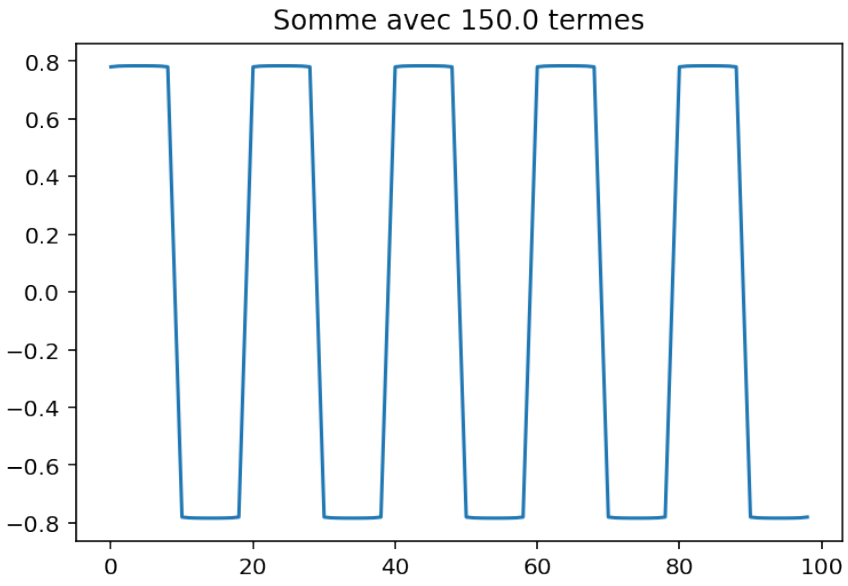
2.1 Simple examples

Read the script below, execute (CTRL-Enter), experiment with the parameters.

```
N = 100
L = 20
s = np.zeros(N - 1)

for k in np.arange(1, 300, 2):
    s = s + 1 / float(k) * sin(2 * pi * k / L * np.arange(1, N, 1))
plt.plot(s)
plt.title("Somme avec " + str((k - 1) / 2 + 1) + " termes")
```

```
Text(0.5, 1.0, 'Somme avec 150.0 termes')
```

The next example is more involved in that it sums sin a cos of different frequencies and with different amplitudes. We also add widgets (sliders) which enable to interact more easily with the program.




```
@out.capture(clear_output=True, wait=True)
def sfou_exp(Km):
    #clear_output(wait=True)
    Kmax = int(Km)
    L = 400
    N = 1000
    k = 0
    s = np.zeros(N - 1)
    #plt.clf()
    for k in np.arange(1, Kmax):
        ak = 0
        bk = 1.0 / k if (k % 2) == 1 else 0  # k odd

        # ak=0 #if (k % 2) == 1 else  -2.0/(pi*k**2)
        # bk=-1.0/k if (k % 2) == 1 else 1.0/k #

        s = s + ak * cos(2 * pi * k / L * np.arange(1, N, 1)) + bk * sin(
            2 * pi * k / L * np.arange(1, N, 1))
    ax = plt.axes(xlim=(0, N), ylim=(-2, 2))
    ax.plot(s)
    plt.title("Sum with {} terms".format(k + 1))
    plt.show()


### ------------------------------------------------------
```

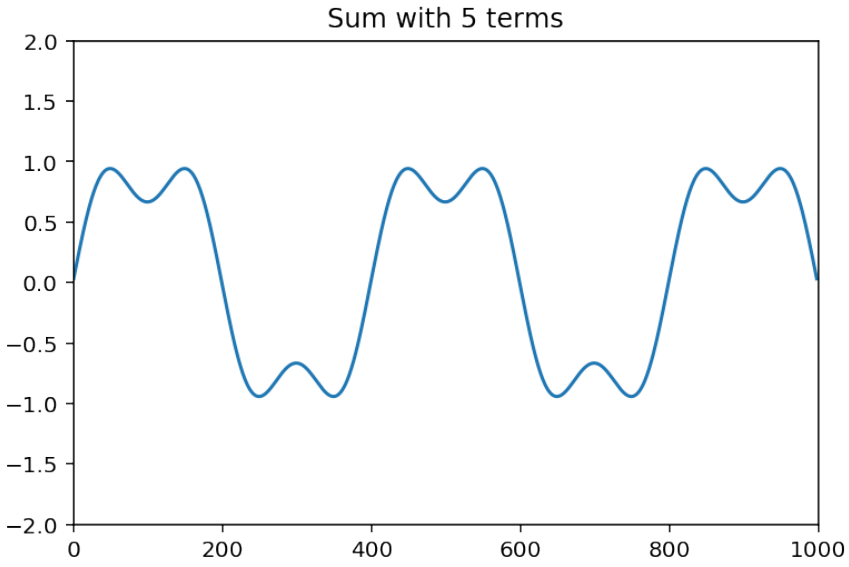
```
    ---------------------------------------------------------------------------

    NameError                                 Traceback (most recent call last)

    <ipython-input-3-67dbb1757edc> in <module>()
----> 1 @out.capture(clear_output=True, wait=True)
```

```
2 def sfou_exp(Km):
3     #clear_output(wait=True)
4     Kmax = int(Km)
5     L = 400
```

```
NameError: name 'out' is not defined
```

```
out = widgets.Output()

fig = plt.figure()
ax = plt.axes(xlim=(0, 100), ylim=(-2, 2))

# ---- Widgets ----------------------
# slider=widgets.FloatSlider(max=100,min=0,step=1,value=1)
slide = widgets.IntSlider(max=100, min=0, step=1, value=5)
val = widgets.IntText(value='1')


#------ Callbacks des widgets ------------
@out.capture(clear_output=True, wait=True)
def sfou_exp(Km):
    #clear_output(wait=True)
    Kmax = int(Km)
    L = 400
    N = 1000
    k = 0
    s = np.zeros(N - 1)
    #plt.clf()
    for k in np.arange(1, Kmax):
        ak = 0
        bk = 1.0 / k if (k % 2) == 1 else 0  # k odd

        # ak=0 #if (k % 2) == 1 else -2.0/(pi*k**2)
        # bk=-1.0/k if (k % 2) == 1 else 1.0/k #

        s = s + ak * cos(2 * pi * k / L * np.arange(1, N, 1)) + bk * sin(
            2 * pi * k / L * np.arange(1, N, 1))
    ax = plt.axes(xlim=(0, N), ylim=(-2, 2))
    ax.plot(s)
    plt.title("Sum with {} terms".format(k + 1))
    plt.show()


### ------------------------------------------------


#@out.capture(clear_output=True, wait=True)
def sfou1_Km(param):
    Km = param['new']
    val.value = str(Km)
    sfou_exp(Km)


#@out.capture(clear_output=True, wait=True)
def sfou2_Km(param):
```

```
    Km = param.new
    slide.value = Km
    #sfou_exp(Km.value)

# ---- Display ------------------
#display(slide)
#display(val)


slide.observe(sfou1_Km, names=['value'])
sfou_exp(5)
#val.observe(sfou2_Km,names='value')
display(widgets.VBox([slide, out]))
```



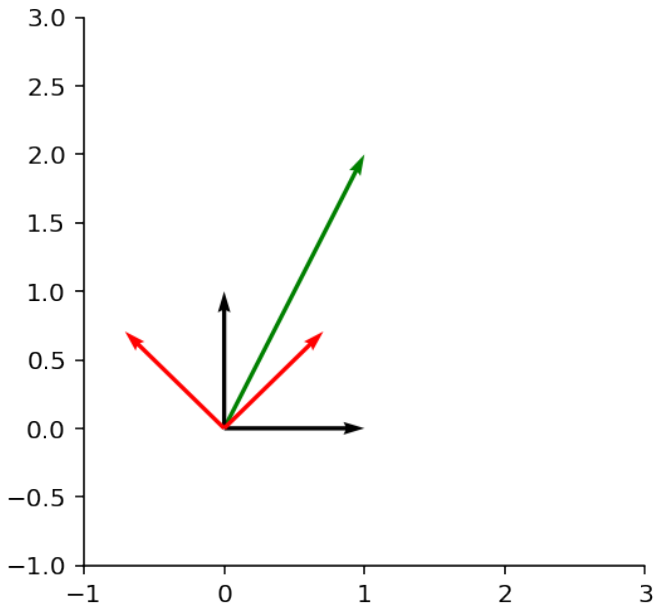

```
Widget Javascript not detected.  It may not be installed or enabled properly.
```

2.1.1 Decomposition on basis - scalar producs

We recall here that any vector can be expressed on a orthonormal basis, and that the coordinates are the scalar product of the vector with the basis vectors.

```
z = array([1, 2])
u = array([0, 1])
v = array([1, 0])
u1 = array([1, 1]) / sqrt(2)
v1 = array([-1, 1]) / sqrt(2)

f, ax = subplots(1, 1, figsize=(4, 4))
ax.set_xlim([-1, 3])
ax.set_ylim([-1, 3])
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
#ax.spines['bottom'].set_position('center')
```

```
ax.quiver(
    0, 0, z[0], z[1], angles='xy', scale_units='xy', scale=1, color='green')
ax.quiver(
    0, 0, u[0], u[1], angles='xy', scale_units='xy', scale=1, color='black')
ax.quiver(
    0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='black')
ax.quiver(
    0, 0, u1[0], u1[1], angles='xy', scale_units='xy', scale=1, color='red')
ax.quiver(
    0, 0, v1[0], v1[1], angles='xy', scale_units='xy', scale=1, color='red')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
```



From a coordinate system to another: Take a vector (in green in the illustration). Its coordinates in the system (u,v) are [1,2]. In order to obtain the coordinates in the new system (O,u_1,v_1), we have to project the vector on u_1 and u_2. This is done by the scalar products:

```
x = z.dot(u1)
y = z.dot(v1)
print('New coordinates: ', x, y)
```

```
New coordinates:  2.1213203435596424 0.7071067811865475
```

2.2 Decomposition of periodic functions – Fourier series

This idea can be extended to (periodic) functions. Consider the set of all even periodic functions, with a given period, say L. The cosine wave functions of all the multiple or the *fundamental* frequency $1/L$ constitute a basis of even periodic functions with period T. Let us check that these functions are normed and ortogonal with each other.

```
L = 200
k = 8
l = 3
sk = sqrt(2 / L) * cos(2 * pi / L * k * np.arange(0, L))
sl = sqrt(2 / L) * cos(2 * pi / L * l * np.arange(0, L))
```

```
sl.dot(sl)
```

```
1.0000000000000004
```

Except in the case $l = 0$ where a factor 2 entails

```
l = 0
sl = sqrt(2 / L) * cos(2 * pi / L * l * np.arange(0, L))
sl.dot(sl)
```

```
1.9999999999999998
```

Therefore, the decomposition of any even periodic function $x(n)$ with period L on the basis of cosines expresses as

$$x(n) = \sqrt{\frac{2}{L}} \left(\frac{a_0}{2} + \sum_{k=1}^{L-1} a_k \cos(2\pi k/Ln) \right)$$

with

$$a_k = \sqrt{\frac{2}{L}} \sum_{n \in [L]} x(n) \cos(2\pi k/Ln).$$

Regrouping the factors, the series can also be expressed as

$$x_{\text{even}}(n) = \left(\frac{a_0}{2} + \sum_{k=1}^{L-1} a_k \cos(2\pi k/Ln) \right)$$

with

$$a_k = \frac{2}{L} \sum_{n \in [L]} x(n) \cos(2\pi k/Ln),$$

where the notation $n \in [L]$ indicates that the sum has to be done on any length-L interval. The very same reasoning can be done for odd functions, which introduces a decomposition into sine waves:

$$x_{\text{odd}}(n) = \sum_{k=0}^{L-1} b_k \sin(2\pi k/Ln)$$

with

$$b_k = \frac{2}{L} \sum_{n \in [L]} x(n) \sin(2\pi k/Ln),$$

Since any function can be decomposed into an odd + even part

$$x(n) = x_{\text{even}}(n) + x_{\text{odd}}(n) = \frac{x(n) + x(-n)}{2} + \frac{x(n) - x(-n)}{2},$$

we have the sum of the decompositions:

$$\boxed{x(n) = \frac{a_0}{2} + \sum_{k=1}^{L-1} a_k \cos(2\pi k/Ln) + \sum_{k=1}^{+\infty} b_k \sin(2\pi k/Ln)}$$

with

$$\boxed{\begin{cases} a_k = \frac{2}{L}\sum_{n\in[L]} x(n)\cos(2\pi k/Ln), \\ b_k = \frac{2}{L}\sum_{n\in[L]} x(n)\sin(2\pi k/Ln), \end{cases}}$$

This is the definition of the Fourier series, and this is no more compicated than that... A remaining question is the question of convergence. That is, does the series converge to the true function? The short answer is Yes: the equality in the series expansion is a true equality, not an approximation. This is a bit out of scope for this course, but you may have a look at this article.

There of course exists a continuous version, valid for time-continuous dignals.

2.3 Complex Fourier series

2.3.1 Introduction

Another series expansion can be defined for complex valued signals. In such case, the trigonometric functions will be replaced by complex exponentials $\exp(j2\pi k/Ln)$. Let us check that they indeed form a basis of signals:

```
L = 200
k = 8
l = 3
sk = sqrt(1 / L) * exp(1j * 2 * pi / L * k * np.arange(0, L))
sl = sqrt(1 / L) * exp(1j * 2 * pi / L * l * np.arange(0, L))
print("scalar product between sk and sl: ", np.vdot(sk, sl))
print("scalar product between sk and sk (i.e. norm of sk): ", np.vdot(sk, sk
    ))
```

```
scalar product between sk and sl:  (-1.9932252838852267e-17+5.886840063482993e-17j)
scalar product between sk and sk (i.e. norm of sk):  (1+0j)
```

It is thus possible to decompose a signal as follows:

$$\boxed{\begin{aligned} x(n) &= \sum_{k=0}^{L-1} c_k e^{j2\pi\frac{kn}{L}} \\ \text{with } c_k &= \frac{1}{L}\sum_{n\in[L]} x(n) e^{-j2\pi\frac{kn}{L}} \end{aligned}}$$

where c_k is the dot product between $x(n)$ and $\exp(j2\pi k/Ln)$, i.e. the 'coordinate' of x with respect to the 'vector' $\exp(j2\pi k/Ln)$. This is nothing but the definition of the `complex Fourier series`.

Exercise – Show that c_k is periodic with period L; i.e. $c_k = c_{k+L}$.

Since c_k is periodic in k of period L, we see that in term or the *"normalized frequency"* k/L, it is periodic with period 1.

Relation of the complex Fourier Series with the standard Fourier Series

It is easy to find a relation between this complex Fourier series and the classical Fourier series. The series can be rewritten as

$$x(n) = c_0 + \sum_{k=1}^{+\infty} c_k e^{j2\pi k/Ln} + c_{-k} e^{-j2\pi k/Ln}.$$

By using the Euler formulas, developping and rearranging, we get

$$x(n) = c_0 + \sum_{k=1}^{+\infty} \mathscr{R}\{c_k + c_{-k}\}\cos(2\pi k/Ln) + \mathscr{I}\{c_{-k} - c_k\}\sin(2\pi k/Ln) \tag{2.1}$$

$$+ j\left(\mathscr{R}\{c_k - c_{-k}\}\sin(2\pi k/Ln) + \mathscr{I}\{c_k + c_{-k}\}\cos(2\pi k/Ln)\right). \tag{2.2}$$

Suppose that $x(n)$ is real valued. Then by direct identification, we have

$$\begin{cases} a_k = \mathscr{R}\{c_k + c_{-k}\} \\ b_k = \mathscr{I}\{c_{-k} - c_k\} \end{cases}$$

and, by the cancellation of the imaginary part, the following symmetry relationships for real signals:

$$\begin{cases} \mathscr{R}\{c_k\} = \mathscr{R}\{c_{-k}\} \\ \mathscr{I}\{c_k\} = -\mathscr{I}\{c_{-k}\}. \end{cases}$$

This symmetry is called 'Hermitian symmetry'.

2.3.2 Computer experiment

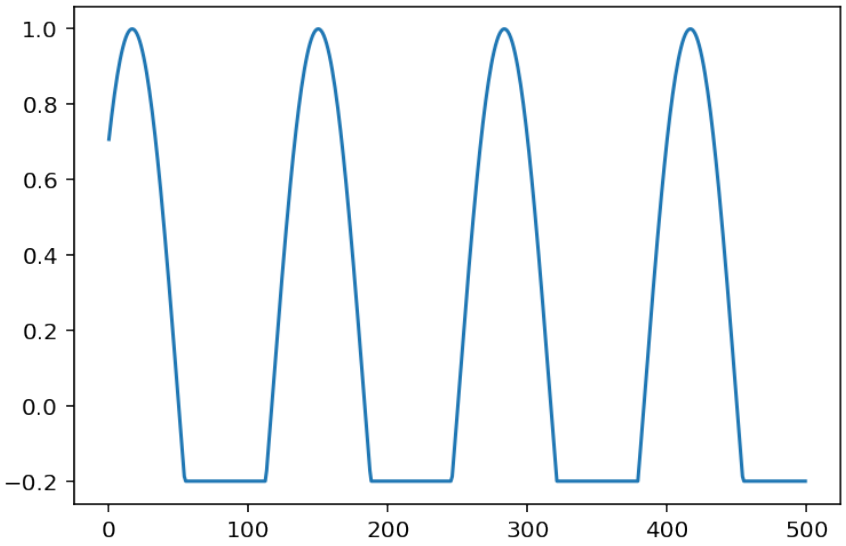
Experiment. Given a signal, computes its decomposition and then reconstruct the signal from its individual components.

```
%matplotlib inline
L = 400
N = 500
t = np.arange(N)
s = sin(2 * pi * 3 * t / L + pi / 4)
x = [ss if ss > -0.2 else -0.2 for ss in s]
plt.plot(t, x)
```

```
[<matplotlib.lines.Line2D at 0x7f17d7e863c8>]
```

A function for computing the Fourier series coefficients

```
# compute the coeffs ck
def coeffck(x, L, k):
    assert np.size(x) == L, "input must be of length L"
    karray = []
    res = []
    if isinstance(k, int):
        karray.append(k)
    else:
        karray = np.array(k)
```



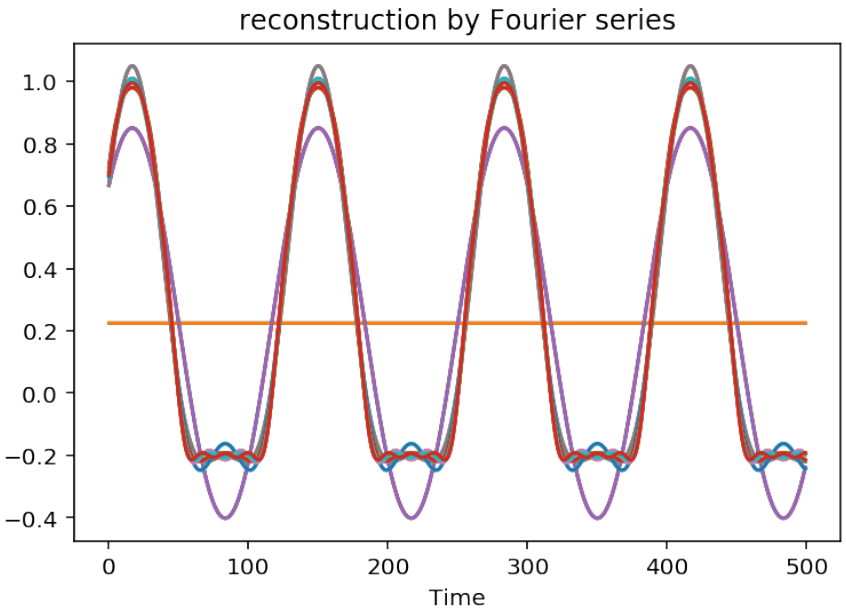

```
    for k in karray:
        res.append(np.vdot(exp(1j * 2 * pi / L * k * np.arange(0, L)), x))
    return 1 / L * np.array(res)


#test: coeffck(x[0:L],L,[-12,1,7])
# --> array([   1.51702135e-02 +4.60742555e-17j,
#      -1.31708229e-05 -1.31708229e-05j,    1.37224241e-05 -1.37224241e-05j])
```

Now let us compute the coeffs for actual signal

```
# compute the coeffs for actual signal
c1 = coeffck(x[0:L], L, np.arange(0, 100))
c2 = coeffck(x[0:L], L, np.arange(0, -100, -1))
s = c1[0] * np.ones((N))
for k in np.arange(1, 25):
    s = s + c1[k] * exp(1j * 2 * pi / L * k * np.arange(0, N)) + c2[k] * exp
        (
        -1j * 2 * pi / L * k * np.arange(0, N))
    plt.plot(t, np.real(s))
plt.title("reconstruction by Fourier series")
plt.xlabel("Time")
```

```
Text(0.5, 0, 'Time')
```

```
plt.figure()
kk = np.arange(-50, 50)
c = coeffck(x[0:L], L, kk)
plt.stem(kk, np.abs(c))
plt.title("Fourier series coefficients (modulus)")
plt.xlabel("k")
msg = """In the frequency representation, the x axis corresponds to the
    frequencies k/L
of the complex exponentials.
```

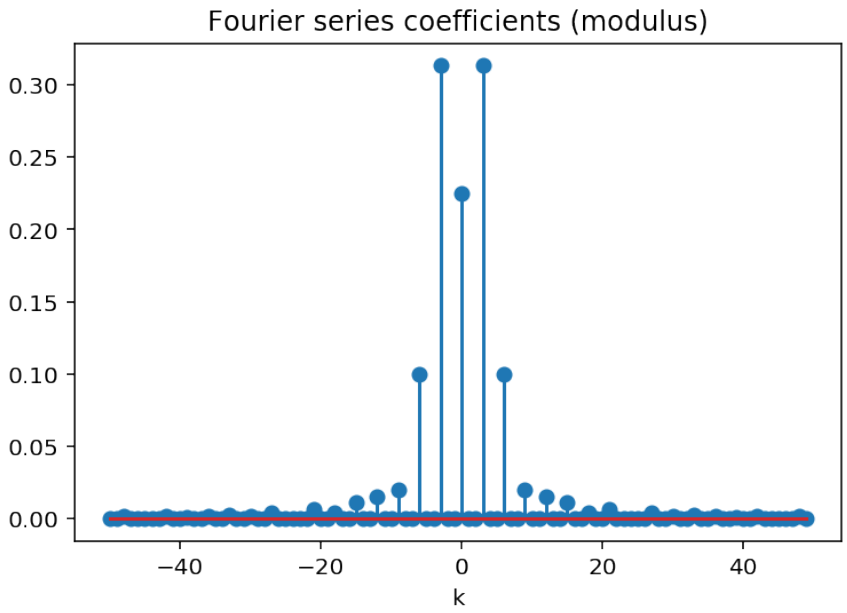



```
Therefore, if a signal is periodic of period M, the corresponding
    fundamental frequency
is 1/M. This frequency then appears at index ko=L/M (if this ratio is an
    integer).
Harmonics will appear at multiples of ko."""
print(msg)
```

```
In the frequency representation, the x axis corresponds to the frequencies k/L
of the complex exponentials.
Therefore, if a signal is periodic of period M, the corresponding fundamental frequ
is 1/M. This frequency then appears at index ko=L/M (if this ratio is an integer).
Harmonics will appear at multiples of ko.
```

A pulse train corrupts our original signal

```
L = 400


# define a pulse train which will corrupt our original signal
def sign(x):
    if isinstance(x, (int, float)):
        return 1 if x >= 0 else -1
    else:
        return np.array([1 if u >= 0 else -1 for u in x])


#test: sign([2, 1, -0.2, 0])


def repeat(x, n):
    if isinstance(x, (np.ndarray, list, int, float)):
        return np.array([list(x) * n]).flatten()
```




```
    else:
        raise ('input must be an array,list,or float/int')


#t=np.arange(N)
#sig=sign(sin(2*pi*10*t/L))
```

```
rect = np.concatenate((np.ones(20), -np.ones(20)))
#[1,1,1,1,1,-1,-1,-1,-1,-1]

sig = repeat(rect, 15)
sig = sig[0:N]
plt.plot(sig)
plt.ylim({-1.1, 1.1})
```

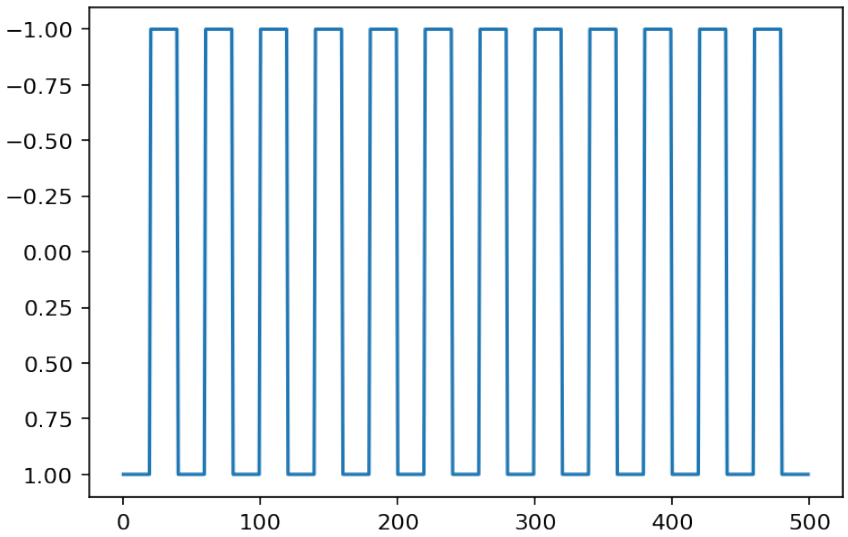
```
(1.1, -1.1)
```

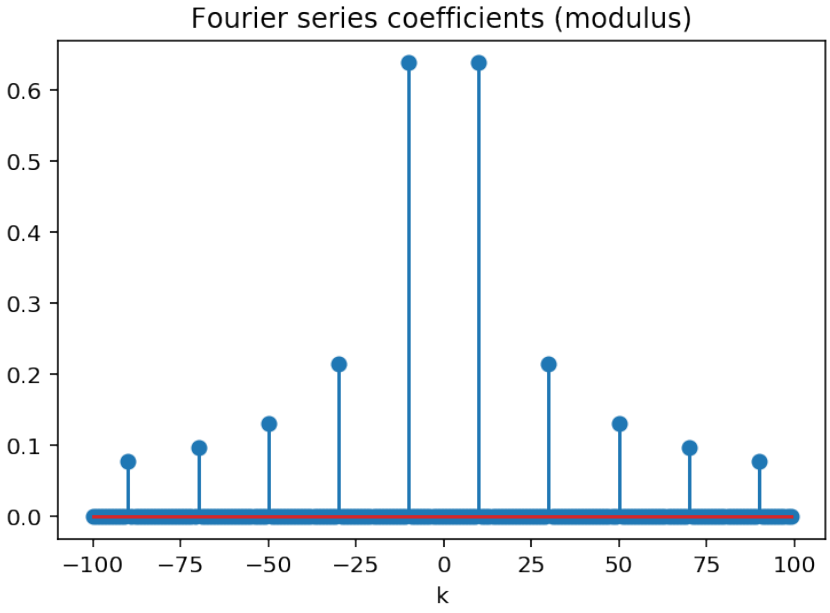
Compute and represent the Fourier coeffs of the pulse train

```
kk = np.arange(-100, 100)
c = coeffck(sig[0:L], L, kk)
plt.figure()
plt.stem(kk, np.abs(c))
plt.title("Fourier series coefficients (modulus)")
plt.xlabel("k")
```

```
Text(0.5, 0, 'k')
```

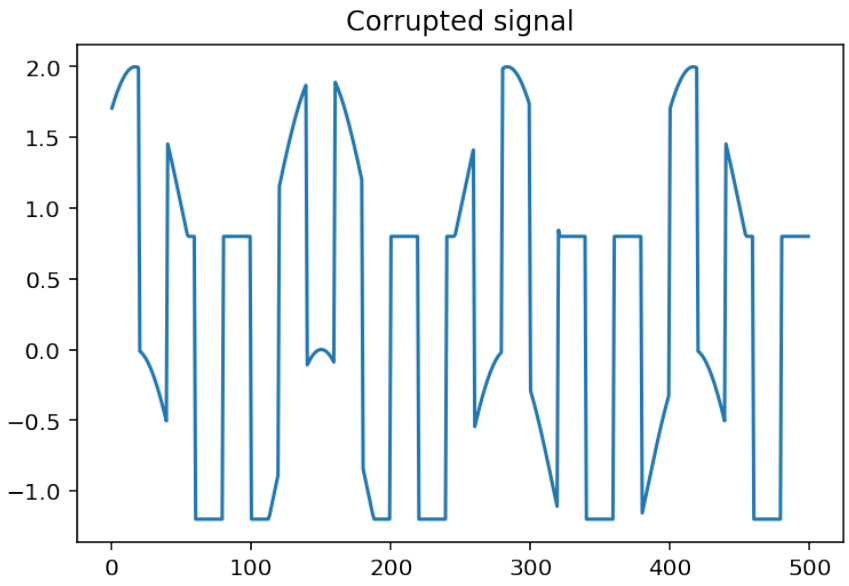
The fundamental frequency of the pulse train is 1 over the length of the pulse, that is 1/40 here. Since The Fourier series is computed on a length L=400, the harmonics appear every 10 samples (ie at indexes k multiples of 10).


−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0
100
200
300
400
500


Fourier series coefficients (modulus)
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−100
−75
−50
−25
0
25
50
75
100
k

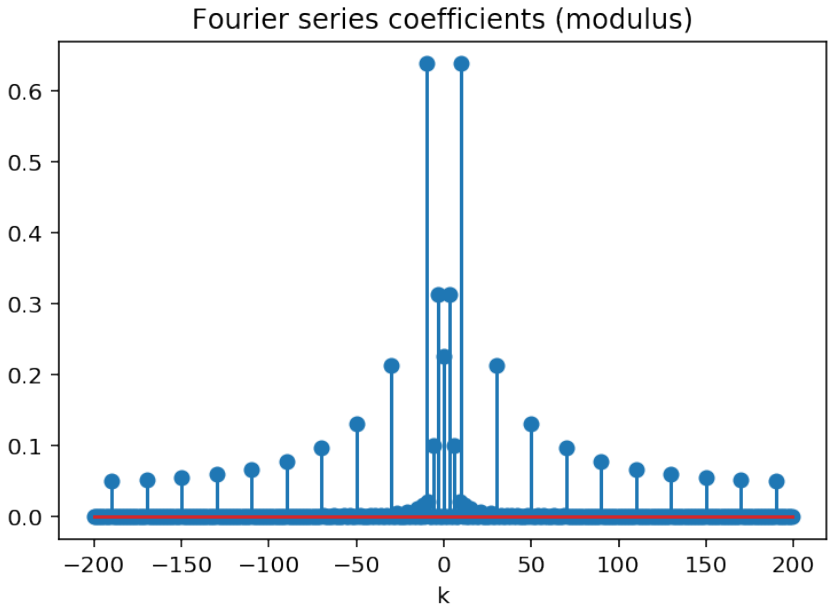
```
z = x + 1 * sig
plt.plot(z)
plt.title("Corrupted signal")

kk = np.arange(-200, 200)
cz = coeffck(z[0:L], L, kk)
plt.figure()
plt.stem(kk, np.abs(cz))
plt.title("Fourier series coefficients (modulus)")
plt.xlabel("k")
```

```
Text(0.5, 0, 'k')
```

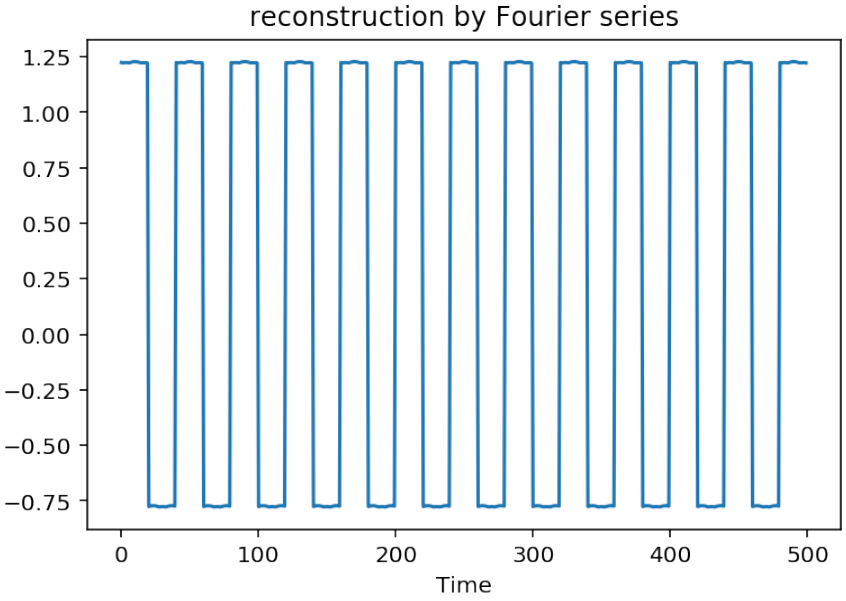


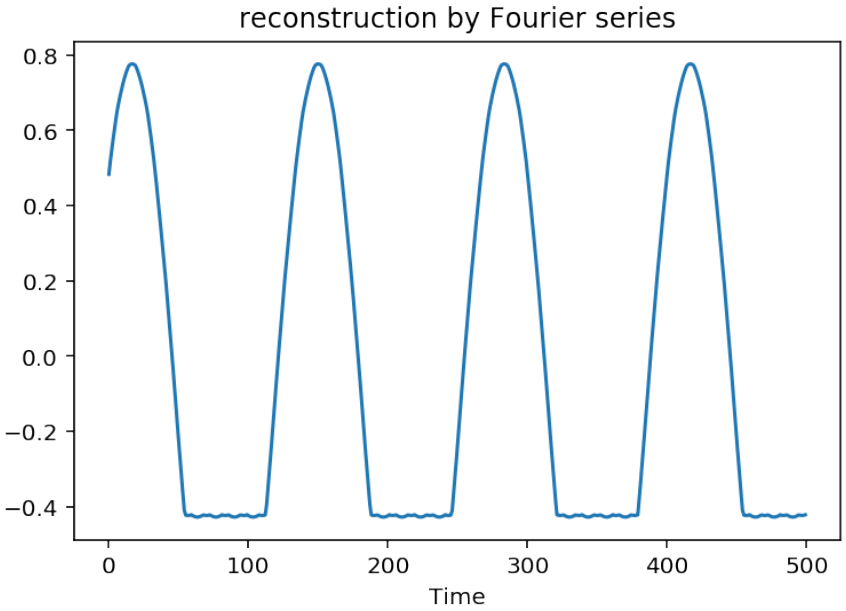
Now, we try to kill all the frequencies harmonics of 10 (the fundamental frequency of the pulse train), and reconstruct the resulting signal...

```
# kill frequencies harmonics of 10 (the fundamental frequency of the pulse
    train)
# and reconstruct the resulting signal

s = np.zeros(N)
kmin = np.min(kk)
for k in kk:
    if not k % 10:  #true if k is multiple of 10
        s = s + cz[k + kmin] * exp(1j * 2 * pi / L * k * np.arange(0, N))
plt.figure()
plt.plot(t, np.real(s))
plt.title("reconstruction by Fourier series")
plt.xlabel("Time")

plt.figure()
plt.plot(t, z - np.real(s))
plt.title("reconstruction by Fourier series")
plt.xlabel("Time")
```




```
Text(0.5, 0, 'Time')
```


reconstruction by Fourier series
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
100
200
300
400
500
Time


reconstruction by Fourier series
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
100
200
300
400
500
Time

3

From Fourier Series to Fourier transforms

In this section, we go from the Fourier series to the Fourier transform for discrete signal. So doing, we also introduce the notion of Discrete Fourier Transform that we will study in more details later. For now, we focus on the representations in the frequency domain, detail and experiment with some examples.

3.1 Introduction and definitions

Suppose that we only have an observation of length N. So no periodic signal, but a signal of size N. We do not know if there were data before the first sample, and we do not know if there were data after sample N. What to do? Facing to such situation, we can still - imagine that the data are periodic outside of the observation interval, with a period N. Then the formulas for the Fourier series **are** valid, for n in the observation interval. Actually there is no problem with that. The resulting transformation is called the *Discrete Fourier Transform* . The corresponding formulas are

$$x(n) = \sum_{k=0}^{N-1} X(k) e^{j2\pi \frac{kn}{N}}$$
$$\text{with } X(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{kn}{N}}$$

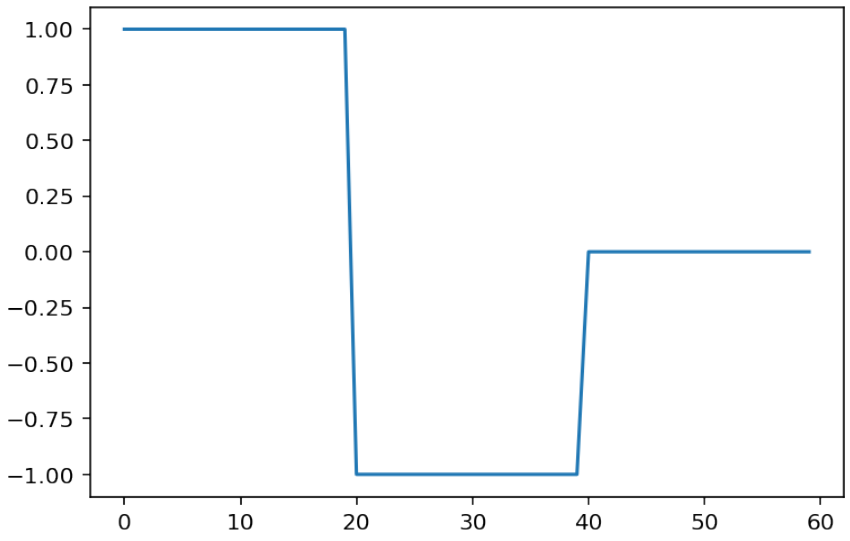
- we may also consider that there is nothing –that is zeros, outside of the observation interval. In such condition, we can still imagine that we have a periodic signal, but with an infinite period. Since the separation of two harmonics in the Fourier series is Δf=1/period, we see that $\Delta f \to 0$. Then the Fourier representation becomes continuous. This is illustrated below.

```
# compute the coeffs ck
def coeffck(x, L, k):
    assert np.size(x) == L, "input must be of length L"
    karray = []
    res = []
    if isinstance(k, int):
        karray.append(k)
    else:
        karray = np.array(k)

    for k in karray:
```

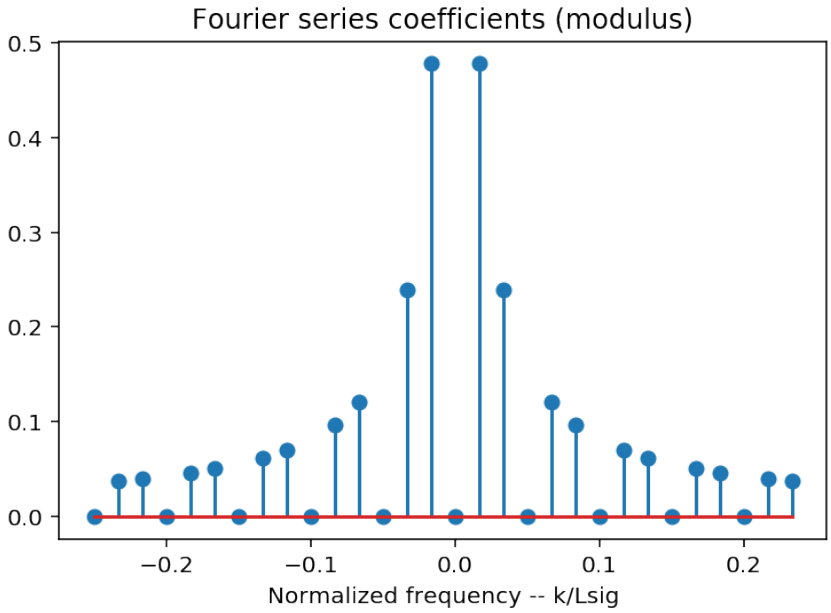
```
        res.append(np.vdot(exp(1j * 2 * pi / L * k * np.arange(0, L)), x))
    return 1 / L * np.array(res)


#test: coeffck(x[0:L],L,[-12,1,7])
# --> array([  1.51702135e-02 +4.60742555e-17j,
#       -1.31708229e-05 -1.31708229e-05j,   1.37224241e-05 -1.37224241e-05j])
```

```
Lpad = 20  # then 200, then 2000
# define a rectangular pulse
rect = np.concatenate((np.ones(20), -np.ones(20)))
# Add zeros after:
rect_zeropadded = np.concatenate((rect, np.zeros(Lpad)))
sig = rect_zeropadded
plt.plot(sig)
# compute the Fourier series for |k/Lsig|<1/4
Lsig = np.size(sig)
fmax = int(Lsig / 4)
kk = np.arange(-fmax, fmax)
c = coeffck(sig[0:Lsig], Lsig, kk)
# plot it
plt.figure()
plt.stem(kk / Lsig, np.abs(c))
plt.title("Fourier series coefficients (modulus)")
plt.xlabel("Normalized frequency -- k/Lsig")
```

```
Text(0.5, 0, 'Normalized frequency -- k/Lsig')
```



Hence we obtain a formula where the discrete sum for reconstructing the time-series $x(n)$ becomes a



continuous sum, since f is now continuous:

$$x(n) = \sum_{k=0}^{N-1} c_k e^{j2\pi\frac{kn}{N}} = \sum_{k/N=0}^{1-1/N} NX(k)e^{j2\pi\frac{kn}{N}}\frac{1}{N}$$
$$\rightarrow x(n) = \int_0^1 X(f)e^{j2\pi fn}\mathrm{d}f$$

Finally, we end with what is called the **Discrete-time Fourier transform** :

$$\boxed{\begin{aligned} x(n) &= \int_0^1 X(f)e^{j2\pi fn}\mathrm{d}f \\ \text{with } X(f) &= \sum_{n=-\infty}^{\infty} x(n)e^{-j2\pi fn} \end{aligned}}$$

Even before exploring the numerous properties of the Fourier transform, it is important to stress that The Fourier transform of a discrete signal is periodic with period one.

Check it as an exercise! Begin with the formula for $X(f)$ an compute $X(f+1)$. use the fact that n is an integer and that $\exp(j2\pi n) = 1$.

3.2 Examples

Exercise 2. .

- *Compute the* Fourier transform *of a rectangular window given on N points. The result is called a (discrete) cardinal sine (or sometimes Dirichlet kernel). Sketch a plot, and study the behaviour of this function with N.*
- *Experiment numerically... See below the provided functions.*

- *Compute the Fourier transform of a sine wave* $\sin(2\pi f_0 n)$ *given on N points.*
- *Examine what happens when the N and* f_0 *vary.*

3.2.1 The Fourier transform of a rectangular window

The derivation of the formula will be done in class. Let us see the experimental part.

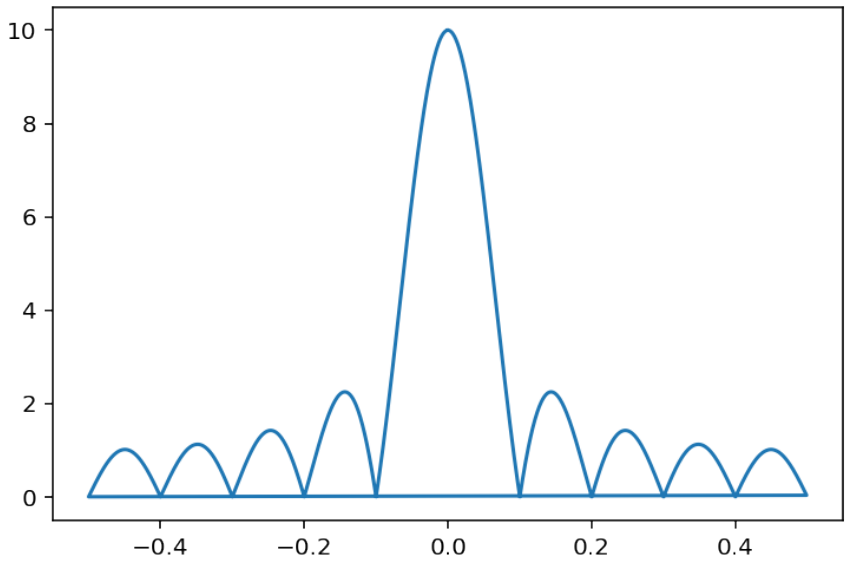
For the numerical experiments, import the fft (Fast Fourier Transform) function,

```
from numpy.fft import fft, ifft
```

define a sine wave, complute and plot its Fourier transform. As the FFT is actually an implementation of a discrete Fourier transform, we will have an approximation of the true Fourier transform by using zero-padding (check that a parameter in the fft enables to do this zero-padding).

```
from numpy.fft import fft, ifft

#Define a rectangular window, of length L
#on N points, zeropad to NN=1000
# take eg L=100, N=500
NN = 1000
L = 10  # 10, then 6, then 20, then 50, then 100...
r = np.ones(L)
Rf = fft(r, NN)
f = fftfreq(NN)
plt.plot(f, np.abs(Rf))
```

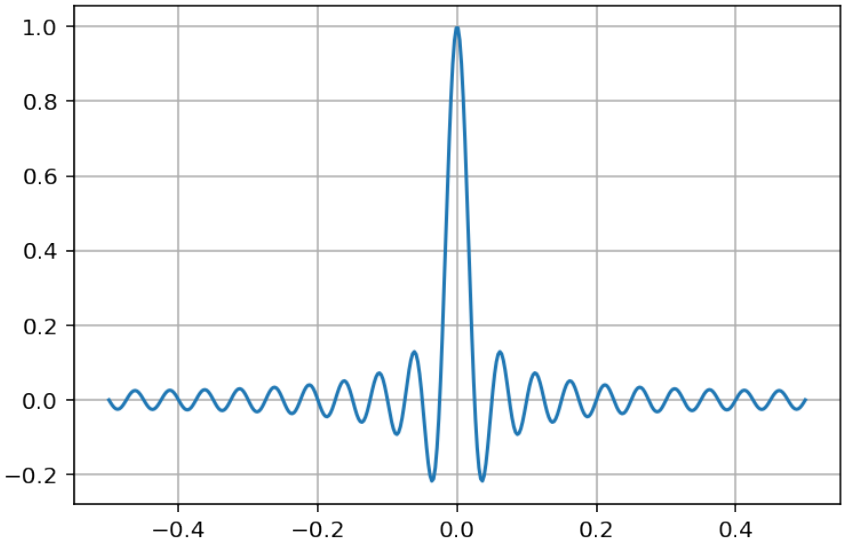
```
[<matplotlib.lines.Line2D at 0x7f5d27641d68>]
```



It remain to compare this to a discrete cardinal sinus. First we define a function and then compare the results.

```
def dsinc(x, L):
    if isinstance(x, (int, float)): x = [x]
    x = np.array(x)
    out = np.ones(np.shape(x))
    I = np.where(x != 0)
    out[I] = np.sin(x[I]) / (L * np.sin(x[I] / L))
    return out
```

```
N = 1000
L = 40
f = np.linspace(-0.5, 0.5, 400)
plt.plot(f, dsinc(pi * L * f, L))
plt.grid(b=True)
```

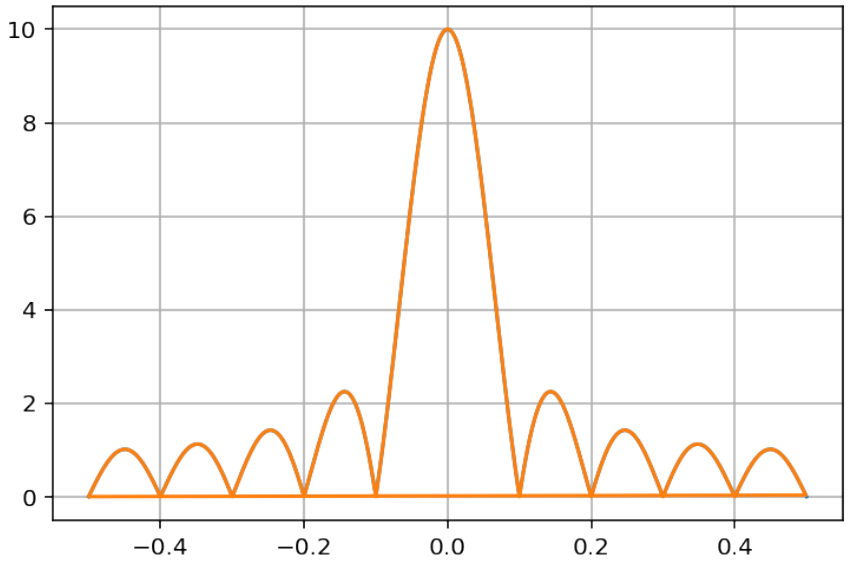


Comparison of the Fourier transform of a rectangle and a cardinal sine:

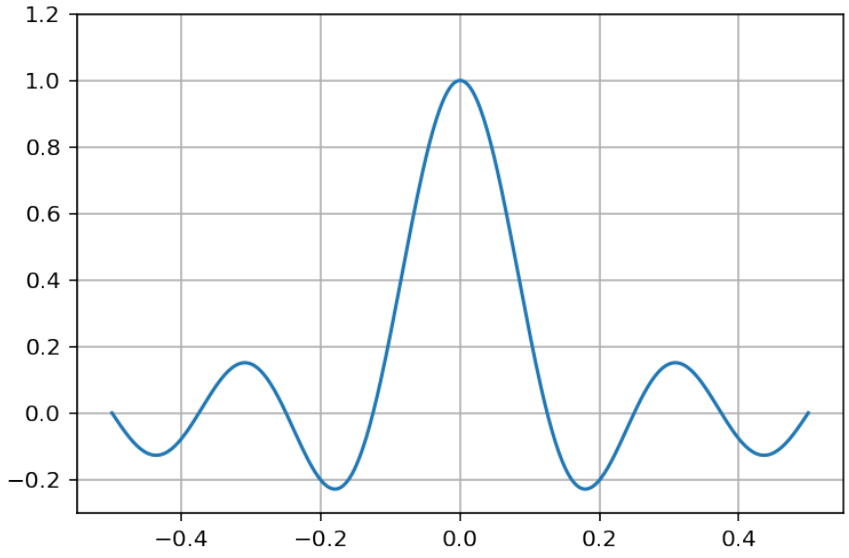
```
NN = 1000
L = 10  # 10, then 6, then 20, then 50, then 100...
r = np.ones(L)
Rf = fft(r, NN)

N = 1000
f = np.linspace(-0.5, 0.5, 400)
plt.plot(f, L * np.abs(dsinc(pi * L * f, L)))
f = fftfreq(NN)
plt.plot(f, np.abs(Rf))
plt.grid(b=True)
```

Interactive versions...

```
# using %matplotlib use a backend that allows external figures
# using %matplotlib inline plots the results in the notebook
%matplotlib inline
slider = widgets.FloatSlider(min=0.1, max=100, step=0.1, value=8)
display(slider)


#——— Callbacks des widgets ——————————
```




```
def pltsinc(change):
    L = change['new']
    plt.clf()
    clear_output(wait=True)
    #val.value=str(f)
    f = np.linspace(-0.5, 0.5, 400)
    plt.plot(f, dsinc(pi * L * f, L))
    plt.ylim([-0.3, 1.2])
    plt.grid(b=True)


pltsinc({'new': 8})
slider.observe(pltsinc, names=['value'])
```

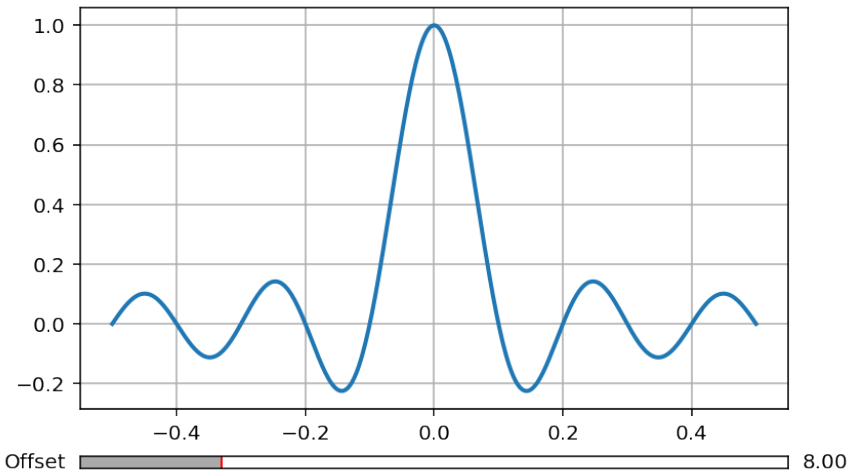


This is an example with matplotlib widgets interactivity, (instead of html widgets). The docs can be found here

```
%matplotlib
from matplotlib.widgets import Slider

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2, left=0.1)

slider_ax = plt.axes([0.1, 0.1, 0.8, 0.02])
slider = Slider(slider_ax, "Offset", 0, 40, valinit=8, color='#AAAAAA')
L = 10
f = np.linspace(-0.5, 0.5, 400)

line, = ax.plot(f, dsinc(pi * L * f, L), lw=2)
#line2, = ax.plot(f,sinc(pi*L*f), lw=2)
#line2 is in order to compare with the "true" sinc
ax.grid(b=True)


def on_change(L):
    line.set_ydata(dsinc(pi * L * f, L))


#    line2.set_ydata(sinc(pi*L*f))

slider.on_changed(on_change)
```

```
Using matplotlib backend: TkAgg
```

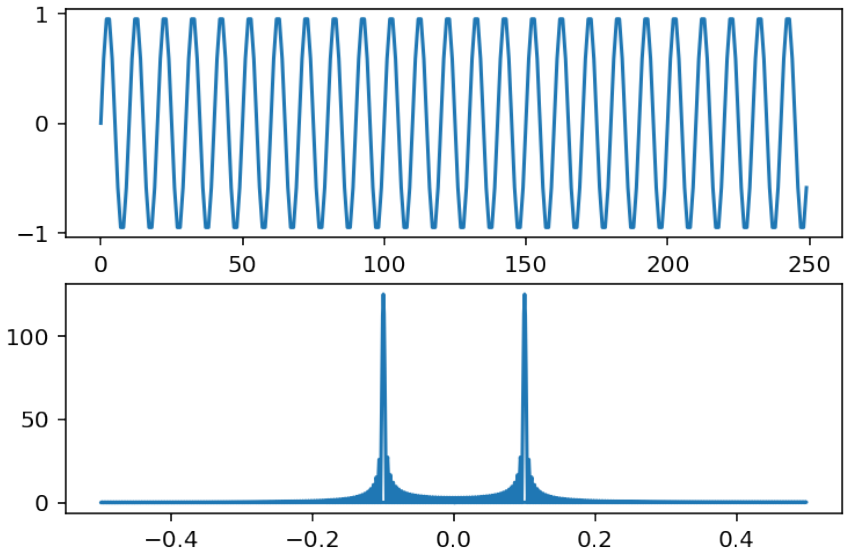
```
0
```



3.2.2 Fourier transform of a sine wave

Again, the derivation will be done in class.

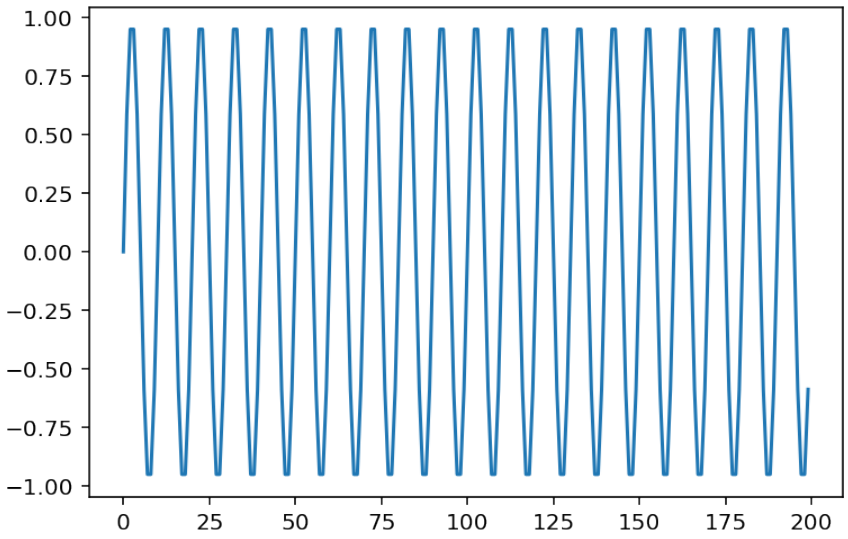
```
%matplotlib inline

from numpy.fft import fft, ifft
N = 250
f0 = 0.1
NN = 1000
fig, ax = plt.subplots(2, 1)


def plot_sin_and_transform(N, f0, ax):
    t = np.arange(N)
    s = np.sin(2 * pi * f0 * t)
    Sf = fft(s, NN)
    ax[0].plot(t, s)
    f = np.fft.fftfreq(NN)
    ax[1].plot(f, np.abs(Sf))


plot_sin_and_transform(N, f0, ax)
```



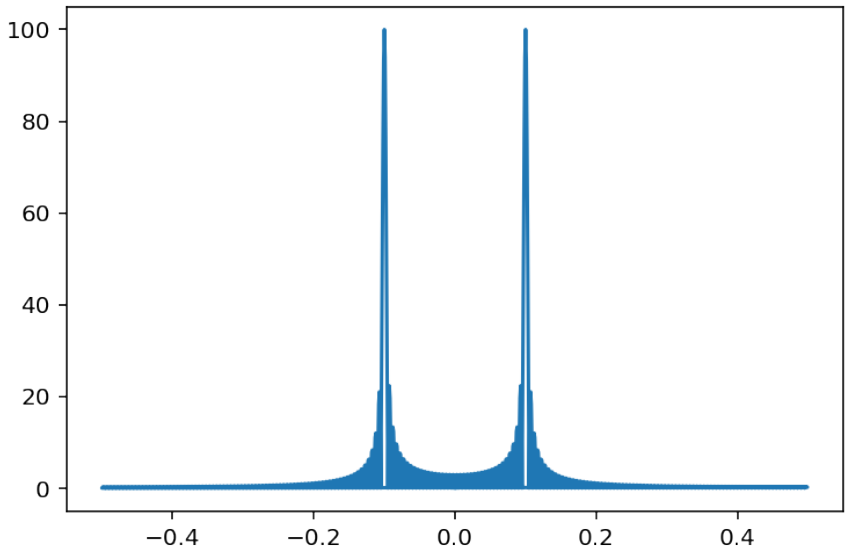
Interactive versions

```
# using %matplotlib use a backend that allows external figures
# using %matplotlib inline plots the results in the notebook
%matplotlib inline
sliderN = widgets.IntSlider(
    description="N", min=1, max=1000, step=1, value=200)
sliderf0 = widgets.FloatSlider(
    description="f0", min=0, max=0.5, step=0.01, value=0.1)
c1 = widgets.Checkbox(description="Display time signal", value=True)
c2 = widgets.Checkbox(description="Display frequency signal", value=True)

#display(sliderN)
#display(sliderf0)
N = 500
f0 = 0.1
t = np.arange(N)
```

```
s = np.sin(2 * pi * f0 * t)
Sf = fft(s, NN)
f = np.fft.fftfreq(NN)

out = widgets.Output()


#------ Callbacks des widgets ----------------
@out.capture(clear_output=True, wait=True)
def pltsin(dummy):
    #clear_output(wait=True)
    N = sliderN.value
    f0 = sliderf0.value
    t = np.arange(N)
    s = np.sin(2 * pi * f0 * t)
    Sf = fft(s, NN)
    f = np.fft.fftfreq(NN)
    if c1.value:
        plt.figure(1)
        plt.clf()
        plt.plot(t, s)
    if c2.value:
        plt.figure(2)
        plt.clf()
        plt.plot(f, np.abs(Sf))
    plt.show()


pltsin(8)
sliderN.observe(pltsin, names='value')
sliderf0.observe(pltsin, names='value')
c1.observe(pltsin, names='value')
c2.observe(pltsin, names='value')
display(widgets.VBox([sliderN, sliderf0, c1, c2, out]))
```

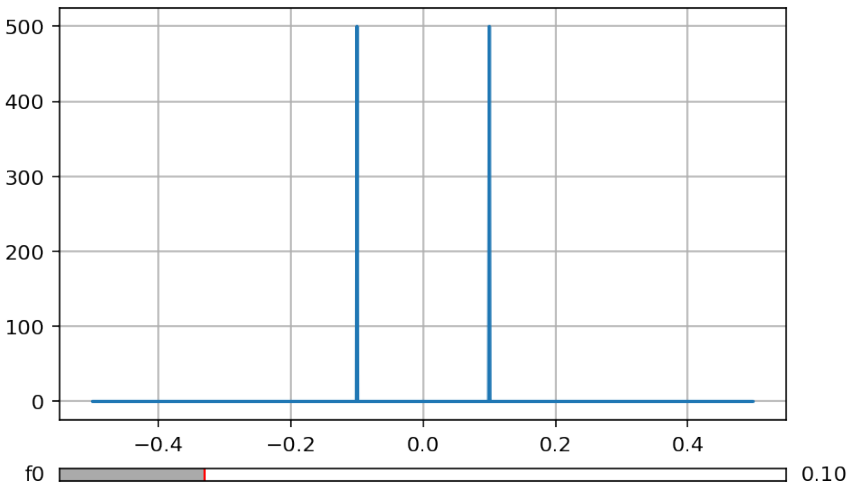


Widget Javascript not detected. It may not be installed or enabled properly.




```
%matplotlib tk
from matplotlib.widgets import Slider

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2, left=0.1)

slider_ax = plt.axes([0.1, 0.1, 0.8, 0.02])
slider = Slider(slider_ax, "f0", 0, 0.5, valinit=0.1, color='#AAAAAA')
f = np.linspace(-0.5, 0.5, 400)
N = 1000
t = np.arange(N)
s = np.sin(2 * pi * f0 * t)
Sf = fft(s, NN)
f = np.fft.fftfreq(NN)
line, = ax.plot(f, np.abs(Sf))

ax.grid(b=True)


def on_change(f0):
    s = np.sin(2 * pi * f0 * t)
    Sf = fft(s, NN)
    line.set_ydata(np.abs(Sf))


#    line2.set_ydata(sinc(pi*L*f))

slider.on_changed(on_change)
```

0




```
%matplotlib inline
```

Some definitions

3.3 Symmetries of the Fourier transform.

Consider the Fourier pair

$$x(n) \rightleftharpoons X(f).$$

When $x(n)$ is complex valued, we have

$$\boxed{x^*(n) \rightleftharpoons X^*(-f)}.$$

This can be easily checked beginning with the definition of the Fourier transform:

$$\begin{aligned} \mathrm{FT}\{x^*(n)\} &= \sum_n x^*(n) e^{-j2\pi fn}, \\ &= \left(\int_{[1]} x(n) e^{j2\pi fn} \mathrm{d}f \right)^*, \\ &= X^*(-f). \end{aligned}$$

In addition, for any signal $x(n)$, we have

$$\boxed{x(-n) \rightleftharpoons X(-f)}.$$

This last relation can be derived directly from the Fourier transform of $x(-n)$

$$\mathrm{FT}\{x(-n)\} = \int_{-\infty}^{+\infty} x(-n)\, e^{-j2\pi ft}\, \mathrm{d}t,$$

using the change of variable $-t \to t$, we get

$$\begin{aligned} \mathrm{FT}\{x(-n)\} &= \int_{-\infty}^{+\infty} x(n)\, e^{j2\pi tf}\, \mathrm{d}t, \\ &= X(-f). \end{aligned}$$

using the two last emphasized relationships, we obtain

$$\boxed{x^*(-n) \rightleftharpoons X^*(f)}.$$

To sum it all up, we have

$$\boxed{\begin{array}{lcl} x(n) & \rightleftharpoons & X(f) \\ x(-n) & \rightleftharpoons & X(-f) \\ x^*(n) & \rightleftharpoons & X^*(-f) \\ x^*(-n) & \rightleftharpoons & X^*(f) \end{array}}$$

These relations enable to analyse all the symetries of the Fourier transform. We begin with the *Hermitian symmetry* for **real signals**:

$$\boxed{X(f) = X^*(-f)}$$

from that, we observe that if $x(n)$ is real, then

- the real part of $X(f)$ is *even*,
- the imaginary part of $X(f)$ is *odd*,
- the modulus of $X(f)$, $|X(f)|$ is *even*,
- the phase of $X(f)$, $\theta(f)$ is *odd*.

Moreover, if $x(n)$ is odd or even ($x(n)$ is not necessarily real), we have

$$\boxed{\begin{array}{llcll} \text{[even]} & x(n) = x(-n) & \rightleftharpoons & X(f) = X(-f) & \text{[even]} \\ \text{[odd]} & x(n) = -x(-n) & \rightleftharpoons & X(f) = -X(-f) & \text{[odd]} \end{array}}$$

The following table summarizes the main symmetry properties of the Fourier transform:

$\mathbf{x(n)}$	Symmetry	Time	Frequency	consequence on $X(f)$
real	any	$x(n) = x^*(n)$	$X(f) = X^*(-f)$	Re. even, Im. odd
real	even	$x(n) = x^*(n) = x(-n)$	$X(f) = X^*(-f) = X(-f)$	Real and even
real	odd	$x(n) = x^*(n) = -x(-n)$	$X(f) = X^*(-f) = -X(-f)$	Imaginary and odd
imaginary	any	$x(n) = -x^*(n)$	$X(f) = -X^*(-f)$	Re. odd, Im. even
imaginary	even	$x(n) = -x^*(n) = x(-n)$	$X(f) = -X^*(-f) = X(-f)$	Imaginary and even
imaginary	odd	$x(n) = -x^*(n) = -x(-n)$	$X(f) = -X^*(-f) = -X(-f)$	Real and odd

Finally, we have

$$\boxed{\begin{array}{lcl} \text{Real even + imaginary odd} & \rightleftharpoons & \text{Real} \\ \text{Real odd + imaginary even} & \rightleftharpoons & \text{Imaginary} \end{array}}$$

3.4 Table of Fourier transform properties

The following table lists the main properties of the Discrete time Fourier transform. The table is adapted from the article on discrete time Fourier transform on Wikipedia.

Property	Time domain $x(n)$	Frequency domain $X(f)$
Linearity	$ax(n)+by(n)$	$aX(f)+bY(f)$
Shift in time	$x(n-n_0)$	$X(f)e^{-j2\pi fn_0}$
Shift in frequency (modulation)	$x(n)e^{j2\pi f_0 n}$	$X(f-f_0)$
Time scaling	$x(n/k)$	$X(kf)$
Time reversal	$x(-n)$	$X(-f)$
Time conjugation	$x(n)^*$	$X(-f)^*$
Time reversal & conjugation	$x(-n)^*$	$X(f)^*$
Sum of $x(n)$	$\sum_{n=-\infty}^{\infty} x(n)$	$X(0)$
Derivative in frequency	$\frac{n}{i}x(n)$	$\frac{dX(f)}{df}$
Integral in frequency	$\frac{i}{n}x(n)$	$\int_{[1]} X(f)df$
Convolve in time	$x(n)*y(n)$	$X(f)\cdot Y(f)$
Multiply in time	$x(n)\cdot y(n)$	$\int_{[1]} X(f_1)\cdot Y(f-f_1)df_1$
Area under $X(f)$	$x(0)$	$\int_{[1]} X(f)\,df$
Parseval's theorem	$\sum_{n=-\infty}^{\infty} x(n)\cdot y^*(n)$	$\int_{[1]} X(f)\cdot Y^*(f)df$
Parseval's theorem	$\sum_{n=-\infty}^{\infty} \lvert x(n)\rvert^2$	$\int_{[1]} \lvert X(f)\rvert^2 df$

Some examples of Fourier pairs are collected below:

Time domain	Frequency domain
$x[n]$	$X(f)$
$\delta[n]$	$X(f)=1$
$\delta[n-M]$	$X(f)=e^{-j2\pi fM}$
$\sum_{k=-\infty}^{\infty}\delta[n-kM]$	$\frac{1}{M}\sum_{k=-\infty}^{\infty}\delta\left(f-\frac{k}{M}\right)$
$u[n]$	$X(f)=\frac{1}{1-e^{-j2\pi f}}+\frac{1}{2}\sum_{k=-\infty}^{\infty}\delta(f-k)$
$a^n u[n]$	$X(f)=\frac{1}{1-ae^{-j2\pi f}}$
$e^{-j2\pi f_a n}$	$X(f)=\delta(f+f_a)$
$\cos(2\pi f_a n)$	$X(f)=\frac{1}{2}[\delta(f+f_a)+\delta(f-f_a)]$
$\sin(2\pi f_a n)$	$X(f)=\frac{1}{2j}[\delta(f+f_a)-\delta(f-f_a)]$
$\mathrm{rect}_M[(n-(M-1)/2)]$	$X(f)=\frac{\sin[\pi fM]}{\sin(\pi f)}e^{-j\pi f(M-1)}$
$\begin{cases}0 & n=0\\ \frac{(-1)^n}{n} & \text{elsewhere}\end{cases}$	$X(f)=j2\pi f$
$\begin{cases}0 & n \text{ even}\\ \frac{2}{\pi n} & n \text{ odd}\end{cases}$	$X(f)=\begin{cases}j & f<0\\ 0 & f=0\\ -j & f>0\end{cases}$

4 Filters and convolution

4.1 Representation formula

Any signal $x(n)$ can be written as follows:

$$\boxed{x(n) = \sum_{m=-\infty}^{+\infty} x(m)\delta(n-m).}$$

It is very important to understand the meaning of this formula.

- Since $\delta(n-m) = 1$ if and only if $n = m$, then all the terms in the sum cancel, excepted the one with $n = m$ and therefore we arrive at the identity $x(n) = x(n)$.
- The set of delayed Dirac impulses $\delta(n-m)$ form a basis of the space of discrete signals. Then the coordinate of a signal on this basis is the scalar product $\sum_{n=-\infty}^{+\infty} x(n)\delta(n-m) = x(m)$. Hence, the reprentation formula just expresses the decomposition of the signal on the basis, where the $x(m)$ are the coordinates.

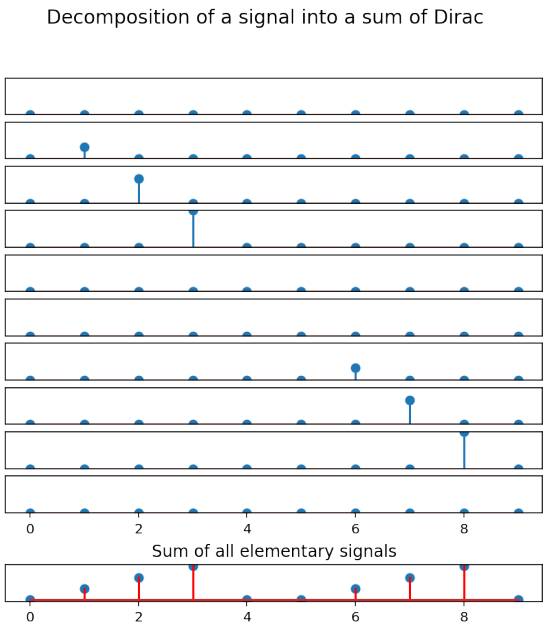
This means that $x(n)$, as a waveform, is actually composed of the sum of many Dirac impulses, placed at each integer, with a weight $x(m)$ which is nothing but the amplitude of the signal at time $m = n$. The formula shows how the signal can be seen as the superposition of Dirac impulses with the correct weights. Lets us illustrate this with a simple Python demonstration:

```
L = 10
z = np.zeros(L)
x = np.zeros(L)
x[5:9] = range(4)
x[0:4] = range(4)
print("x=", x)
s = np.zeros((L, L))
for k in range(L):
    s[k][k] = x[k]
# this is equivalent as s=np.diag(x)
f, ax = plt.subplots(L + 2, figsize=(7, 7))
for k in range(L):
    ax[k].stem(s[k][:])
    ax[k].set_ylim([0, 3])
    ax[k].get_yaxis().set_ticks([])
    if k != L - 1: ax[k].get_xaxis().set_ticks([])
```

```
ax[L].axis('off')

ax[L + 1].get_yaxis().set_ticks([])
ax[L + 1].stem(x, linefmt='r')
ax[L + 1].set_title("Sum of all elementary signals")
#f.tight_layout()
f.suptitle("Decomposition of a signal into a sum of Dirac", fontsize=14)
```

```
x= [0. 1. 2. 3. 0. 0. 1. 2. 3. 0.]
```

```
Text(0.5, 0.98, 'Decomposition of a signal into a sum of Dirac')
```



4.2 The convolution operation

4.2.1 Definition

Using previous elements, we are now in position of characterizing more precisely the *filters*. As already mentioned, a filter is a linear and time-invariant system, see Intro_Filtering.

The system being time invariant, the output associated with $x(m)\delta(n-\tau)$ is $x(m)h(n-m)$, if h is the impulse response.

$$x(m)\delta(n-m) \rightarrow x(m)h(n-m).$$

Since we know that any signal $x(n)$ can be written as (representation formula)

$$x(n) = \sum_{m=-\infty}^{+\infty} x(m)\delta(n-m),$$

we obtain, by linearity –that is superposition of all outputs, that

$$\boxed{y(n) = \sum_{m=-\infty}^{+\infty} x(m)h(n-m) = [x*h](n).}$$

This relation is called *convolution* of x and h, and this operation is denoted $[x*h](t)$, so as to indicate that the *result* of the convolution operation is evaluated at time n and that the variable m is simply a dummy variable that disappears by the summation.

The convolution operation is important since it enables to compute the output of the system using only its impulse response. It is not necessary to know the way the system is build, its internal design and so on. The only thing one must have is its impulse response. Thus we see that the knowledge of the impulse response enable to fully characterize the input-output relationships.

4.2.2 Illustration

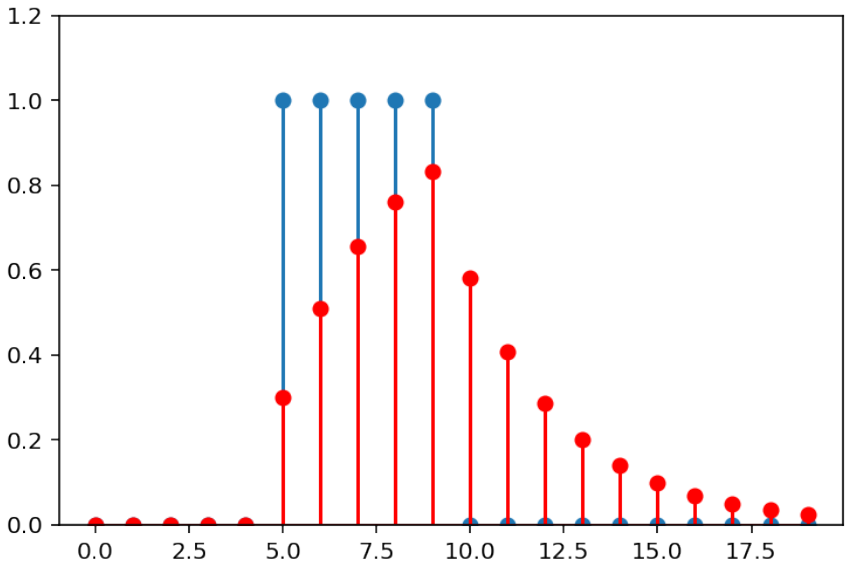
We show numerically that the output of a system is effectively the weightened sum of delayed impulse responses. This indicates that the output of the system can be computed either by using its difference equation, or by the convolution of its input with its impulse response.

Direct response

```
def op3(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]= 0.7*transformed_signal[t-1]+0.3*signal[t]
    return transformed_signal
#
# rectangular pulse
N=20; L=5; M=10
r=np.zeros(N)
r[L:M]=1
#
plt.stem(r)
plt.stem(op3(r),linefmt='r-',markerfmt='ro')
_=plt.ylim([0, 1.2])
```

Response by the sum of delayed impulse responses

```
s = np.zeros((N, N))
for k in range(N):
    s[k][k] = r[k]
# this is equivalent to s=np.diag(x)
ll = range(5, 10)
llmax = ll[-1]
f, ax = plt.subplots(len(ll) + 2, figsize=(7, 7))
u = 0
sum_of_responses = np.zeros(N)
for k in ll:
    ax[u].stem(s[k][:])
    ax[u].stem(2 * op3(s[k][:]), linefmt='r-', markerfmt='ro')
    ax[u].set_ylim([0, 1.3])
    ax[u].set_ylabel('k={}'.format(k))
    ax[u].get_yaxis().set_ticks([])
    sum_of_responses += op3(s[k][:])
    if u != llmax - 1: ax[u].get_xaxis().set_ticks([])
```



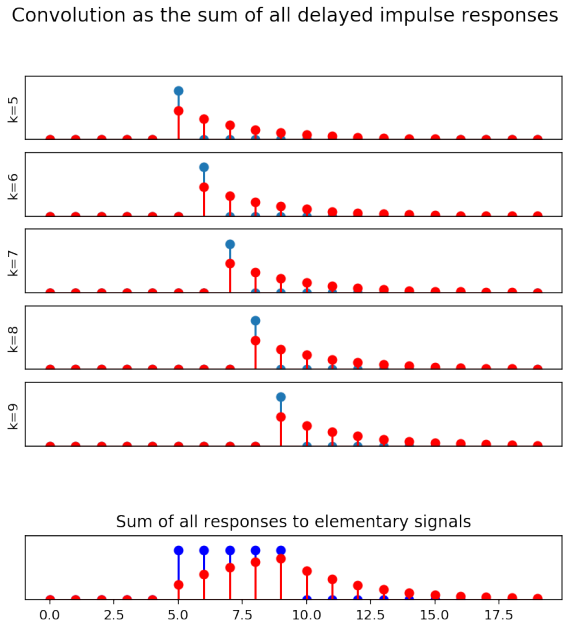

```
    u += 1

ax[u].axis('off')

ax[u + 1].get_yaxis().set_ticks([])
ax[u + 1].stem(r, linefmt='b-', markerfmt='bo')
ax[u + 1].stem(sum_of_responses, linefmt='r-', markerfmt='ro')
ax[u + 1].set_ylim([0, 1.3])
ax[u + 1].set_title("Sum of all responses to elementary signals")

#
#f.tight_layout()
f.suptitle(
    "Convolution as the sum of all delayed impulse responses", fontsize=14)
```

```
Text(0.5, 0.98, 'Convolution as the sum of all delayed impulse responses')
```

4.2.3 Exercises

Exercise 3. *1. Compute* by hand *the convolution between two rectangular signals,*

2. *propose a python program that computes the result, given two arrays. Syntax:* `def myconv(x,y): return z`
3. *Of course, convolution functions have already be implemented, in many languages, by many people and using many algorithms. Implementations also exist in two or more dimensions. So, we do need to reinvent the wheel. Consult the help of* `np.convolve` *and of* `sig.convolve` *(respectively from numpy and scipy modules).*
4. *use this convolution to compute and display the convolution between two rectangular signals*




```
def myconv(x, y):
    L = np.size(x)
    # we do it in the simple case where both signals have the same length
    assert np.size(x) == np.size(
        y), "The two signals must have the same lengths"
    # as an exercise, you can generalize this

    z = np.zeros(2 * L - 1)
    #
    ## -> FILL IN
    #
    return z


# test it:
z = myconv(np.ones(L), np.ones(L))
print('z=', z)
```

```
z= [0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

```
def myconv(x, y):
    L = np.size(x)
    # we do it in the simpla case where both signals have the same length
    assert np.size(x) == np.size(
        y), "The two signals must have the same lengths"
    # as an exercise, you can generalize this

    z = np.zeros(2 * L - 1)
    # delay<L
    for delay in np.arange(0, L):
        z[delay] = np.sum(x[0:delay + 1] * y[-1:-1 - delay - 1:-1])
    # delay>=L
```

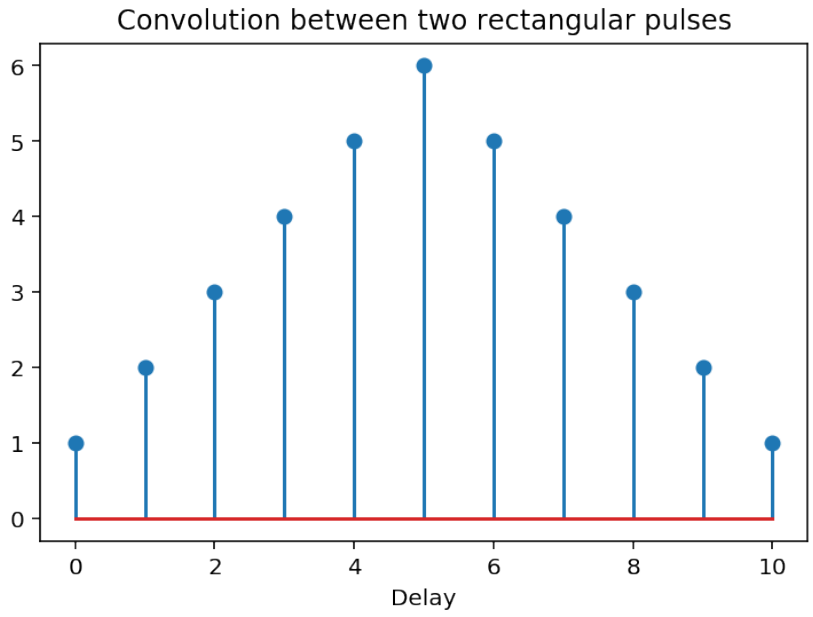
```
    for delay in np.arange(L, 2 * L - 1):
        z[delay] = np.sum(x[delay + 1 - L:L] * y[-delay - 1 + L:0:-1])
    return z


# test it:
z = myconv(np.ones(L), np.ones(L))
print('z=', z)
```

```
z= [1. 2. 3. 4. 5. 4. 3. 2. 1.]
```

Convolution with legacy convolve:

```
#help(np.convolve)
# convolution between two squares of length L
L = 6
z = sig.convolve(np.ones(L), np.ones(L))
plt.stem(z)
plt.title("Convolution between two rectangular pulses")
plt.xlabel("Delay")
```

```
Text(0.5, 0, 'Delay')
```



5 Transfer function

Given a filter with input $x(n)$ and output $y(n)$, it is always possible to compute the Fourier transform of he input and of the output, say $X(f)$ and $Y(f)$. The ratio of these two quantities is called the **transfer function**. For now, let us denote it by $T(f)$. Interestingly, we will see that the transfer function do not depend on x, and thus is a global characteristic of the system. More than that, we will see that the transfer function is intimately linked to the impulse response of the system.

5.1 The Plancherel relation

Convolution enables to express the output of a filter characterized by its impulse response. Consider a system with impulse response $h(n)$ and an input

$$x(n) = X_0 e^{j2\pi f_0 n}.$$

Its output is given by

$$\begin{aligned} y(n) &= \sum_m h(m) X_0 \, e^{j2\pi f_0 (n-m)} \\ &= X_0 e^{j2\pi f_0 n} \sum_m h(m) e^{-j2\pi f_0 m}. \end{aligned}$$

We recognize above the expression of the Fourier transform of $h(m)$ at the frequency f_0:

$$\boxed{H(f_0) = \sum_m h(m) e^{-j2\pi f_0 m}}.$$

Hence, the output can be simply written as

$$y(n) = X_0 e^{j2\pi f_0 n} H(f_0).$$

For a linear system excited by a complex exponential at frequency f_0 , we obtain that output is the **same** signal, up to a complex factor $H(f_0)$. This gives us another insight on the interest of the decomposition on complex exponentials: they are *eigen-functions* of filters, and $H(f_0)$ plays the role of the associated *eigenvalue*.

Consider now an arbitrary signal $x(n)$. It is possible to express $x(n)$ as an infinite sum of complex exponentials (this is nothing but the inverse Fourier transform);

$$x(n) = \int_{[1]} X(f) e^{j2\pi f n} \mathrm{d}f.$$

To each component $X(f)e^{j2\pi fn}$ corresponds an output $X(f)H(f)e^{j2\pi fn}$, and, by superposition,

$$y(n) = \int_{[1]} X(f)H(f)e^{j2\pi fn}\mathrm{d}f.$$

Thefore, we see that the Fourier transform of the output, $Y(f)$, is simply

$$\boxed{Y(f) = X(f)H(f)}.$$

The time domain description, in terms of convolution product, becomes a simple product in the Fourier domain.

$$\boxed{[x*h](n) \rightleftharpoons X(f)H(f)}.$$

It is easy to check that reciprocally,

$$\boxed{x(n)h(n) \rightleftharpoons [X*H](f)}.$$

Try to check it as an exercise. You will need to introduce a convolution for function of a continuous variable, following the model of the convolution for discrete signals.

Begin with the Fourier transform of $x(n)y(n)$, and replace the signals by their expression as the inverse Fourier transform:

$$\begin{aligned}
\mathrm{FT}[x(n)y(n)] &= \sum_n x(n)y(n)e^{-j2\pi fn} && (5.1)\\
&= \sum_n \int X(f_1)e^{j\pi f_1 n}\mathrm{d}f_1 \int Y(f_2)e^{j\pi f_2 n}\mathrm{d}f_2 e^{-j2\pi fn} && (5.2)\\
&= \iint X(f_1)Y(f_2)\sum_n e^{j\pi f_1 n}e^{j\pi f_2 n}e^{-j2\pi fn}\mathrm{d}f_1\mathrm{d}f_2 && (5.3)
\end{aligned}$$

It remains to note that the sum of exponentials is nothing but the Fourier transform of the complex exponential $e^{j\pi(f_1+f_2)n}$, and thus that

$$\sum_n e^{j\pi f_1 n}e^{j\pi f_2 n}e^{-j2\pi fn} = \delta(f - f_1 - f_2).$$

Therefore, the double integral above reduces to a simple one, since $f_2 = f - f_1$, and we obtain

$$\mathrm{FT}[x(n)y(n)] = \int X(f_1)Y(f - f_1)\mathrm{d}f_1 = [X * Y](f).$$

(Another proof is possible, beginning with the inverse Fourier transform of the convolution $[X*Y](f)$, and decomposing the exponential so as to exhibit the inverse Fourier transform of $x(n)$ and $y(n)$). Try it.

The transform of a convolution into a simple product, and reciprocally, constitutes the Plancherel theorem:

$$\boxed{\begin{array}{rcl} [x*y](t) & \rightleftharpoons & X(f)Y(f), \\ x(t)y(t) & \rightleftharpoons & [X*Y](f). \end{array}}$$

This theorem has several important consequences.

5.2 Consequences

The Fourier transform of $x(n)y(n)^*$ is

$$x(n)y(n)^* \rightleftharpoons \int_{[1]} X(u)Y(f-u)^*\mathrm{d}u,$$

since $\mathrm{FT}y^*(n) = Y^*(-f)$. Therefore,

$$\mathrm{FT}x(n)y(n)^* = \int_{[1]} X(u)Y(u-f)^* \mathrm{d}u,$$

that is, for $f = 0$,

$$\boxed{\sum_{-\infty}^{+\infty} x(n)y^*(n) = \int_{[1]} X(u)Y^*(u)\mathrm{d}u}.$$

This relation shows that *the scalar product is conserved* in the different basis for signals. This property is called the **Plancherel-Parseval theorem**. Using this relation with $y(n) = x(n)$, we have

$$\boxed{\sum_{-\infty}^{+\infty} |x(n)|^2 = \int_{[1]} |X(f)|^2 \mathrm{d}f},$$

which is a relation indicating *energy conservation*. It is the **Parseval relation**.

6

Basic representations for digital signals and systems

Par J.-F. Bercher – march 5, 2014

This lab is an elementary introduction to the analysis of a filtering operation. In particular, we will illustrate the notions of impulse response, convolution, frequency representation transfer function.

In these exercises, we will work with digital signals. Experiments will be done with Python.

We will work with the filtering operation described by the following difference equation

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

where $x(n)$ is the filter's input and $y(n)$ its output.

6.1 Study in the time domain

1. Compute analytically the impuse response (IR), as a function of the parameter a, assuming that the system is causal and that the initial conditions are zero.
2. Under Python, look at the help of function lfilter, by `help(lfilter)` and try to understand how it works. Propose a method for computing numerically the impulse response. Then, check graphically the impulse response, with $a = 0.8$. The following Dirac function enables to generate a Dirac impulse in discrete time: def dirac(n): """ dirac(n): returns a Dirac impulse on N points""" d=zeros(n); d[0]=1 return d
3. Compute and plot the impulse responses for $a = -0.8$, $a = 0.99$, and $a = 1.01$. Conclusions.

6.2 Study in the fequency domain

2. Give the expression of the transfer function $H(f)$, and of its modulus $|H(f)|$ for any a. Give the theoretical amplitudes at $f = 0$ and $f = 1/2$ (in normalized frequencies, *i.e.* normalized with respect to Fe. Compute numerically the transfer function as the Fourier transform of the impulse response, for $a = 0.8$ and $a = -0.8$, and plot the results. Conclusions.

7 Filtering

1. Create a sine wave x of frequency $f0 = 3$, sampled at $Fe = 32$ on $N = 128$ points

2. Filter this sine wave by the previous filter

– using the function filter, y1=lfilter([1],[1 -0.8],x);

– using a convolution, y2=lfilter(h,1,x); with h the impulse response of the filter for $a = 0.8$ Explain why this last operation effectively corresponds to a convolution. Compare the two results.

3. Plot the transfer function and the Fourier transform of the sine wave. What will be the result of the product? Measure the gain and phase of the transfer function at the frequency of the sinusoid ($f_0 = 3$). Compare these values to the values of gain and phase measured in the time domain.

4. Do this experiment again, but with a pulse train instead of a sine. This is done simply in order to illustrate the fact that this time, the output of the filter is deformed. You may use def rectpulse(x): """rectpulse(x): Returns a pulse train with period 2pi""" return sign(sin(x))

8

Lab – Basic representations for digital signals and systems

Par J.-F. Bercher – le 12 novembre 2013 English translation and update: february 21, 2014 – last update: 2018

```
%matplotlib inline
#import mpld3
#mpld3.enable_notebook()
```

This lab is an elementary introduction to the analysis of a filtering operation. In particular, we will illustrate the notions of impulse response, convolution, frequency representation transfer function.

In these exercises, we will work with digital signals. Experiments will be done with Python.

We will work with the filtering operation described by the following difference equation

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

where $x(n)$ is the filter's input and $y(n)$ its output.

8.1 Study in the time domain

1. Compute analytically the impuse response (IR), as a function of the parameter a, assuming that the system is causal and that the initial conditions are zero.
2. Under Python, look at the help of function lfilter, by `help(lfilter)` and try to understand how it works. Propose a method for computing numerically the impulse response. Then, check graphically the impulse response, with $a = 0.8$. The following Dirac function enables to generate a Dirac impulse in discrete time:

def dirac(n): """ dirac(n): returns a Dirac impulse on N points""" d=zeros(n); d[0]=1 return d

3. Compute and plot the impulse responses for $a = -0.8$, $a = 0.99$, and $a = 1.01$. Conclusions.

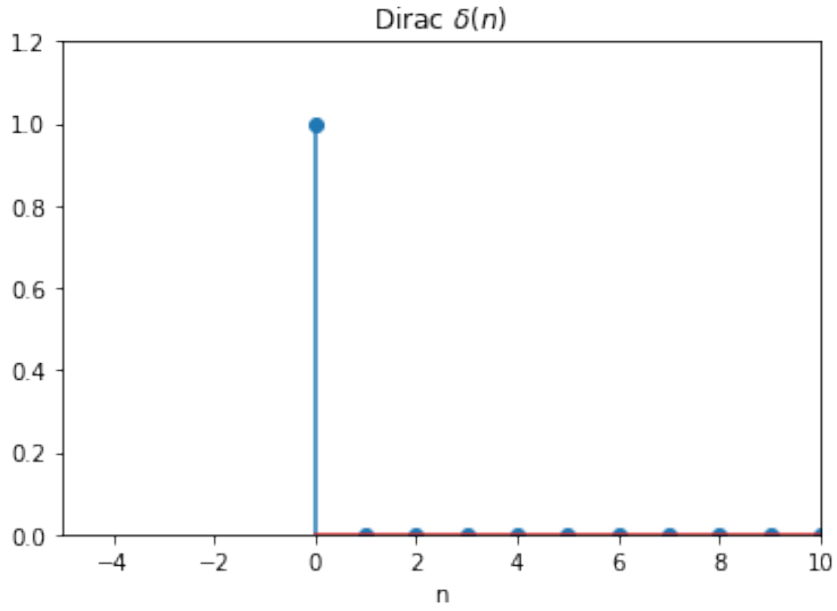
```
from pylab import *
```

We begin by creating a function that returns a **Dirac impulse**, and test the result

```
def dirac(n):
    """ dirac(n): returns a Dirac impulse on N points"""
    d=zeros(n); d[0]=1
    return d
```

```
# Representation
N=100
stem(range(N),dirac(N))
title("Dirac $\delta(n)$")
xlabel("n")
ylim([0, 1.2])     # zoom for better visualization
xlim([-5, 10])
```

```
(-5, 10)
```



8.1.1 The function scipy.signal *lfilter()*

```
import scipy
from scipy.signal import lfilter
help(lfilter)
```

```
Help on function lfilter in module scipy.signal.signaltools:

lfilter(b, a, x, axis=-1, zi=None)
    Filter data along one-dimension with an IIR or FIR filter.

    Filter a data sequence, `x`, using a digital filter.  This works for many
    fundamental data types (including Object type).  The filter is a direct
    form II transposed implementation of the standard difference equation
```

```
(see Notes).

Parameters
----------
b : array_like
    The numerator coefficient vector in a 1-D sequence.
a : array_like
    The denominator coefficient vector in a 1-D sequence.  If ``a[0]``
    is not 1, then both `a` and `b` are normalized by ``a[0]``.
x : array_like
    An N-dimensional input array.
axis : int, optional
    The axis of the input data array along which to apply the
    linear filter. The filter is applied to each subarray along
    this axis.  Default is -1.
zi : array_like, optional
    Initial conditions for the filter delays.  It is a vector
    (or array of vectors for an N-dimensional input) of length
    ``max(len(a), len(b)) - 1``.  If `zi` is None or is not given then
    initial rest is assumed.  See `lfiltic` for more information.

Returns
-------
y : array
    The output of the digital filter.
zf : array, optional
    If `zi` is None, this is not returned, otherwise, `zf` holds the
    final filter delay values.

See Also
--------
lfiltic : Construct initial conditions for `lfilter`.
lfilter_zi : Compute initial state (steady state of step response) for
             `lfilter`.
filtfilt : A forward-backward filter, to obtain a filter with linear phase.
savgol_filter : A Savitzky-Golay filter.
sosfilt: Filter data using cascaded second-order sections.
sosfiltfilt: A forward-backward filter using second-order sections.

Notes
-----
The filter function is implemented as a direct II transposed structure.
This means that the filter implements::

   a[0]*y[n] = b[0]*x[n] + b[1]*x[n-1] + ... + b[M]*x[n-M]
                         - a[1]*y[n-1] - ... - a[N]*y[n-N]

where `M` is the degree of the numerator, `N` is the degree of the
```

```
denominator, and `n` is the sample number.  It is implemented using
the following difference equations (assuming M = N)::

     a[0]*y[n] = b[0] * x[n]                + d[0][n-1]
       d[0][n] = b[1] * x[n] - a[1] * y[n] + d[1][n-1]
       d[1][n] = b[2] * x[n] - a[2] * y[n] + d[2][n-1]
     ...
     d[N-2][n] = b[N-1]*x[n] - a[N-1]*y[n] + d[N-1][n-1]
     d[N-1][n] = b[N] * x[n] - a[N] * y[n]

where `d` are the state variables.

The rational transfer function describing this filter in the
z-transform domain is::

                          -1              -M
              b[0] + b[1]z  + ... + b[M] z
     Y(z) = ---------------------------------- X(z)
                          -1              -N
              a[0] + a[1]z  + ... + a[N] z

Examples
--------
Generate a noisy signal to be filtered:

>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(-1, 1, 201)
>>> x = (np.sin(2*np.pi*0.75*t*(1-t) + 2.1) +
...      0.1*np.sin(2*np.pi*1.25*t + 1) +
...      0.18*np.cos(2*np.pi*3.85*t))
>>> xn = x + np.random.randn(len(t)) * 0.08

Create an order 3 lowpass butterworth filter:

>>> b, a = signal.butter(3, 0.05)

Apply the filter to xn.  Use lfilter_zi to choose the initial condition of
the filter:

>>> zi = signal.lfilter_zi(b, a)
>>> z, _ = signal.lfilter(b, a, xn, zi=zi*xn[0])

Apply the filter again, to have a result filtered at an order the same as
filtfilt:

>>> z2, _ = signal.lfilter(b, a, z, zi=zi*z[0])
```

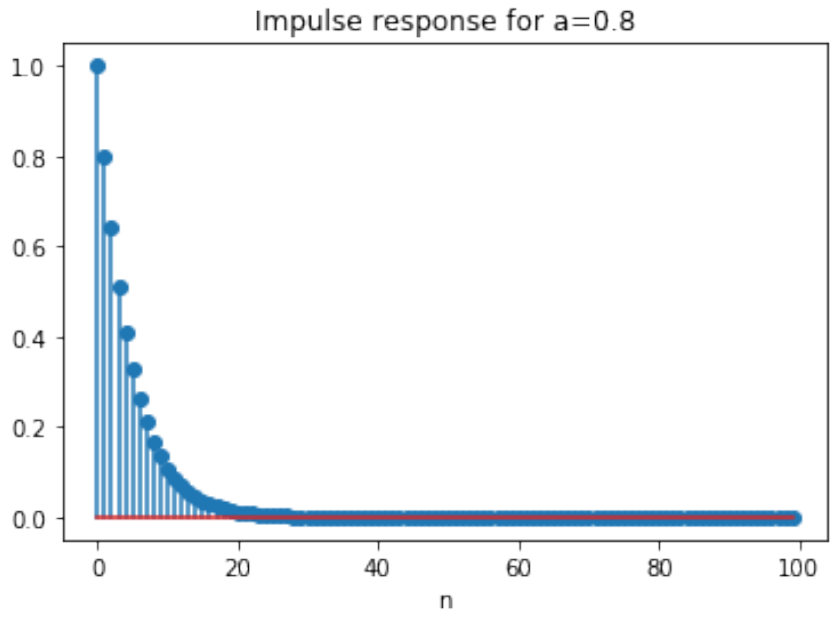
```
Use filtfilt to apply the filter:

>>> y = signal.filtfilt(b, a, xn)

Plot the original signal and the various filtered versions:

>>> plt.figure
>>> plt.plot(t, xn, 'b', alpha=0.75)
>>> plt.plot(t, z, 'r--', t, z2, 'r', t, y, 'k')
>>> plt.legend(('noisy signal', 'lfilter, once', 'lfilter, twice',
...             'filtfilt'), loc='best')
>>> plt.grid(True)
>>> plt.show()
```

according to the difference equation $y(n) = ay(n-1) + x(n)$ corresponds to the command `y=lfilter([1],[1, -a],x)`, whre, of course, x and a have been previously initialized.

⇒ **In order to obtain the impulse response, one simply have to excite the system with an impulse!**

```
a=0.8
N=100
x=dirac(N)
y=lfilter([1],[1, -a],x)
stem(y),
title("Impulse response for a={}".format(a)), xlabel("n")
```

```
(Text(0.5, 1.0, 'Impulse response for a=0.8'), Text(0.5, 0, 'n'))
```



The first values are:

```
print("First values \n y[:6]=" , y[:6])
print("to compare with a**n :\n", a**arange(0,6))
```

```
First values
 y[:6]= [1.       0.8      0.64     0.512    0.4096   0.32768]
to compare with a**n :
 [1.       0.8      0.64     0.512    0.4096   0.32768]
```

We note that the `experimental` impulse response corresponds to the theoretical one, which is $h(n) = a^n$.

We will check this for some other values of a.

To ease our explorations, we will first define a function that returns the impulse reponse, for two vectors [b] and [a] describing any rational filter. It suffices to compute the filter's output, with a Dirac at its input, on a specified length:

```
def ri(b,a,n):
    """ Returns an impulse response of length n (int)
    of a filter with coefficients a and b
    """
    return lfilter(array(b),array(a),dirac(n))
```

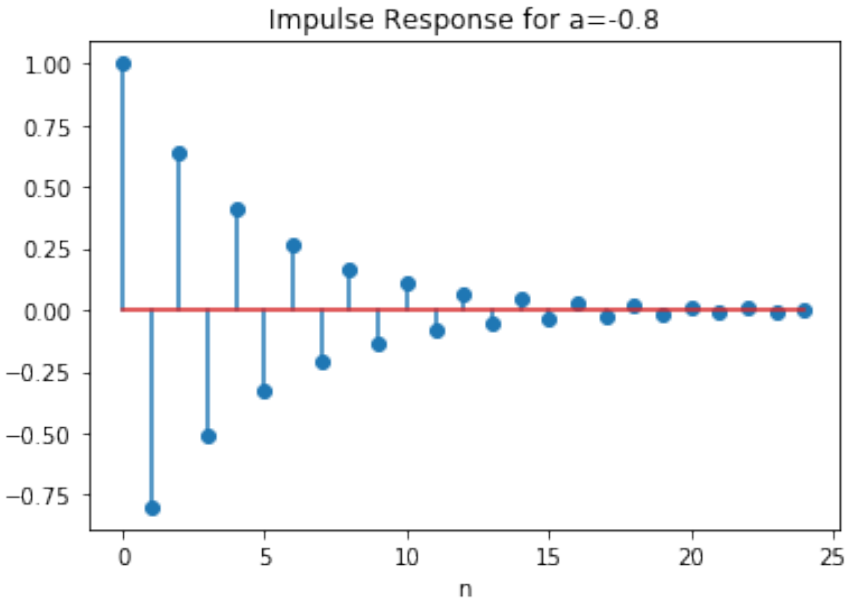
8.2 Display of results

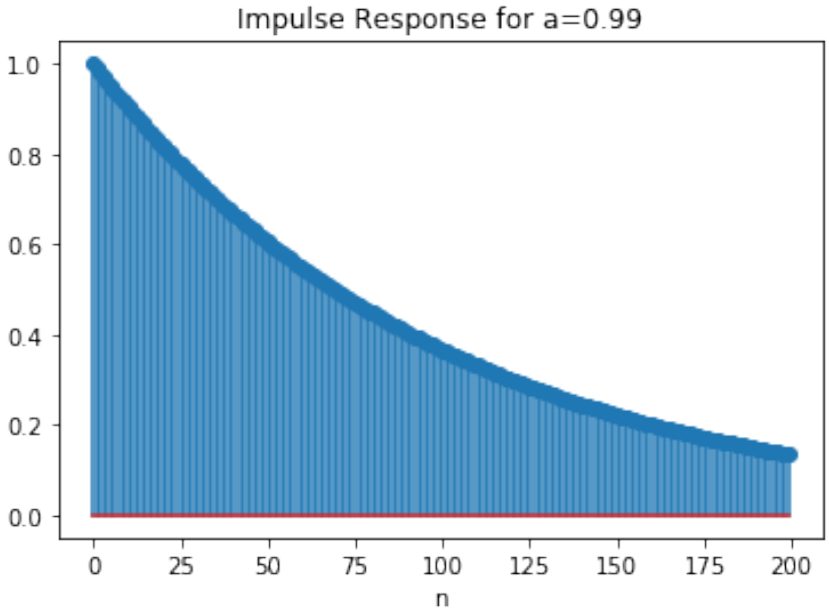
```
N=25
axe_n=range(N)
a=-0.8
figure()
stem(axe_n,ri([1],[1, -a],N))
title("Impulse Response for a={}".format(a))
xlabel("n")
#
N=200
axe_n=range(N)
a=0.99
figure()
stem(axe_n,ri([1],[1, -a],N))
title("Impulse Response for a={}".format(a))
xlabel("n")
#
a=1.01
figure()
stem(axe_n,ri([1],[1, -a],N))
title("Impulse Response for a={}".format(a))
xlabel("n")
```

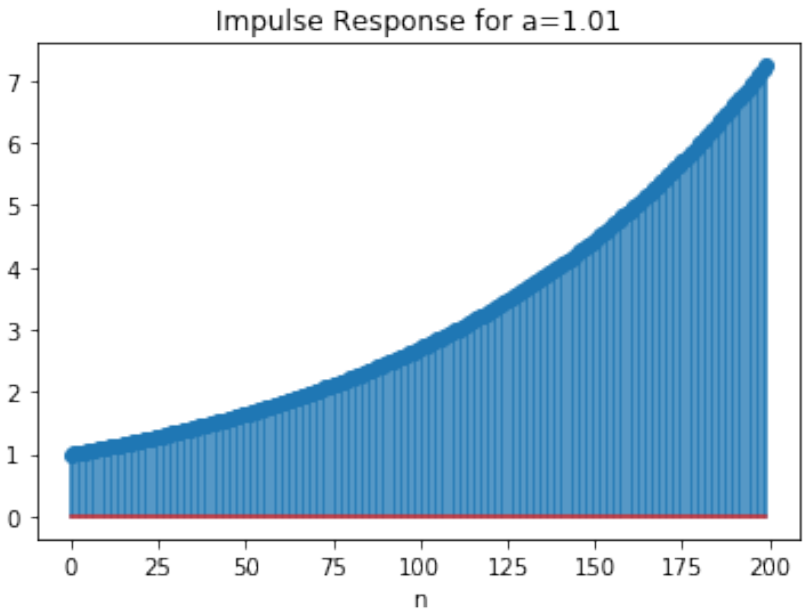
```
Text(0.5, 0, 'n')
```

Conclusions:

- For $a < 0$, the impulse response, theoretically a^n, is indeed of *alternate sign*


Impulse Response for a=-0.8
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
5
10
15
20
25
n


Impulse Response for a=0.99
1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175
200
n



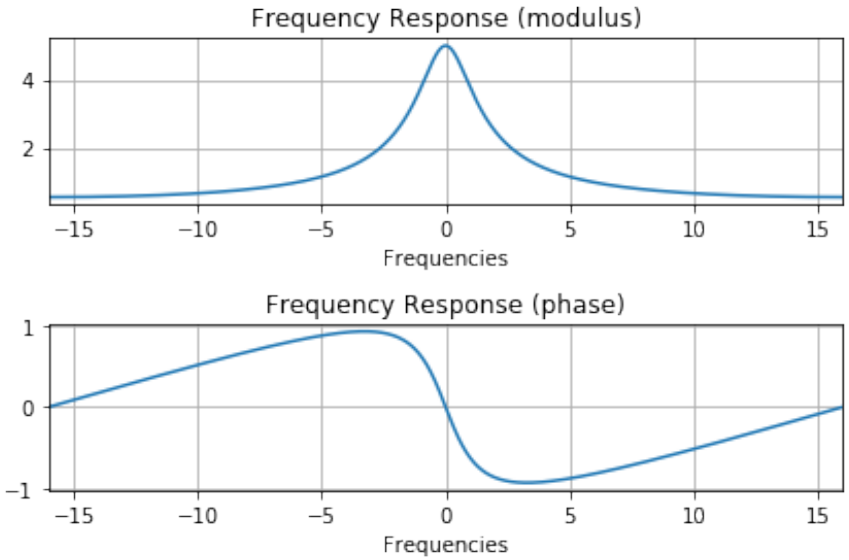
- for a near 1, $a < 1$, the impulse response is nearly constant
- for $a > 1$, the impulse response diverges...

8.3 Study in the frequency domain

2. Give the expression of the transfer function $H(f)$, and of its modulus $|H(f)|$ for any a. Give the theoretical amplitudes at $f = 0$ and $f = 1/2$ (in normalized frequencies, *i.e.* normalized with respect to Fe. Compute numerically the transfer function as the Fourier transform of the impulse response, for $a = 0.8$ and $a = -0.8$, and plot the results. Conclusions.

```
# We will need the fft functions
from numpy.fft import fft, ifft

# Computation of the impulse response
a=0.8
h=ri([1],[1, -a],300)

# Computation of the frequency response
M=1000
Fe=32
H=fftshift(fft(h,M))     # We use fftshift in order to center
                         #the reprentation
f=arange(M)/M*Fe -Fe/2   # definition of the frequency axis

fig=figure(4)    # and display
subplot(2,1,1)
plot(f,abs(H),label="Frequency Response")
xlabel("Frequencies")
title("Frequency Response (modulus)")
```

```
grid(b=True)
xlim([-Fe/2, Fe/2])
subplot(2,1,2)
plot(f,angle(H),label=u"Frequency Response")
xlabel("Frequencies")
title("Frequency Response (phase)")
grid(b=True)
xlim([-Fe/2, Fe/2])
fig.tight_layout()   # avoid recovering of titles and labels
```




```
# Value at f=x: we look for it by find(f==x)
print ("Value at f=0 : ".rjust(20),H[find(f==0)].real)
print ("Value at f=Fe/2 : ",H[find(f==-Fe/2)].real)
print ("To compare with theoretical values")
```

```
     Value at f=0 :  [5.]
Value at f=Fe/2 :  [0.55555556]
To compare with theoretical values


/home/bercherj/.local/lib/python3.5/site-packages/ipykernel_launcher.py:2: Matplotl

/home/bercherj/.local/lib/python3.5/site-packages/ipykernel_launcher.py:3: Matplotl
  This is separate from the ipykernel package so we can avoid doing imports until
```

8.4 Filtering

1. Create a sine wave x of frequency $f0 = 3$, sampled at $Fe = 32$ on $N = 128$ points

2. Filer this sine wave by the previous filter

– using the function filter, y1=lfilter([1],[1 -0.8],x);

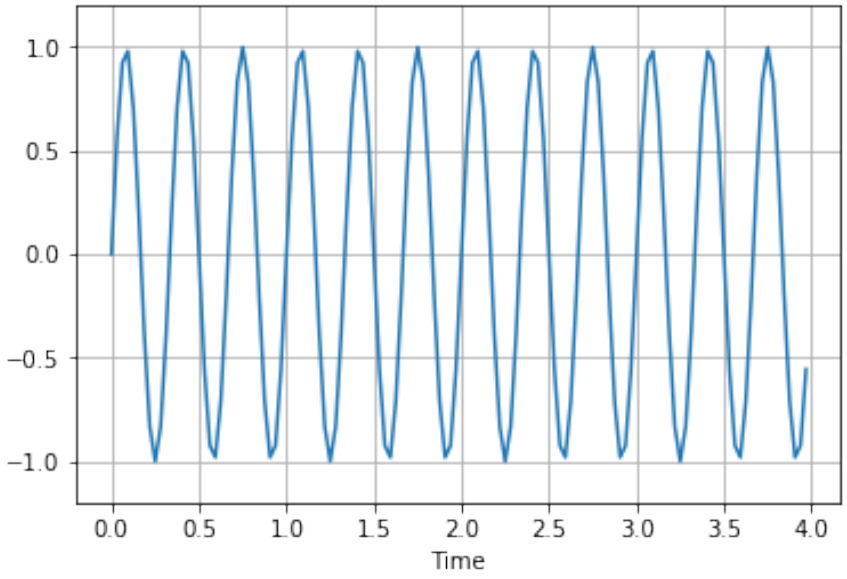
– using a convolution, y2=lfilter(h,1,x); with h the impulse resonse of the filter for $a = 0.8$

Explain why this last operation effectively corresponds to a convolution. Compare the two results.

8.4.1 Analysis in the time domain

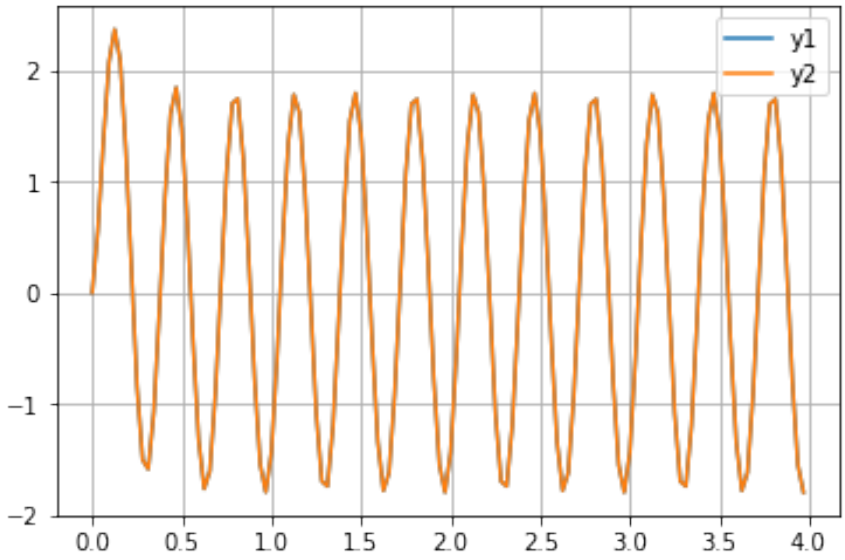
```
# Creation of the simple sine wave
N, fo, Fe = 128, 3, 32
t=arange(N)/Fe
x=sin(2*pi*fo*t)
figure(3)
plot(t,x)
xlabel("Time")
grid(b=True)
ylim([-1.2, 1.2])
```

```
(-1.2, 1.2)
```




```
# Filtering with filter h
a=0.8
h=ri([1],[1, -a],N) # h computed again, but on N points
y1=lfilter([1],[1, -0.8],x)
y2=lfilter(h,[1],x)
figure()
plot(t,y1,label='y1')
plot(t,y2,label='y2')
grid(b=True)
legend()
show()
```

```
/usr/local/lib/python3.5/site-packages/scipy/signal/signaltools.py:1344: FutureWarn
  out = out_full[ind]
```



One can also plot the difference between the two signals, so things are clear!

```
figure()
plot(t,y1-y2,label='y1-y2')
xlabel("Time")
grid(b=True)
legend()
```

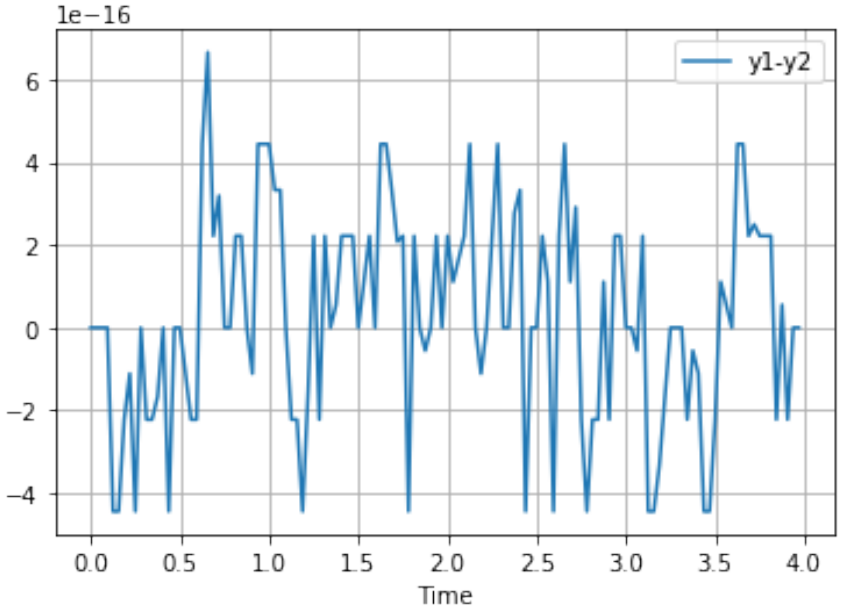
We are now going to check **Plancherel's theorem** which says that the Fourier transform of a convolution product is the product of the Fourier transforms. We will simply observe that the output of a system, computed as the inverse Fourier transform of the product of the transfer function H with the Fourier transform X of the input signal is identical (or at least extremely similar) to the output computed by convolution or as solution of the difference equation.

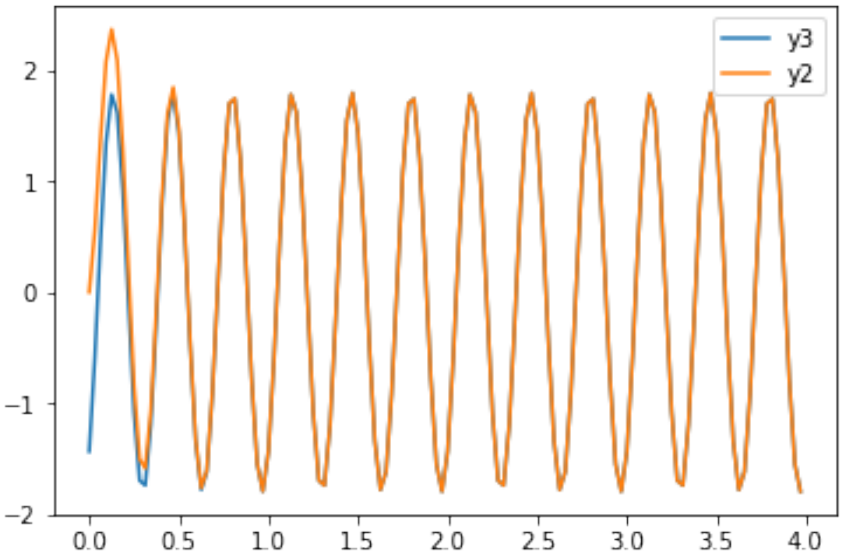
```
y3=real(ifft(fft(h)*fft(x)))
plot(t,y3,label='y3')
plot(t,y2,label='y2')
legend()
```

The difference observed at the beginning of the two plots comes from a different assumption on the values of the signals at negative (non observed) times. Actually, function `lfilter` assumes that the signal is zero where non observed, which implies a transient response at the output of the filter. The Fourier transform is computed with the algorithm of `fft`, which assumes that all signals are periodics, thus periodised outside the observation interval. We will discuss this in more details later.

8.4.2 Frequency representation

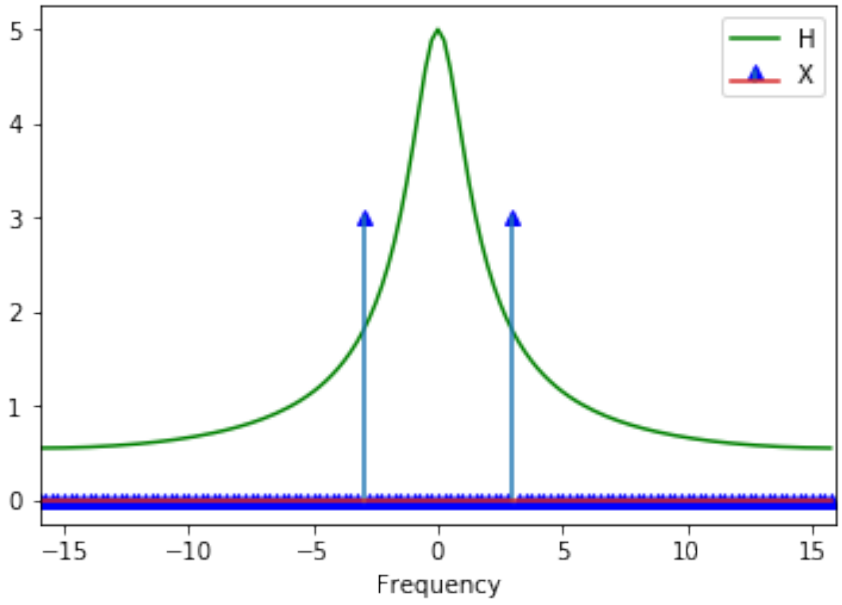
3. Plot the transfer function and the Fourier transform of the sine wave. What will be the result of the product? Measure the gain and phase of the transfer function at the frequency


1e−16
y1-y2
6
4
2
0
−2
−4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Time


y3
y2
2
1
0
−1
−2
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

of the sinusoid ($f_0 = 3$). Compare these values to the values of gain and phase measured in the time domain.

```
X=fftshift(fft(x))
H=fftshift(fft(h))
M=len(x)
f=arange(M)/M*Fe -Fe/2
plot(f,abs(H),color='green',label="H")
stem(f,abs(X)*6/M,markerfmt='b^',label="X")
xlim([-16, 16])
xlabel("Frequency")
legend()
```



The sine wave has frequency $fo = 3$. let us measure the values of gain and phase at this frequency:

```
H3=H[find(f==3)]
print("Value of the complex gain:", H3)
print("Modulus :", abs(H3))
print("Phase (degrees):", angle(H3)/pi*180)
```

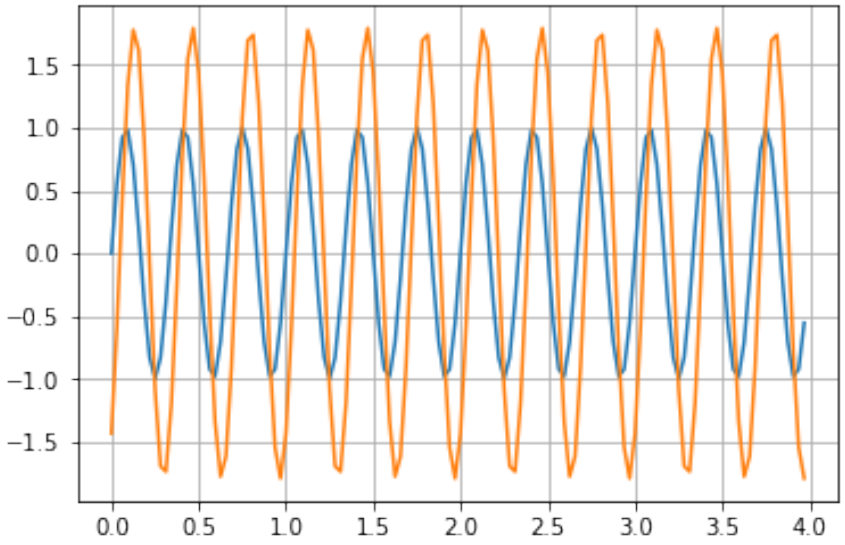
```
Value of the complex gain: [1.08130406-1.43535659j]
Modulus : [1.79707178]
Phase (degrees): [-53.00801337]
```

```
/home/bercherj/.local/lib/python3.5/site-packages/ipykernel_launcher.py:1: Matplotl
  """Entry point for launching an IPython kernel.
```

Now, let us look at this on the time representation.

```
figure()
plot(t,x,t,y3)
grid('on')
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/cbook/__init__.py:424: Matplotlib
Passing one of 'on', 'true', 'off', 'false' as a boolean is deprecated; use an actu
  warn_deprecated("2.2", "Passing one of 'on', 'true', 'off', 'false' as a "
```



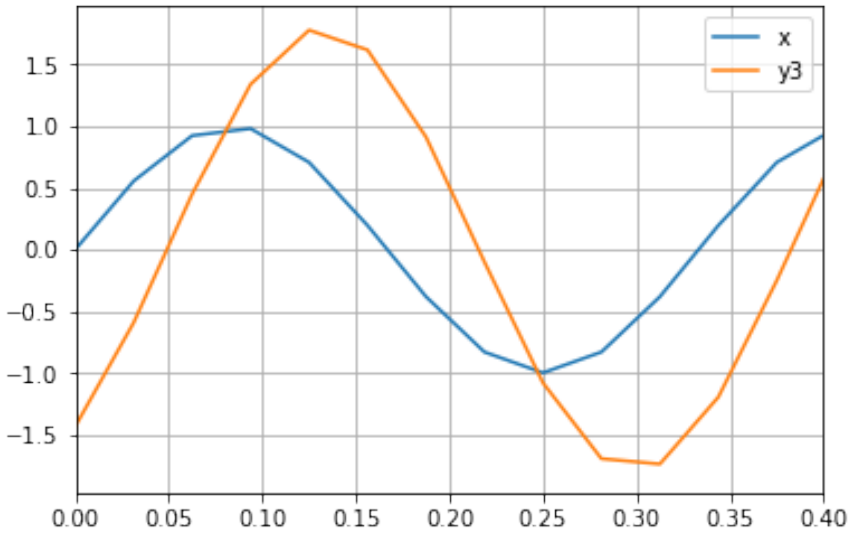
Measure of phase: we first measure the delay between the two signals

```
figure()
plot(t,x,label="x")
plot(t,y3,label="y3")
legend()
grid('on')
xlim([0, 0.4])
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/cbook/__init__.py:424: Matplotlib
Passing one of 'on', 'true', 'off', 'false' as a boolean is deprecated; use an actu
  warn_deprecated("2.2", "Passing one of 'on', 'true', 'off', 'false' as a "
```

```
(0, 0.4)
```

```
deltaT=min(find(y3>0))/Fe
# x begins at 0, the delay is given by the first value where
# y3 becomes >0
print("The value of the phase difference, in degrees, is ", (2*pi*fo)*
    deltaT/pi*180,"ř")
```




```
The value of the phase difference, in degrees,  is  67.5 ř


/home/bercherj/.local/lib/python3.5/site-packages/ipykernel_launcher.py:1: Matplotl
  """Entry point for launching an IPython kernel.
```

Observations : We see that if the input is a sine wave, then the output is **also** a sine wave, up to a gain and phase shift. These gain and phase corresponds exactly to the gain and phase given by the tranfer function.

9 The continuous time case

- Section 9
  - Section 9.1
    - Section 12.3.1
    - Section **??**
    - Section 9.1.3
    - Section **??**
  - Section 9.2

9.1 The continuous time Fourier transform

9.1.1 Definition

We saw above that any discrete sequence can be expressed exactly as an infinite sum of complex exponentials. The same kind of result exist for continuous time signals. Any $x(t)$ can be expressed as the Fourier integral

$$x(t) = \int_{-\infty}^{+\infty} X(f)\, e^{j2\pi ft}\, \mathrm{d}f,$$

where

$$X(f) = \int_{-\infty}^{+\infty} x(t)\, e^{-j2\pi ft}\, \mathrm{d}t.$$

The Fourier transform exists if the three sufficient conditions of Dirichlet are verified:

1. $x(t)$ possesses a finite number of discontinuities on any finite interval,
2. $x(t)$ possesses a finite number of maxima and minima on any finite interval,
3. $x(t)$ is absolutely integrable, that is
$$\int_{-\infty}^{+\infty} |x(t)|\, \mathrm{d}t < +\infty.$$
Indeed, if $x(t)$ is absolutely integrable, then
$$\int_{-\infty}^{+\infty} |x(t)\, e^{-j2\pi ft}|\, \mathrm{d}t < \int_{-\infty}^{+\infty} |x(t)|\, \mathrm{d}t < +\infty$$
(since $|x(t)\, e^{j2\pi ft}| = |x(t)|\, |e^{j2\pi ft}| < |x(t)|$).

9.1.2 Example - The Fourier transform of a rectangular pulse

Example 1. *rectangular pulse. We denote* $\text{rect}_T(t)$ *the rectangular pulse defined by*

$$\text{rect}_T(t) = \begin{cases} 1 & \text{if } t \in [-T/2, T/2], \\ 0 & \text{elsewhere.} \end{cases}$$

We look for the Fourier transform of $x(t) = A\text{rect}_T(t)$*. It is enough to write down the definition of the Fourier transform:*

$$X(f) = \text{FT}\{A\text{rect}_T(t)\} = A\int_{-T/2}^{T/2} e^{-j2\pi ft}\text{d}t,$$

that is

$$X(f) = A\left[\frac{e^{-j2\pi ft}}{-j2\pi f}\right]_{-\frac{T}{2}}^{\frac{T}{2}} = A\frac{1}{j2\pi f}\left[e^{j\pi fT} - e^{-j\pi fT}\right]$$

so that finally

$$\boxed{X(f) = AT\frac{\sin(\pi fT)}{\pi fT} \triangleq AT\,\text{sinc}\,(\pi fT).} \tag{9.1}$$

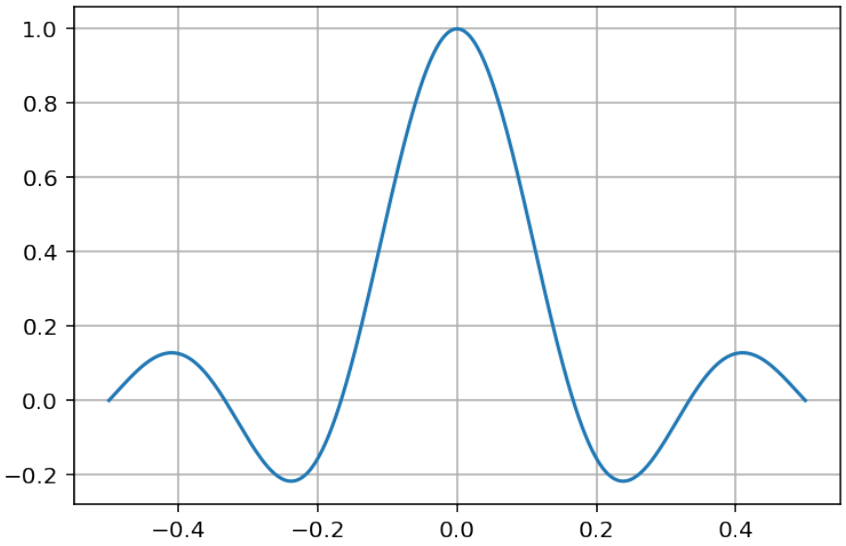
where $\text{sinc}\,(.)$ *is called a* `cardinal sinus`*. We note that this Fourier tyransform is peal and even. We will see later that this property is true for the Fourier transforms of all real and even signals. The function* $\text{sinc}\,(\pi fT)$ *vanishes for* $\pi fT = k\pi$*, that is for* $f = k/T$*; except for* $k = 0$*, since* $\text{sinc}\,(x) = 1$ *for* $x \to 0$*.*

Let us look at this sinc function (you may play with several values of the width):

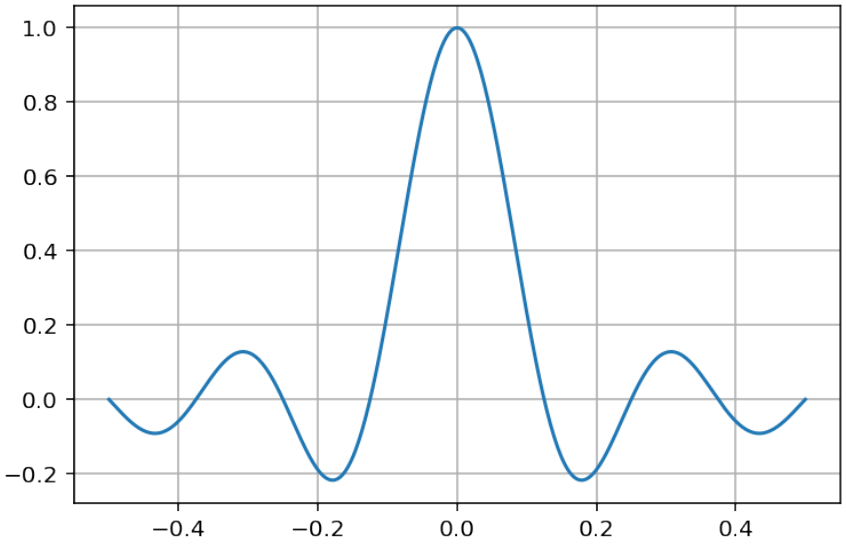
```
%matplotlib inline
def sinc(x):
    if isinstance(x,(int,float)): x=[x]
    x=np.array(x)
    out=np.ones(np.shape(x))
    I=np.where(x!=0)
    out[I]=np.sin(x[I])/x[I]
    return out

def dsinc(x,L):    # This is the "discrete time" cardinal sinus
    if isinstance(x,(int,float)): x=[x]
    x=np.array(x)
    out=np.ones(np.shape(x))
    I=np.where(x!=0)
    out[I]=np.sin(x[I])/(L*np.sin(x[I]/L))
    return out

N=1000
f=np.linspace(-0.5,0.5,400)
plt.plot(f,sinc(pi*6*f))
plt.grid(b=True)
```

Playing with values using a slider.

```
N=1000
f=np.linspace(-0.5,0.5,400)
out = widgets.Output()

#------ Callbacks des widgets ----------------
@out.capture(clear_output=True, wait=True)
def pltsinc(value):
```




```
    #clear_output(wait=True)
    T = s.value
    plt.plot(f,sinc(pi*T*f))
    plt.grid(b=True)
    plt.show()
s=widgets.FloatSlider(min=0, max=20, step=0.1, value=8)
pltsinc('Width')
s.observe(pltsinc,'value')
display(widgets.VBox([s, out]))
#alternatively
#interact(pltsinc,value=fixed(1), T=[0.1,10,0.1])
```



Widget Javascript not detected. It may not be installed or enabled properly.

The integral of $\text{sinc}(\pi fT)$ – Using the fact the Dirichlet integral is

$$\int_0^{+\infty} \text{sinc}(x)\mathrm{d}x = \frac{\pi}{2},$$

the symmetry of sinc(), and a change of variable, we obtain that

$$\int_{-\infty}^{+\infty} T\text{sinc}(\pi fT)\mathrm{d}f = 1.$$

It is now useful to look at the limit cases. - First, let $T \to +\infty$, that is let the rectangular pulse tends to a constant value. Its Fourier transform, $T\text{sinc}(\pi fT)$ tends to a mass on zero, since all the zero crossings occurs at zero. Furthermore, the amplitude is proportionnal to T and then goes to infinity. Hence, the Fourier transform of a constant is a mass with infinite amplitude, located at 0. As we noted above, the integral of $T\text{sinc}(\pi fT)$ equals to 1, which implies that the integral of this mass at zero is 1. This Fourier transform is not a function in the classical sense, but a distribution, see also the Encyclopedia of mathematics. In fact, it is the generalization of the Dirac δ function we had in discrete time.It is called *Dirac distribution* (or function) and we end with the following pair

$$1 \rightleftharpoons \delta(f)$$

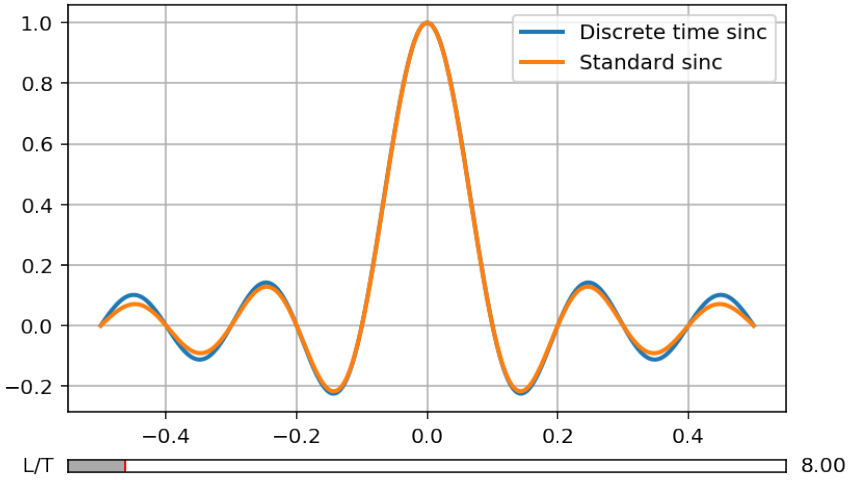
- Second, consider a rectangular pulse with amplitude $1/T$ and width T. When $T \to O$, this pulse tends to a Dirac distribution, a mass at zero, with infinite amplitude but also with a unit integral. By the Fourier transform of a rectangular pulse (9.1), we obtain that the Fourier transform of a Dirac function is a unit constant

$$\delta(t) \rightleftharpoons 1$$

```
%matplotlib tk
from matplotlib.widgets import Slider

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2, left=0.1)

slider_ax = plt.axes([0.1, 0.1, 0.8, 0.02])
slider = Slider(slider_ax, "L/T", 0, 100, valinit=8, color='#AAAAAA')
L=10
f=np.linspace(-0.5,0.5,400)

line, = ax.plot(f,dsinc(pi*L*f,L), lw=2,label="Discrete time sinc")
line2, = ax.plot(f,sinc(pi*L*f), lw=2,label="Standard sinc")
#line2 is in order to compare with the "true" sinc
ax.grid(b='on')
ax.legend()

def on_change(L):
    line.set_ydata(dsinc(pi*L*f,L))
    line2.set_ydata(sinc(pi*L*f))


slider.on_changed(on_change)
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/cbook/__init__.py:424: Matplotlib
Passing one of 'on', 'true', 'off', 'false' as a boolean is deprecated; use an actu
  warn_deprecated("2.2", "Passing one of 'on', 'true', 'off', 'false' as a "
```

```
0
```



9.1.3 Table of Fourier transform properties

This table is adapted and reworked from Dr Chris Jobling's resources, see this page. Many pages give tables and proofs of Fourier transform properties or Fourier pairs, e.g.: - Properties of the Fourier Transform (Wikpedia), - thefouriertransform.com,
- Wikibooks: Engineering Tables/Fourier Transform Properties - Fourier Transfom—WolframMathworld.

	Name	$x(t)$	$X(f)$
1	Linearity	$\sum_i a_i x_i(t)$	$\sum_i a_i X_i(f)$
2	Duality	$x(-f)$	$X(t)$
3.	Time and frequency scaling	$x(\alpha t)$	$\frac{1}{\lvert\alpha\rvert} S\left(\frac{f}{\alpha}\right)$
4.	Time shifting	$x(t-t_0)$	$e^{-j2\pi f t_0} X(f)$
5.	Frequency shifting	$e^{j2\pi f_0 t} x(t)$	$X(f-f_0)$
7.	Frequency differentiation	$(-jt)^k x(t)$	$\frac{d^k}{df^k} X(f)$
8.	Time integration	$\int_{-\infty}^{t} f(t)\mathrm{d}t$	$\frac{X(f)}{j2\pi f} + X(0)\delta(f)$
9.	Conjugation	$s^*(t)$	$S^*(-f)$
10.	Time convolution	$x_1(t) * x_2(t)$	$X_1(f)X_2(f)$
11.	Frequency convolution	$x_1(t)x_2(t)$	$X_1(f) * X_2(f)$
12.	Sum of x(t)	$\int_{-\infty}^{\infty} x(t)\mathrm{d}t$	$X(0)$
13.	Area under$X(f)$	$f(0)$	$\int_{-\infty}^{\infty} X(f)\,df$
15.	Parseval's theorem	$\int_{-\infty}^{\infty} \lvert x(t)\rvert^2 \mathrm{d}t$	$\int_{-\infty}^{\infty} \lvert X(f)\rvert^2 \mathrm{d}f.$

Property 1. *This property enables to express the Fourier transform of a delayed signal as a function of the Fourier transform of the initial signal and a delay term:*

$$x(t-t_0) \rightleftharpoons X(f)e^{-j2\pi f t_0}.$$

Proof. This property can be obtained almost immediately from the definition of the Fourier transform:

$$\mathrm{FT}\{x(t-t_0)\} = \int_{-\infty}^{+\infty} x(t-t_0)\, e^{-j2\pi f t}\, \mathrm{d}t;$$

Noting that $e^{-j2\pi f t} = e^{-j2\pi f(t-t_0)}e^{-j2\pi f t_0}$, we obtain

$$\mathrm{FT}\{x(t-t_0)\} = \int_{-\infty}^{+\infty} x(t-t_0)e^{-j2\pi f(t-t_0)}e^{-j2\pi f t_0}\mathrm{d}t,$$

that is

$$\mathrm{FT}\{x(t-t_0)\} = e^{-j2\pi f t_0}\int_{-\infty}^{+\infty} x(t-t_0)e^{-j2\pi f(t-t_0)}\mathrm{d}t = e^{-j2\pi f t_0}X(f).$$

□

9.1.4 Symmetries of the Fourier transform.

Time domain	Frequency domain
real	hermitian(real=even, imag=odd modulus=even, phase=odd)
imaginary	anti-hermitian(real=odd, imag=even modulus=even, phase=odd)
even	even
odd	odd
real and even	real and even (i.e. cosine transform)
real and odd	imaginary and odd (i.e. sine transform)
imaginary and even	imaginary and even
imaginary and odd	real and odd

(table adapted from cv.nrao.edu)

9.2 Dirac impulse, representation formula and convolution

9.2.1 Dirac impulse

Recall that the Dirac impulse $\delta(t)$ satisfies

$$\delta(t) = \begin{cases} 0 & \text{if } t \neq 0, \\ +\infty & \text{for } t = 0, \end{cases}$$

and is such that

$$\int_{-\infty}^{+\infty} \delta(t)\mathrm{d}t = 1.$$

9.2.2 Representation formula

The Dirac impulse plays the role of an indicator function. In particular, we have

$$x(t)\delta(t-t_0) = x(t_0)\delta(t-t_0).$$

Consequently,

$$\int_{-\infty}^{+\infty} x(t)\delta(t-t_0)\mathrm{d}t = x(t_0)\int_{-\infty}^{+\infty}\delta(t-t_0)\mathrm{d}t = x(t_0).$$

Therefore, we always have

$$\begin{cases} x(t) = \int_{-\infty}^{+\infty} x(\tau)\delta(t-\tau)\mathrm{d}\tau \\ \text{with } x(\tau) = \int_{-\infty}^{+\infty} x(t)\delta(t-\tau)\mathrm{d}t. \end{cases}$$

This is nothing but the continuous-time version of the *representation formula*.

The set of distributions $\{\delta_\tau(t) : \delta(t-\tau)\}$, forms an orthonormal basis and $x(\tau)$ can be viewed as a coordinate of $x(t)$ on this basis. Indeed, the scalar product between $x(t)$ and $\delta_\tau(t)$ is nothing but

$$x(\tau) = < x(t), \delta_\tau(t) > = \int_{-\infty}^{+\infty} x(t)\delta(t-\tau)\mathrm{d}t,$$

and $x(t)$ is then given as the sum of the basis functions, weighted by the associated coordinates:

$$x(t) = \int_{-\infty}^{+\infty} x(\tau)\delta(t-\tau)\mathrm{d}\tau.$$

Following the same approach as in the discrete case, we define the *impulse response* $h(t)$ as the output of a linear invariant system to a Dirac impulse. By linearity, the output of the system to any input $x(t)$, expressed using the representation formula, is

$$\boxed{y(t) = \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)\mathrm{d}\tau = [x*h](t).}$$

This is the time-continuous *convolution* between x and h, denoted $[x*h](t)$. It enables to express the output of the filter using only the input and the impulse response. This shows the importance of the impulse response as a description of the system. The other notions we studied in the discrete case, namely transfer function, Plancherel and Parseval theorems, etc, extends straightforwardly to the continuous case.

10 Periodization, discretization and sampling

10.1 Periodization-discretization duality

10.1.1 Relation between Fourier series and Fourier transform

Remember that we defined the Fourier transform as the limit of the Fourier series of periodic signal, when the period tends to infinity. A periodic signal can also be viewed as the repetition of a basic pattern. This enables to give a link between Fourier series and transform. Let $x(n)$ be a periodic function with period L_0. Then

$$\boxed{x(n) = \sum_{m=-\infty}^{+\infty} x_{L_0}(n - mL_0),} \tag{10.1}$$

where $x_{L_0}(n)$ is the basic pattern with length L_0. $x(n)$ being periodic, it can be expressed using a Fourier series, as

$$x(n) = \sum_{n=0}^{L_0-1} c_k e^{j2\pi k f_0 n},$$

where $f_0 = 1/L_0$ and

$$c_k = \frac{1}{L_0} \sum_{[L_0]} x_{L_0}(n) e^{-j2\pi k f_0 n}.$$

From this relation, we immediately have

$$c_k = \frac{1}{L_0} X_{L_0}(k f_0), \tag{10.2}$$

where $X_{L_0}(f)$ is the Fourier transform of the pattern $x_{L_0}(n)$. Hence

$$\boxed{x(n) = \sum_{m=-\infty}^{+\infty} x_{L_0}(n - mL_0) = \frac{1}{L_0} \sum_{k=0}^{L_0-1} X_{L_0}(k f_0) e^{j2\pi k f_0 n}.} \tag{10.3}$$

From that, we deduce that the Fourier transform of $x(n)$ writes

$$\mathrm{FT}\{x(n)\} = \frac{1}{L_0} \sum_{k=0}^{L_0-1} X_{L_0}(k f_0) \mathrm{FT}\left\{e^{j2\pi k f_0 n}\right\},$$

that is

$$\boxed{X(f) = \mathrm{FT}\{x(n)\} = \frac{1}{L_0} \sum_{k=0}^{L_0-1} X_{L_0}(k f_0) \delta(f - k f_0)}. \tag{10.4}$$

10.1.2 Poisson summation formulas

Hence, we see that the Fourier transform of a periodic signal with period L_0 is constituted of a series of Dirac impulse, spaced by f_0, and whose weights are the Fourier transform of the initial pattern, taken at the respective frequencies.

Periodicity in the time domain yields spectral lines in the frequency domain.

Taking $x_{L_0}(n) = \delta(n)$, we obtain the first *Poisson's formula*:

$$\boxed{\sum_{m=-\infty}^{+\infty} \delta(n - mL_0) = \frac{1}{L_0} \sum_{k=0}^{L_0-1} e^{j2\pi k f_0 n}}. \tag{10.5}$$

The series of delayed Dirac impulses is called a *Dirac comb*. It is often denoted

$$w_{L_0}(n) = \sum_{m=-\infty}^{+\infty} \delta(n - mL_0). \tag{10.6}$$

Taking the Fourier transforms of the two sides of (10.5), we obtain

$$\boxed{\sum_{m=-\infty}^{+\infty} e^{j2\pi f m L_0 n} = \frac{1}{L_0} \sum_{k=0}^{L_0-1} \delta(f - k f_0)}; \tag{10.7}$$

that is the second *Poisson's formula*:

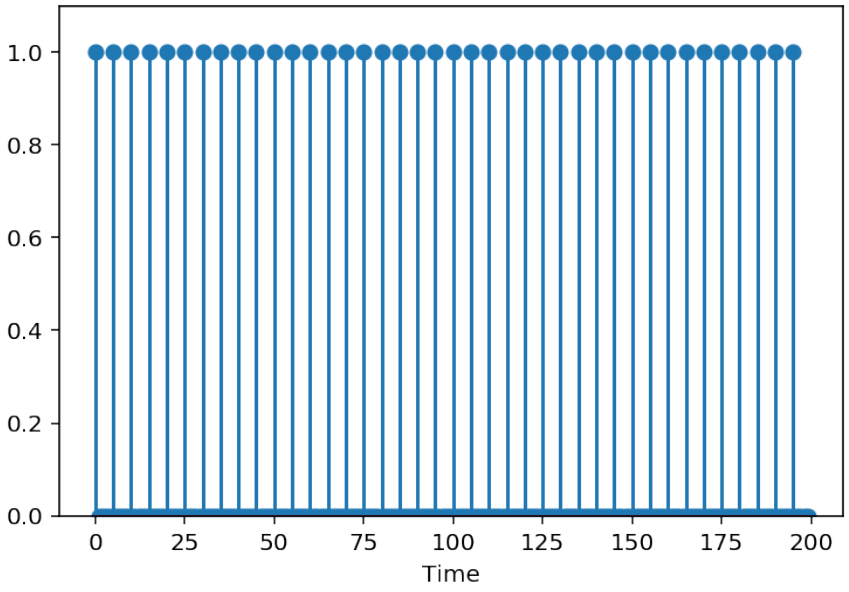
$$\boxed{\sum_{m=-\infty}^{+\infty} \delta(n - mL_0) \rightleftharpoons \frac{1}{L_0} \sum_{k=0}^{L_0-1} \delta(f - k f_0)}. \tag{10.8}$$

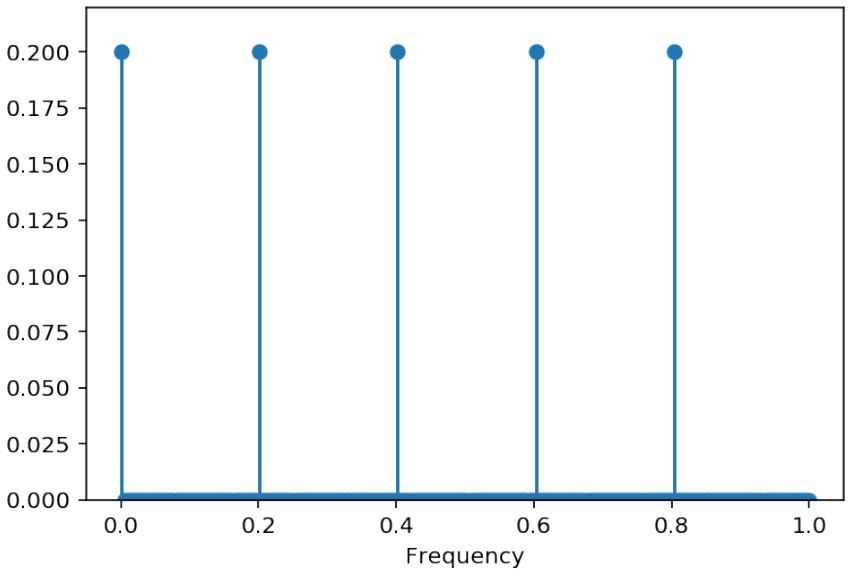
This last relation shows that the Fourier transform of a Dirac comb is also a Dirac comb, these two combs having an inverse spacing.

Exercise 4. *Let us check this numerically. This is very easy: define a Dirac comb, take its Fourier transform using the* `fft` *function, and look at the result.*

```
## DO IT YOURSELF...
#DiracComb=
#DiracComb_f=fft(DiracComb)
#etc
```

```
N = 200
L0 = 5
DiracComb = np.zeros(N)
DiracComb[::L0] = 1
DiracComb_f = fft(DiracComb)
plt.stem(DiracComb)
plt.ylim([0, 1.1])
plt.xlabel("Time")
plt.figure()
f = np.linspace(0, 1, N)
plt.stem(f, 1 / N *
         abs(DiracComb_f))  # Actually there is a factor N in the fft
_ = plt.ylim([0, 1.1 * 1 / L0])
plt.xlabel("Frequency")
```


1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175
200
Time


0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0.0
0.2
0.4
0.6
0.8
1.0
Frequency

```
Text(0.5, 0, 'Frequency')
```

We may now go back to the exploration of the links between Fourier series and transform, using the second Poisson formula (10.8).

Convolution with a delayed Dirac impulse - Let us first look at the result of the convolution of any function with a delayed Dirac impulse: let us denote $\delta_{n_0}(n) = \delta(n - n_0)$. The convolution $[x * \delta_{n_0}](n)$ is eq given by

$$[x * \delta_{n_0}](n) = \sum_m x(m)\delta_{n_0}(n-m) \tag{10.9}$$
$$= \sum_m x(m)\delta(n-m-n_0) \tag{10.10}$$
$$= x(n-n_0) \tag{10.11}$$
$$\tag{10.12}$$

where the last relation follows by the representation formula. Hence

Convolution with a delayed Dirac delays the signal.

This has a simple and direct filtering interpretation. Indeed, if a filters has for impulse response a delayed Dirac impulse, then this means that it is a pure delaying filter. Then to an input $x(n)$ corresponds an output $x(n-n_0)$.

Convolution with a Dirac comb - By linearity, the convolution of any signal $x_L(n)$ of length L with a Dirac comb results in the sum of the delayed responses:

$$x(n) = [x_L * w_{L_0}](n) = \left[x_L * \sum_k \delta_{kL_0}\right](n) \tag{10.13}$$
$$= \sum_k [x_L * \delta_{kL_0}](n) \tag{10.14}$$
$$= \sum_k x_L(n - kL_0). \tag{10.15}$$
$$\tag{10.16}$$

This is nothing but the expression of a periodic signal. If L_0 is larger than the support L of $x_L(n)$, then $x(n)$ is simply the repetition, with a period L_0, of the pattern $x_L(n)$.

Convolution with a Dirac comb periodizes the signal.

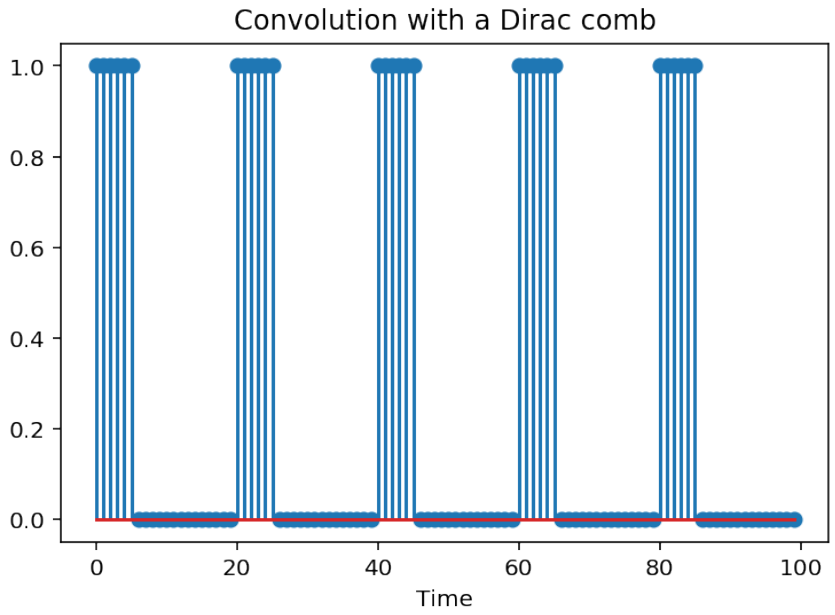
Exercise 5. *Let us check this with some simple Python commands: create a Dirac comb, a test signal (e.g.) a rectangular pulse, convolve the two signals and plot the result.* Experiment with the value L of the length of the test signal.

```
# DO IT YOURSELF!
#DiracComb=
#pulse=
#...
#z=np.convolve(DiracComb,pulse)
#plt.stem(...)
```

```
N=400; L0=20; L=6 # L is the length of the pulse
DiracComb=np.zeros(N)
DiracComb[::L0]=1
pulse=np.zeros(40); pulse[0:L]=1 #or range(L) # <<--
```

```
z=np.convolve(DiracComb,pulse)
plt.stem(z[0:100])
plt.title('Convolution with a Dirac comb')
plt.xlabel('Time')
```

```
Text(0.5, 0, 'Time')
```



We see that the convolution with the Dirac comb effectively periodizes the initial pattern. In the case where the support L of the pulse if larger than the period L_0 of the comb, then the result presents *aliasing* between consecutive patterns (but the resulting signal is still periodic).

Effect in the frequency domain - In the frequency domain, we know, by the Plancherel theorem, that the product of signals results in the convolution of their Fourier transforms (and *vice versa*). As a consequence,

$$x(n) = [x_L * w_{L_0}](n) \rightleftharpoons X_L(f).\mathrm{FT}\{w_{L_0}(n)\}.$$

Since the Fourier transform of a Dirac comb is also a Dirac comb, we obtain that

$$x(n) = [x_L * w_{L_0}](n) \rightleftharpoons X_L(f).\frac{1}{L_0} w_{\frac{1}{L_0}}(f),$$

or

$$X(f) = X_L(f).\frac{1}{L_0} w_{\frac{1}{L_0}}(f) = \frac{1}{L_0}\sum_k X_L(kf_0)\delta(f-kf_0),$$

with $f_0 = 1/L_0$. We see that the Fourier transform of the periodized signal is the product of the Fourier transform of the initial pattern with a Dirac comb in frequency. Hence, periodization in the time domain results in a discretization of the frequency axis, yielding a Fourier transform constituted of spectral lines. Observe that the amplitudes of the spectral lines coincide with the Fourier series coefficients. hence it is immediate to find the Fourier series coefficients from the Fourier transform of the periodized pattern.

Periodization in the time domain results in a discretization in the frequency domain.

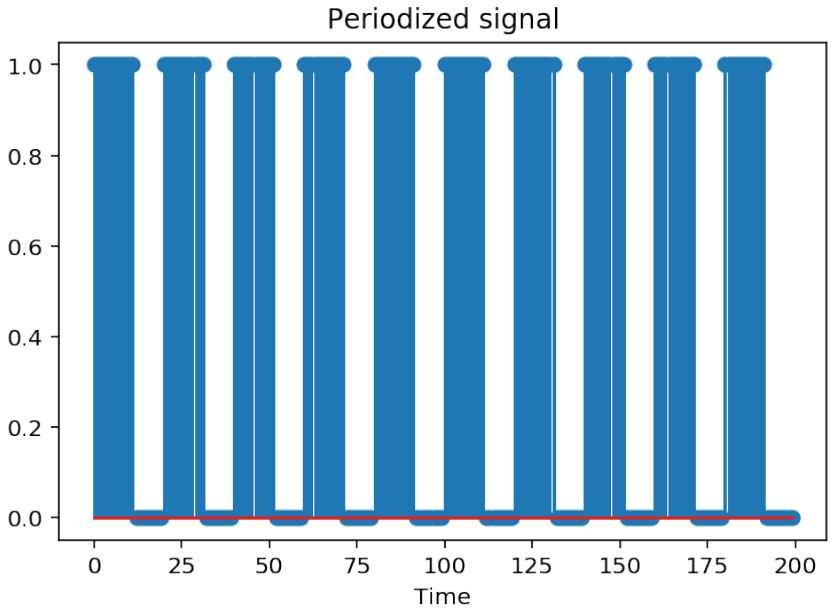
Exercise 6. *Continue the exercise 5 by an analysis of what happens in the Fourier domain: compute the Fourier transforms of the original and periodized signals and compare them on the same plot. The Fourier transform of the periodized signal should be computed* without *zero padding, ie exactly on N points.*

You will have to introduce a factor to account for the fact that there is more signal in the periodized one yan in the initial - the factor to consider is simply the number of periods.

```
#
N = 200
MM = 2000  #for zero padding
plt.figure()
f = np.linspace(0, 1, MM)
fn = np.linspace(0, 1, N)
#
# FILL IN HERE
#
plt.title('Fourier transform of original and periodized pulses')
_ = xlabel('Frequency')
```


```
%matplotlib inline
N = 200
L0 = 20
L = 12  # L is the length of the pulse
DiracComb = np.zeros(N)
DiracComb[::L0] = 1
pulse = np.zeros(40)
pulse[0:L] = 1  #exp(-0.3*arange(L))
z = np.convolve(DiracComb, pulse)
plt.stem(z[0:200])
plt.title('Periodized signal')
plt.xlabel('Time')
```

```
#
MM = 1000
plt.figure()
f = np.linspace(0, 1, MM)
fn = np.linspace(0, 1, N)
plt.plot(f, 10 * abs(fft(pulse, MM)), label="FT original signal")
plt.stem(fn, abs(fft(z, N)), '-or', label="FT periodized signal")
plt.legend()
plt.title('Fourier transform of original and periodized pulses')
_ = xlabel('Frequency')
```

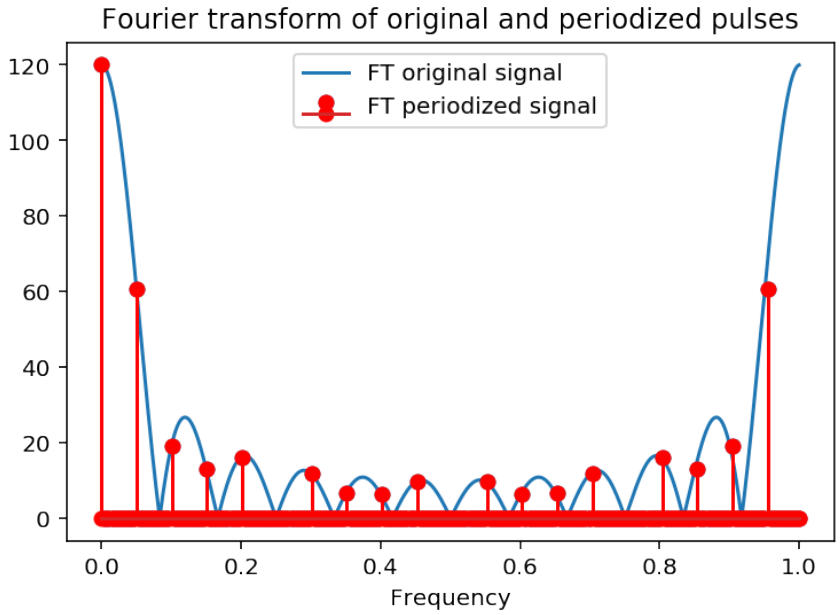


10.2 The Discrete Fourier Transform

From the periodization-discretization duality, we can return to the notion of Discrete Fourier Transform (3.1) we introduced in section 3.1. Recall that the DFT is Fourier transform that appears when we assume that the signal is periodic out of the observation interval (an other option is to assume that the signal is zero outside of the observation interval, and this leads to the discrete-time Fourier transform). Since the signal is considered periodic, it can be expressed as a Fourier series, and this leads to the pair of formulas recalled here for convenience

$$\begin{aligned} x(n) &= \sum_{k=0}^{N-1} X(k) e^{j2\pi \frac{kn}{N}} \\ \text{with } X(k) &= \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{kn}{N}}. \end{aligned} \tag{10.17}$$

In this section, we show that the DFT can also be viewed as a sampled version of the discrete-time Fourier transform or as a simple change of basis for signal representation. We indicate that the assumption of periodized signal in the time-domain implies some caution when studying some properties of the DFT, namely time shifts or convolution.



10.2.1 The Discrete Fourier Transform: Sampling the discrete-time Fourier transform

Given what we learned before, it is very easy to see that the DFT is indeed a sampled version of the discrete-time Fourier transform. We know that periodizing a signal can be interpreted as a convolution of a pattern with a Dirac comb. In turn, this implies in the frequency domain a multiplication of the Fourier transform of the initial pattern with a Dirac comb: if we denote $x_0(n)$ the signal for $n \in [0,N)$,

$$x(n) = [x_0 * w_N](n)$$

and

$$\begin{aligned} X(f) &= X_0(f).\frac{1}{N} w_{\frac{1}{N}}(f) && (10.18)\\ &= X_0(f).\frac{1}{N}\sum_{k=0}^{N-1}\delta\left(f-\frac{k}{N}\right) && (10.19)\\ &= \frac{1}{N}\sum_{k=0}^{N-1} X_0\left(\frac{k}{N}\right)\delta\left(f-\frac{k}{N}\right) && (10.20)\\ & && (10.21) \end{aligned}$$

Then, the expression of $x(n)$ as an inverse Fourier transform becomes

$$\begin{aligned} x(n) &= \int_{[1]} X(f) e^{j2\pi fn} \mathrm{d}f && (10.22)\\ &= \frac{1}{N}\sum_{k=0}^{N-1} X_0\left(\frac{k}{N}\right)\int_{[1]} e^{j2\pi fn}\delta\left(f-\frac{k}{N}\right)\mathrm{d}f && (10.23)\\ &= \frac{1}{N}\sum_{k=0}^{N-1} X_0\left(\frac{k}{N}\right) e^{j2\pi\frac{kn}{N}} && (10.24) \end{aligned}$$

since the integration with the Dirac distribution yields the value of the function for the argument where the Dirac is nonzero. It simply remains to note that

$$X_0\left(f=\frac{k}{N}\right) = \sum_{n=-\infty}^{+\infty} x_0(n)e^{-j2\pi\frac{kn}{N}} \tag{10.25}$$

$$= \sum_{n=0}^{N-1} x_0(n)e^{-j2\pi\frac{kn}{N}} \tag{10.26}$$

$$\tag{10.27}$$

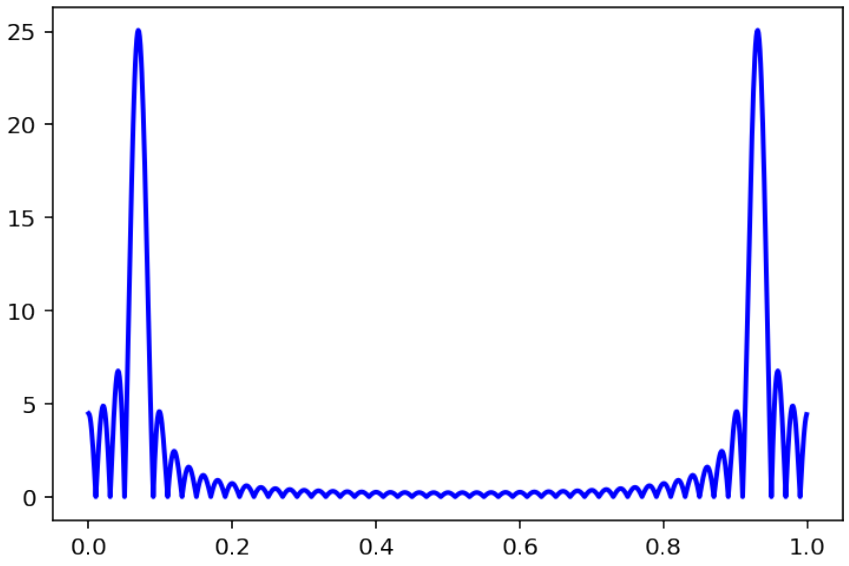
since $x(n) = x_0(n)$ on the interval $[0,N)$. Denoting $X(k) = X_0\left(f=\frac{k}{N}\right)$, we arrive at the formulas (10.17) for the DFT.

We illustrate this numerically. We look at the Fourier transform of a sine wave, with and without zero-padding. In the first case, we obtain something that represents the discrete-time Fourier transform, and which exhibits the consequence of the time-limitation of the signal. In the second case, we obtain the samples of the DFT.

```
##
# experiments on DFT: the DFT as sampled FT
N = 50  # Fourier resolution: 1/N
fo = 0.07  # not on the Fourier grid
t = arange(N)
s = sin(2 * pi * fo * t)
Sz = fft(s, 1000)
f = arange(1000) / 1000
plot(f, abs(Sz), lw=2, color="blue")
S = fft(s)
f2 = arange(N) / N
stem(f2, abs(S), lw=2, linefmt='g-', markerfmt='go')
plot(f2, abs(S), 'r-')
xlabel("Frequencies")

# Here we play with annotations and arrows...
annotate(
    "True Fourier transform \n(zero-padded data)",
    xy=(0.075, 21),
    xytext=(0.11, 23),
    arrowprops=dict(
        arrowstyle="->",
        color="blue",
        connectionstyle="arc3,rad=0.2",
        shrinkA=5,
        shrinkB=10))

annotate(
    "Samples on the DFT\n grid",
    xy=(0.08, 15),
    xytext=(0.13, 17),
    arrowprops=dict(
        arrowstyle="->",
        color="green",
        connectionstyle="arc3,rad=0.2",
        shrinkA=5,
        shrinkB=10))
annotate(
    "Apparent FT..",
    xy=(0.09, 10),
```

```
    xytext=(0.16, 6.5),
    arrowprops=dict(
        arrowstyle="->",
        color="red",
        connectionstyle="arc3,rad=-0.0",
        shrinkA=15,
        shrinkB=10))
xlim([0, 0.3])
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-2-b4669398e50a> in <module>()
     10 S = fft(s)
     11 f2 = arange(N) / N
---> 12 stem(f2, abs(S), lw=2, linefmt='g-', markerfmt='go')
     13 plot(f2, abs(S), 'r-')
     14 xlabel("Frequencies")

TypeError: stem() got an unexpected keyword argument 'lw'
```



Thus we note that without caution and analysis, it is easy to be mistaken. A zero-padding – i.e. compute the FT padded with zeros, often enable to avoid bad interpretations.

10.2.2 The DFT as a change of basis

A signal known on N samples can be seen as a vector in a N-dimensional space. Of course it can be written as

$$\mathbf{x} = \begin{bmatrix} x(0) \\ x(1) \\ \vdots \\ x(N-1) \end{bmatrix} = x(0) \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + x(1) \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} + \ldots + x(N-1) \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

The vectors of complex exponentials

$$\mathbf{e_k} = \frac{1}{\sqrt{N}} \left[1, e^{-j2\pi\frac{k}{N}}, \ldots, e^{-j2\pi\frac{kl}{N}}, \ldots, e^{-j2\pi\frac{k(N-1)}{N}} \right]^T$$

also for a basis of the same space. It is a simple exercise to check that $\mathbf{e_k}^T \mathbf{e_l} = \delta(k-l)$. Thus it is possible to express $\mathbf{x}$ in the basis of complex exponentials. The coordinate $X(k)$ of $\mathbf{x}$ on the vector $\mathbf{e_k}$ is given by the scalar product $\mathbf{e_k}^+\mathbf{x}$, where $^+$ denotes transposition and complex conjugation. If we denote

$$\mathbf{F} = \left[\mathbf{e_0}, \mathbf{e_1}, \ldots \mathbf{e_{N-1}} \right]$$

the **Fourier matrix**, then we can note that $\mathbf{F}^+\mathbf{F} = \mathbf{1}$, which means that $\mathbf{F}$ is a unitary matrix – and that in particular $\mathbf{F}^{-1} = \mathbf{F}^+$. Then, the change of basis to the basis of exponentials can be expressed as

$$\mathbf{X} = \mathbf{F}^+\mathbf{x} \tag{10.28}$$

and $\mathbf{x}$ can be expressed in terms of the $X(k)$ as

$$\mathbf{x} = \mathbf{F}\mathbf{X}. \tag{10.29}$$

Developing line k of (10.28), we obtain

$$X(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n) e^{-j2\pi\frac{kn}{N}},$$

and developing line n of (10.29), we obtain

$$x(n) = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X(k) e^{j2\pi\frac{kn}{N}}.$$

Up to a simple factor (let eg $X'(k) = \frac{1}{\sqrt{N}} X(k)$) we recover the formulas (10.17) of the DFT.

10.2.3 Time-shift property

... to be completed

10.2.4 Circular convolution

... to be completed

```
plt.rcParams['figure.figsize'] = (8, 6)
```

10.3 (Sub)-Sampling of time signals

Let us now turn to the analysis of sampling in the time domain. This topic is important because it has applications for the acquisition and digitization of analog world signals. Subsampling, or downsampling, has also applications in multirate filtering, which frequently occurs in coding problems. We begin with the subsampling of discrete time signals. With the Poisson formulas and the Plancherel theorem, the description of the process is quite easy. Subsampling simply consists in keeping one sample every say N_0 samples. This can be viewed succession of two operations 1. the product of our original signal with a Dirac comb of period N_0, 2. the discarding of the unwanted samples. Of course, we could limit ourselves to the second step, which is the only useful one for downsampling. However, the succession of the two steps is important to understand what happens in the Fourier domain.

Suppose that we have, at the beginning of step 2, a signal with useful samples separated by $N_0 - 1$ zeros. This signal is denoted $x_s(n)$ and its Fourier transform is $X_s(f)$, with $\mathtt{s}$ for 'sampled':

$$X_s(f) = \sum_n x_s(n) e^{-j2\pi fn}.$$

Taking into account that only the samples at indexes $n = kN_0$ are nonzeros, we may denote $x_d(k) = x_s(kN_0)$ ($\mathtt{d}$ for 'downsampled'), and make the change of variable $n = kN_0$

$$X_s(f) = \sum_k x_d(k) e^{-j2\pi fkN_0}.$$

Hence, we see that $X_s(f) = X_d(fN_0)$, that is also

$$X_d(f) = X_s\left(\frac{f}{N_0}\right).$$

The Fourier transform of the downsampled signal is simply a scaled version of the Fourier transform of the sampled signal. Hence, they contain the very same information. In order to understand what happens in the sampling/downsampling operation, we thus have to focus on the sampling operation, that is step 1. above. The sampled signal is

$$x_s(n) = x(n).w_{N_0}(n).$$

By Plancherel's theorem, we have

$$X_s(f) = [X * \mathrm{FT}\{w_{N_0}(n)\}](f) \tag{10.30}$$

$$= \left[X * \frac{1}{N_0} w_{\frac{1}{N_0}}\right](f) \tag{10.31}$$

$$= \frac{1}{N_0} \sum_k \left[X * \delta_{\frac{k}{N_0}}\right](f) \tag{10.32}$$

As in the discrete case, the continuous convolution with a Dirac comb results in the periodization of the initial pattern, that is

$$\boxed{X_s(f) = \frac{1}{N_0} \sum_k X(f - \frac{k}{N_0}).}$$

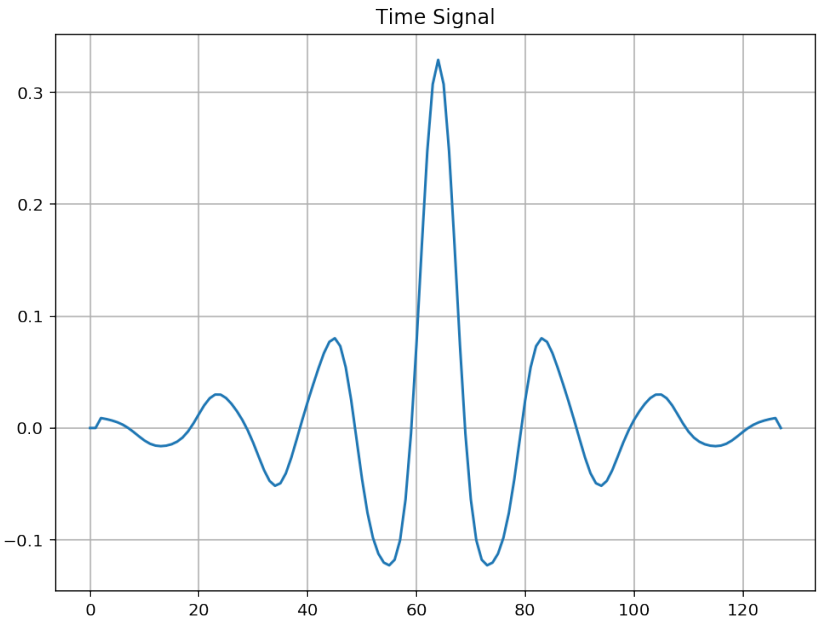
This is a fundamental result:

Sampling in the time domain yields periodization in the frequency domain.

10.4 Illustration 1

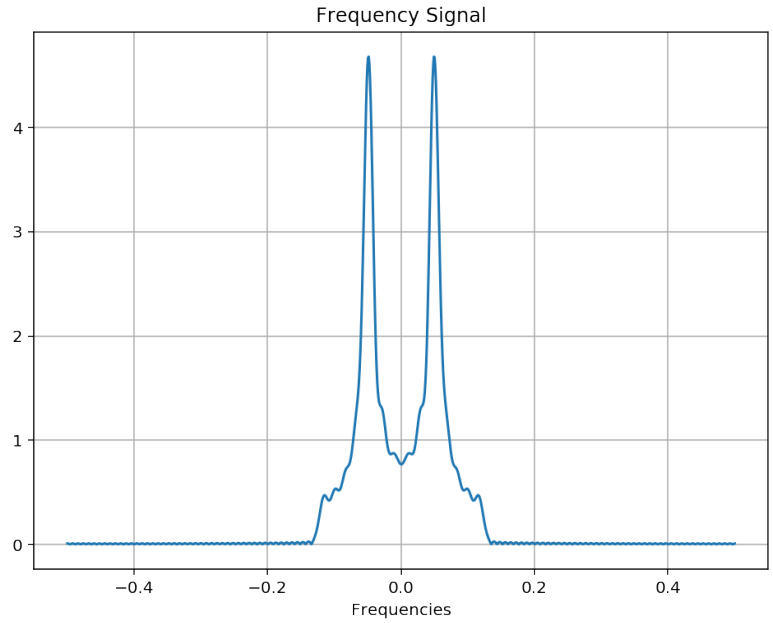
Let us illustrate this on our test signal:

```
loadsig = np.load("signal.npz")  #load the signal
x = loadsig["x"]
N = len(x)
#
M = 8 * N  #Used for fft computations
# Definition of vectors t and f
t = np.arange(N)
f = np.linspace(-0.5, 0.5, M)
# Plot time signal
plot(t, x)
title('Time Signal')
plt.grid(True)
plt.figure()
#plot frequency signal
xf = fftshift(fft(x, M))
plot(f, abs(xf))
title('Frequency Signal')
xlabel('Frequencies')
plt.grid(True)
```



We first define a subsampler function, that takes for argument the signal x and the subsampling factor k.

```
def subsampb(x, k, M=len(x)):
    """ Subsampling with a factor k
    Returns the subsampled signal and its Fourier transform"""
    xs = np.zeros(np.shape(x))
    xs[::k] = x[::k]
```

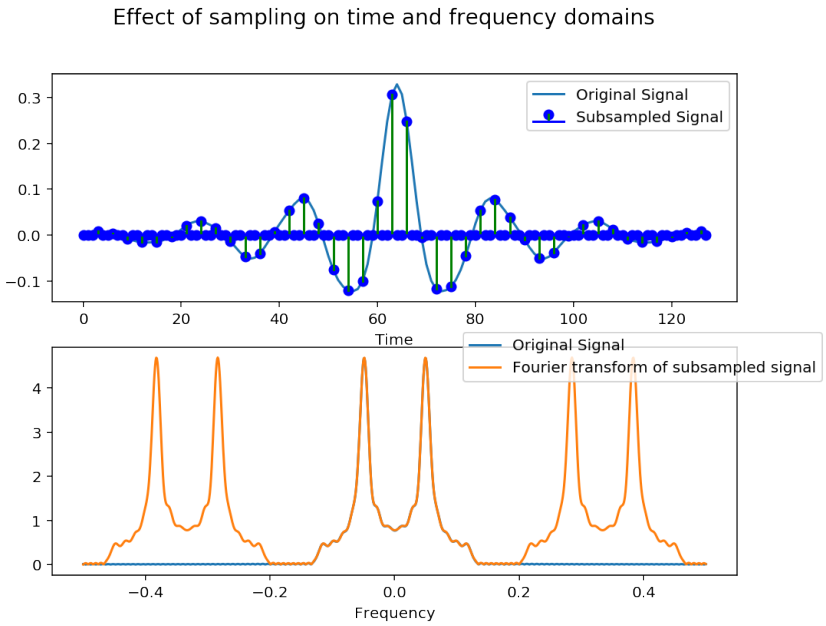



```
    xsf = fftshift(fft(xs, M))
    return (xs, xsf)
```

10.5 Illustration 2

```
%matplotlib inline

out = widgets.Output()

slide_k=widgets.IntSlider(min=1,max=8,value=3, description="Subsampling
    factor")

@out.capture(clear_output=True, wait=True)
def sampling_experiment(val):
    k = slide_k.value
    fig,bx=plt.subplots(2,1,figsize=(8,6))
    # clear_output(wait=True)
    bx[0].plot(t,x, label='Original Signal')
    (xs,xsf)=subsampb(x,k,M)
    bx[0].stem(t,xs,linefmt='g-',markerfmt='bo', basefmt='b-',label='
        Subsampled Signal')
    bx[0].set_xlabel('Time')
    bx[0].legend()
    #
    bx[1].plot(f,abs(xf), label='Original Signal')
    #xef=subsampb(x,k)[1]
    bx[1].plot(f,k*abs(xsf),label='Fourier transform of subsampled signal')
    # The factor k above takes into account the power lost by subsampling
    xlabel('Frequency')
    bx[1].legend(loc=(0.6,0.85))
    fig.suptitle("Effect of sampling on time and frequency domains",fontsize
        =14)
```

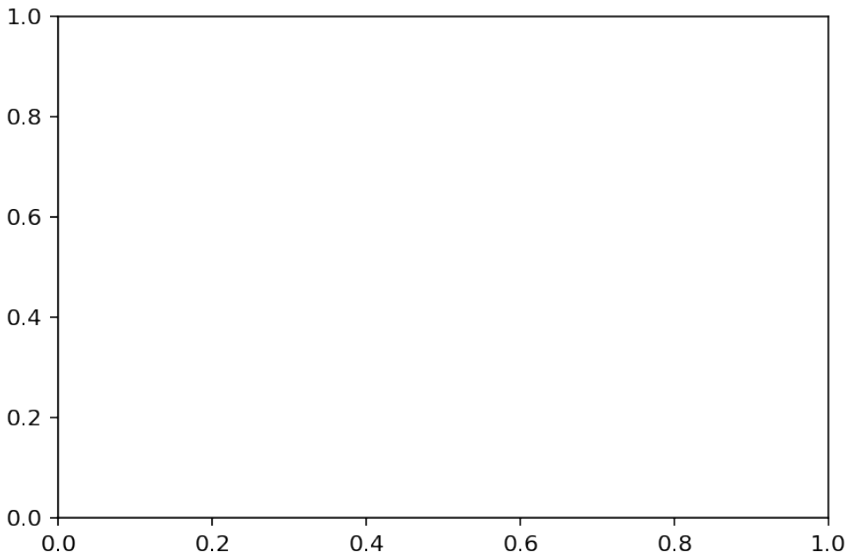
```
    #tight_layout()
    plt.show()

display(widgets.VBox([slide_k, out]))
sampling_experiment('')
slide_k.observe(sampling_experiment,'value')
```



matplotlib external figure version:

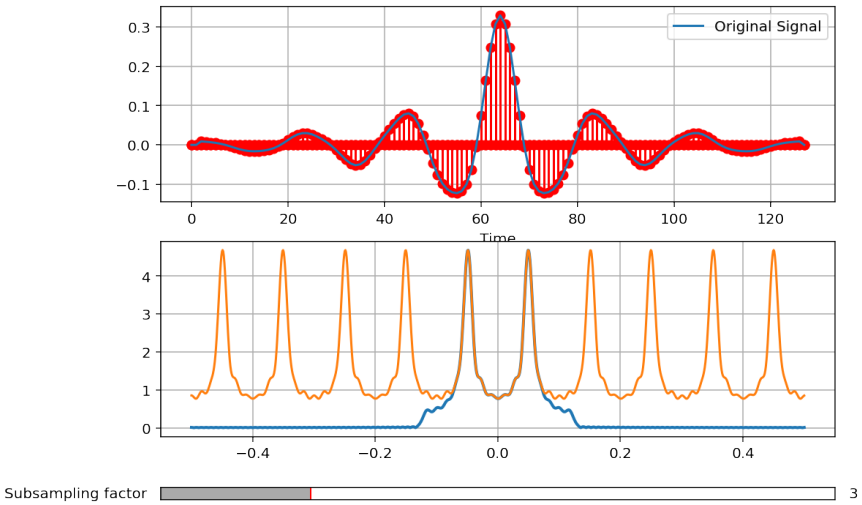
```
def plt_stem(t, x, *args, ax=gca(), **kwargs):
    xx = zeros(3 * len(x))
    xx[1:-1:3] = x
    xx = xx[:3 * len(x)]
    tt = np.repeat(t, 3)
    out = ax.plot(tt, xx, *args, **kwargs)
    return out
```

```
%matplotlib tk
from matplotlib.widgets import Slider

fig, ax = plt.subplots(2, 1)
fig.subplots_adjust(bottom=0.2, left=0.1)

slider_ax = fig.add_axes([0.1, 0.1, 0.8, 0.02])
slider = Slider(
    slider_ax,
    "Subsampling factor",
    1,
    10,
    valinit=3,
    color='#AAAAAA',
    valfmt='%0.0f')
L = 10
k = 5
(xs, xsf) = subsampb(x, k, M)
```




```
linexf, = ax[1].plot(f, abs(xf), lw=2)
linexf_update, = ax[1].plot(
    f, k * abs(xsf), label='Fourier transform of subsampled signal')

#markersx_update,stemsx_update,_=ax[0].stem(t,xs,linefmt='g-',markerfmt='bo
    ', basefmt='b-',label='Subsampled Signal')
linex_update, = plt_stem(
    t, x, '-or', ax=ax[0])  #ax[0].plot(t,xs, 'ob', label='Subsampled Signal
        ')
linex, = ax[0].plot(t, x, label='Original Signal')
ax[0].set_xlabel('Time')
ax[0].legend()

#line2, = ax.plot(f,sinc(pi*L*f), lw=2)
#line2 is in order to compare with the "true" sinc
ax[0].grid(b=True)
ax[1].grid(b=True)


def on_change(k):
    k = int(round(k))
    (xs, xsf) = subsampb(x, k, M)
    linexf_update.set_ydata(k * abs(xsf))
    xxs = zeros(3 * len(xs))
    xxs[1:-1:3] = xs
    linex_update.set_ydata(xxs)

slider.on_changed(on_change)
```

0



10.6 The sampling theorem

10.6.1 Derivation in the case of discrete-time signals

As a consequence, we will obtain a sufficient condition for the reconstruction of the original signal from its samples. Assume that $x(n)$ **is a real band-limited signal with a maximum frequency** B.

$$X(f) = 0 \text{ for } |f| > B$$

with $f \in [-\frac{1}{2}, \frac{1}{2}]$ for discrete time signals. Then, after sampling at rate f_s, the Fourier transform is the periodic summation of the original spectrum.

$$X_s(f) = f_s \sum_k X(f - k f_s). \tag{10.33}$$

```
%matplotlib inline
plt.figure(figsize=(7,2))
plt.plot(f,k*abs(xsf),label='Fourier transform of subsampled signal')
plt.xlim([-0.5,0.5])
_=plt.xticks([-1/2, -1/3, -0.16, 0, 0.16, 1/3, 1/2],
    ['$-\\frac{1}{2}$','-$f_s$','-$B$', '$0$', '$B$','$f_s$','$\\frac{1}{2}$
        '],
             fontsize=14)
```

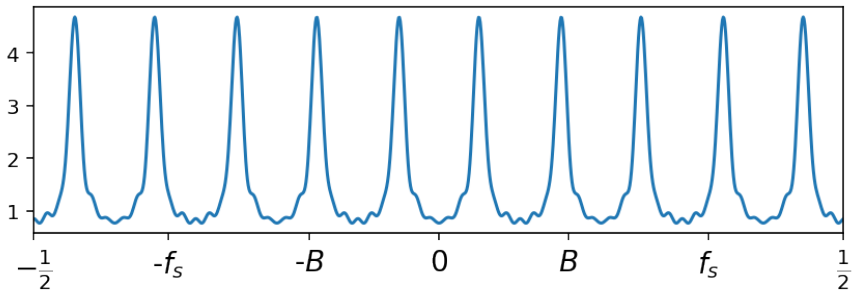
Hence, provided that there is no aliasing between consecutive images, it will be possible to retrieve the initial Fourier transform from this periodized signal. This is a fundamental result, which is known as the **Shannon-Nyquist theorem**, or **sampling theorem**.

In the frequency domain, this simply amounts to introduce a filter $H(f)$ that only keeps the frequencies in $[-f_s/2, f_s/2]$:

$$\begin{cases} H(f) = 1 & \text{for } |f| < f_s/2 \\ H(f) = 0 & \text{for } |f| > f_s/2 \end{cases}$$

in the interval $f \in [-\frac{1}{2}, \frac{1}{2}]$ for discrete time signals. Clearly, we then have

$$X_s(f).H(f) = f_s X(f)$$



and we are able to recover $X(f)$ up to a simple factor. Of course, since we recover our signal in the frequency domain, we can also get it in the time domain by inverse Fourier transform. By Plancherel's theorem, it immediately comes

$$x(n) = T_s[x_s * h](n),$$

with $T_s = 1/f_s$. A simple computation gives us the expression of the impulse response h as the inverse Fourier transform of a rectangular pulse of width f_s:

$$h(n) = \int_{[1]} \mathrm{rect}_{f_s}(f) e^{j2\pi fn} \mathrm{d}f \tag{10.34}$$

$$= \int_{-\frac{f_s}{2}}^{\frac{f_s}{2}} e^{j2\pi fn} \mathrm{d}f \tag{10.35}$$

$$= f_s \frac{\sin(\pi f_s n)}{\pi f_s n} \tag{10.36}$$

In developed form, the convolution then expresses as

$$x(n) = \sum_{k=-\infty}^{+\infty} x(kT_s) \frac{\sin(\pi f_s (n - kT_s))}{\pi f_s (n - kT_s)}.$$

This formula shows that **it is possible to perfectly reconstruct a bandlimited signal** from its samples, provided that the sampling rate f_s is more than twice the maximum frequency B of the signal. Half the sampling frequency, $f_s/2$ is called the Nyquist frequency, while the minimum sampling frequency is the Nyquist rate.

The Shannon-Nyquist theorem can then be stated as follows:

Theorem 1. *–Shannon-Nyquist theorem.*
For a real bandlimited signal with maximum frequency B, a correct sampling requires

$$f_s > 2B.$$

It is then possible to perfectly reconstruct the original signal from its samples, through the Shannon-Nyquist interpolation formula

$$x(n) = \sum_{k=-\infty}^{+\infty} x(kT_s) \frac{\sin(\pi f_s (n - kT_s))}{\pi f_s (n - kT_s)}.$$

10.6.2 Case of continuous-time signals.

The same reasonings can be done in the case of continuous-time signals. Sampling a signal $x(t)$ consists in multiplying the initial signal with a (time-continuous) Dirac comb with period $T_s = 1/f_s$. In the frequency domain, this yields the convolution of the initial spectrum with the Fourier transform of the Dirac comb, which is also, as in the discrete case, a Dirac comb. Then one obtains a periodic summation of the original spectrum:

$$X_s(f) = f_s \sum_k X(f - kf_s).$$

Aliasing is avoided if the sampling frequency f_s is such that

$$f_s > 2B.$$

In such case, it is possible to perfectly recover the original signal from its samples, using the reconstruction formula

$$x(t) = \sum_{k=-\infty}^{+\infty} x(kT_s) \frac{\sin(\pi f_s(t - kT_s))}{\pi f_s(t - kT_s)}.$$

10.6.3 Illustrations

Exercise 7. *Here we want to check the Shannon interpolation formula for correctly sampled signals:*

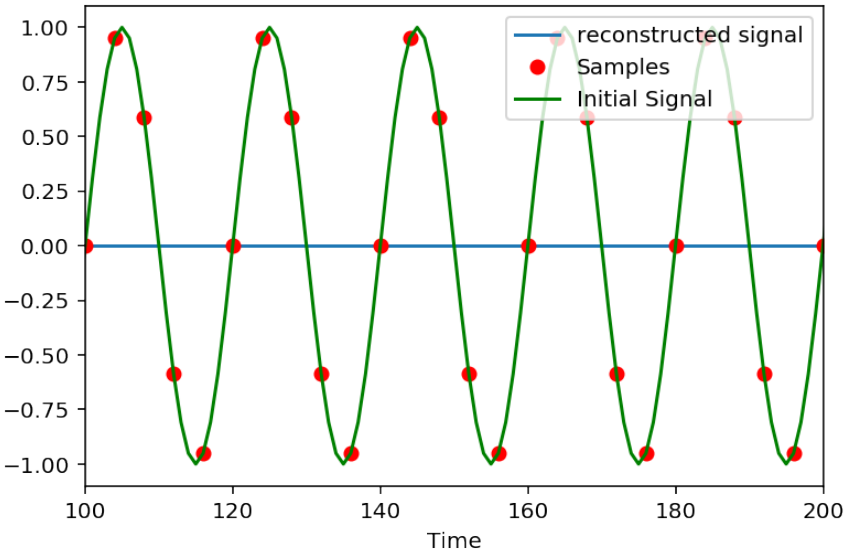
$$x(n) = \sum_{k=-\infty}^{+\infty} x(kT_s) \frac{\sin(\pi f_s(n - kT_s))}{\pi f_s(n - kT_s)}.$$

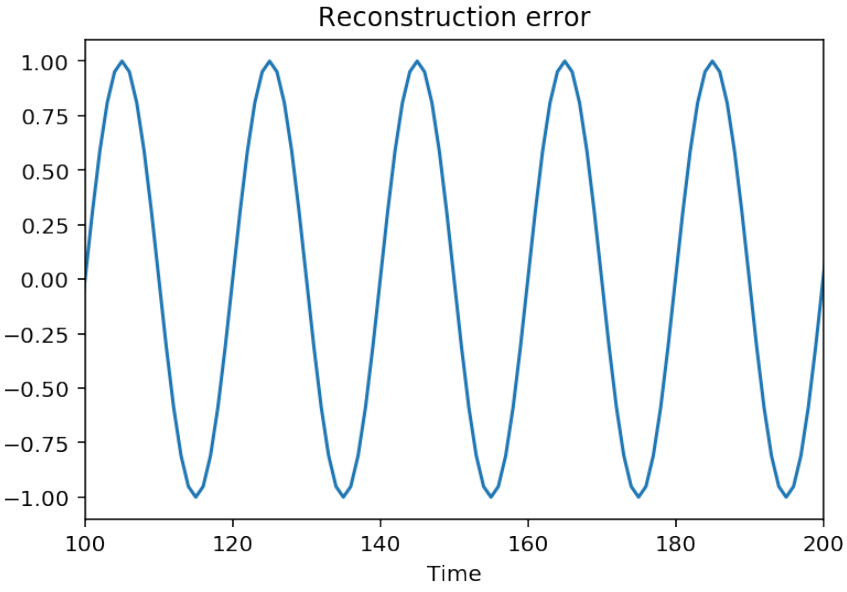
In order to do that, you will first create a sinusoid with frequency f_0 (eg $f_0 = 0.05$). You will sample this sine wave at 4 samples per period ($f_s = 4f_0$). Then, you will implement the interpolation formula and will compare the approximation (finite number of terms in the sum) to the intial signal. The `numpy` *module provides a* `sinc` *function, but you should beware to the fact that the definition used includes the π:* $\mathrm{sinc}(x) = \sin(\pi x)/(\pi x)$

You have to study, complete the following script and implement the interpolation formula.

```
N = 4000
t = np.arange(N)
fo = 0.05  #--> 1/fo=20 samples per periode
x = sin(2 * pi * fo * t)
ts = np.arange(0, N, 4)  # 5 samples per periode
xs = x[::4]  #downsampling, 5 samples per periode
num = np.size(ts)  # number of samples

Ts, Fs = 4, 1 / 4
x_rec = zeros(N)  #reconstructed signal
#
# IMPLEMENT HERE THE RECONSTRUCTION FORMULA x_rec=...
#

#Plotting the rsults
plt.plot(t, x_rec, label="reconstructed signal")
plt.plot(ts, xs, 'ro', label="Samples")
plt.plot(t, x, '-g', label="Initial Signal")
plt.xlabel("Time")
plt.xlim([100, 200])
```

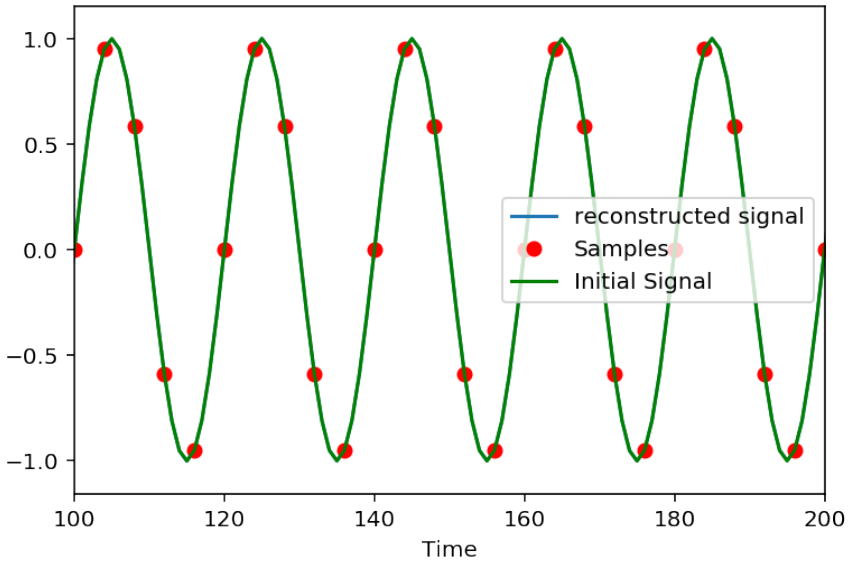
```
plt.legend()

plt.figure()
plt.plot(t, x - x_rec)
plt.title("Reconstruction error")
plt.xlabel("Time")
_ = plt.xlim([100, 200])
```






```
N = 300
t = np.arange(N)
fo = 0.05  #--> 1/fo=20 samples per period
x = sin(2 * pi * fo * t)
```

```
ts = np.arange(0, N, 4)  # 5 samples per period
num = np.size(ts)  # number of samples
xs = x[::4]  #downsampling, 5 samples per period

Ts, Fs = 4, 1 / 4
x_rec = zeros(N)  #reconstructed signal
for k in range(num):
    x_rec = x_rec + xs[k] * np.sinc(Fs*(t - k * Ts))  #! The sinc includes
        the pi

plt.plot(t, x_rec, label="reconstructed signal")
plt.plot(ts, xs, 'ro', label="Samples")
plt.plot(t, x, '-g', label="Initial Signal")
plt.xlabel("Time")
plt.xlim([100, 200])
plt.legend()

plt.figure()
plt.plot(t, x - x_rec)
plt.title("Reconstruction error")
plt.xlabel("Time")
_ = plt.xlim([100, 200])
```



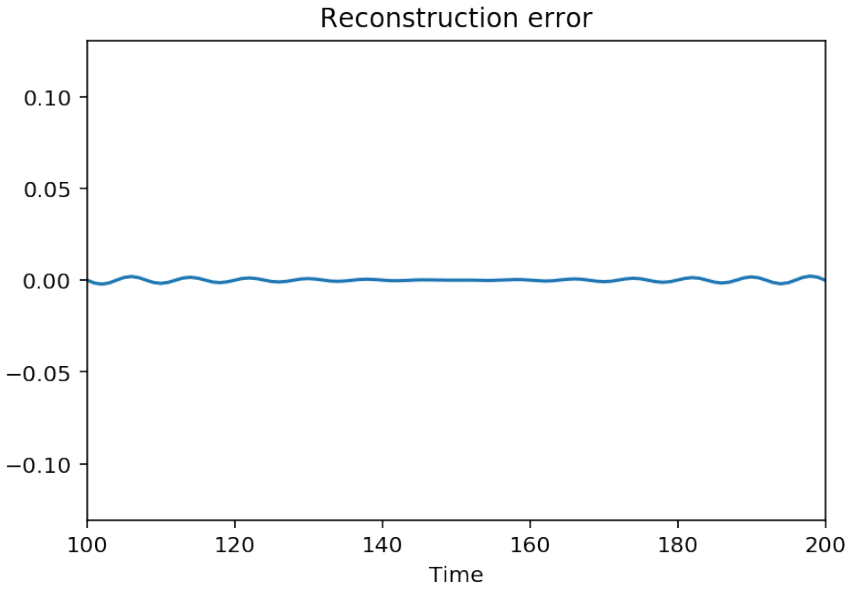
We observe that there still exists a very small error, but an existing one, and if we look carefully at it, we may observe that the error is more important on the edges of the interval.

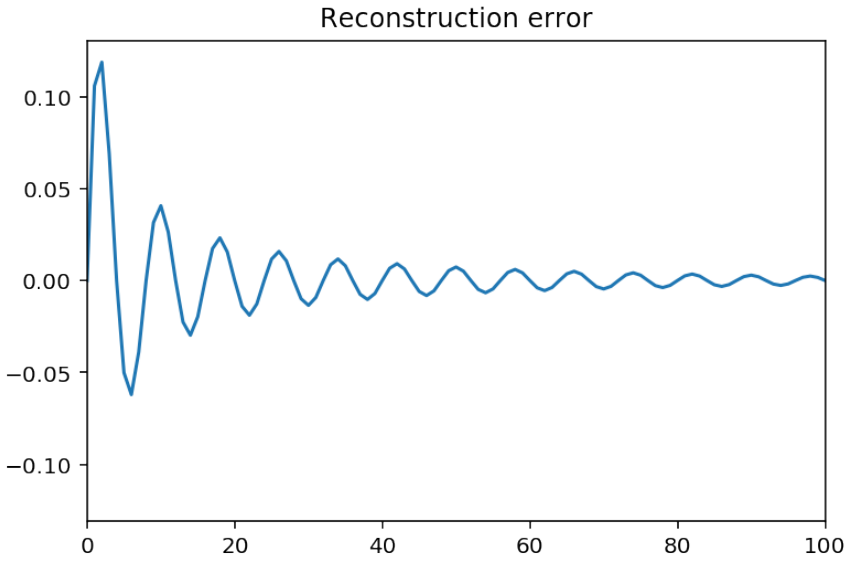
```
plt.figure()
plt.plot(t,x-x_rec)
plt.title("Reconstruction error")
_=plt.xlim([0,100])
```

Actually, there is a duality between the time and frequency domains which implies that

signals with a finite support in one domain have an infinite support in the other.

Consequently, a signal cannot be limited simultaneously in both domains. In the case of our previous sine wave, when we compute the Discrete-time Fourier transform (3.1), we implicitly suppose that the signal


Reconstruction error
0.10
0.05
0.00
−0.05
−0.10
100
120
140
160
180
200
Time


Reconstruction error
0.10
0.05
0.00
−0.05
−0.10
0
20
40
60
80
100

is zero out of the observation interval. Therefore, its Fourier transform has infinite support and time sampling will result in (a small) aliasing in the frequency domain.

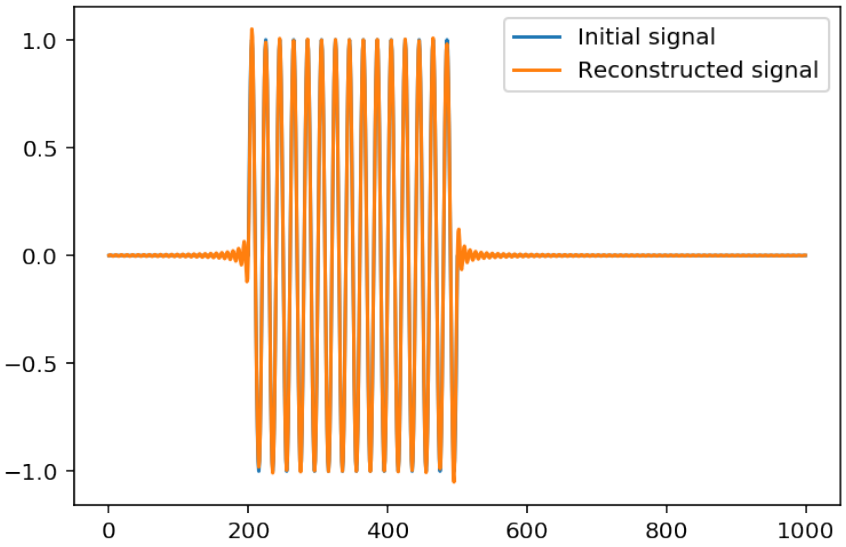
It thus seems that it is not possible to downsample **time-limited** discrete signals without (a perhaps very small) loss. Actually, we will see that this is still possible, using subband coding.

Analysis of the aliasing due to time-limited support.

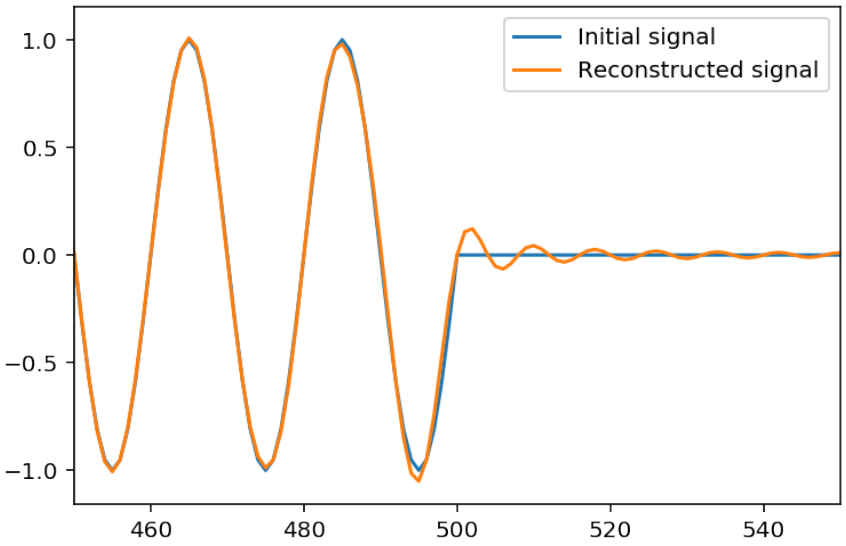
We first zero-pad the initial signal; - this emphasizes that the signal is time-limited - and enables to look at what happens at the edges of the support

```
bigN = 1000
x_extended = np.zeros(bigN)
x_extended[200:200 + N] = x
#
t = np.arange(0, bigN)  #
ts = np.arange(0, bigN, 4)  #
num = np.size(ts)  # number of samples
xs = x_extended[::4]  #downsampling, 5 samples per periode
```

```
# Reconstruction
Ts, Fs = 4, 1 / 4
x_rec = zeros(bigN)  #reconstructed signal
for n in range(num):
    x_rec = x_rec + xs[n] * np.sinc(Fs*(t - n * Ts))  #! The sinc includes
        the pi

# Plotting the results
plt.plot(x_extended, label="Initial signal")
plt.plot(t, x_rec, label="Reconstructed signal")
plt.legend()
plt.figure()
plt.plot(x_extended, label="Initial signal")
plt.plot(t, x_rec, label="Reconstructed signal")
plt.xlim([450, 550])
_ = plt.legend()
```



Analysis in the frequency domain




```
xxs = np.zeros(np.size(x_extended))
xxs[::4] = x_extended[::4]
xf = np.abs(fft(x_extended, 4000))
xxsf = 4 * np.abs(fft(xxs, 4000))
f = np.linspace(0, 1, 4000)
# Plotting
plt.plot(f, xf, label="Initial signal")
plt.ylim([0, 40])
_ = plt.xlim([0, 1 / 2])
#plt.plot(f,xxsf,label="Sampled signal")
# Details
plt.figure()
plt.plot(f, xf, label="Initial signal")
plt.plot(f, xxsf, label="Sampled signal")
plt.legend()
plt.ylim([0, 40])
_ = plt.xlim([0, 1 / 4])
```

We see that - we have infinite support in the frequency domain, the graph of the initial signal shows that it is not band-limited. - This implies **aliasing**: the graph of the Fourier transform of the sampled signal clearly shows that aliasing occurs, which modifies the values below $f_s/2 = 0.125$.

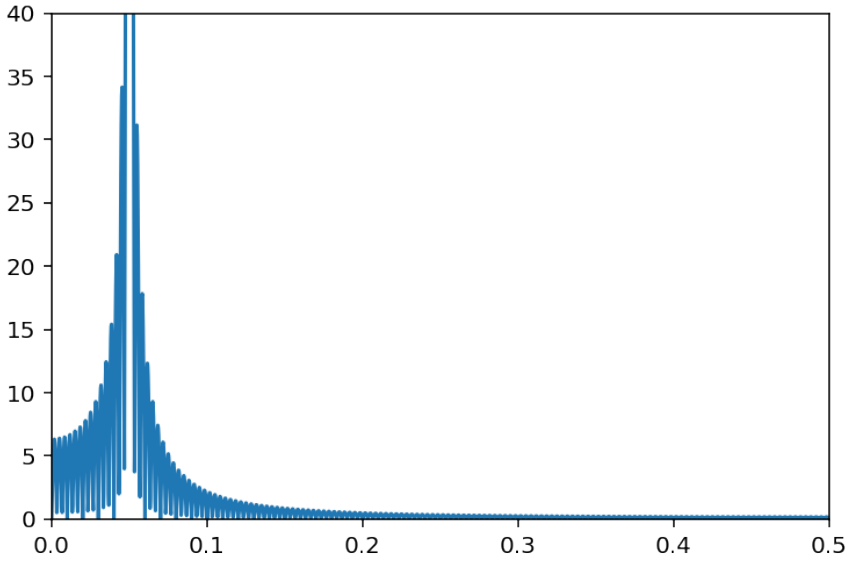
10.6.4 Sampling of band-pass signals

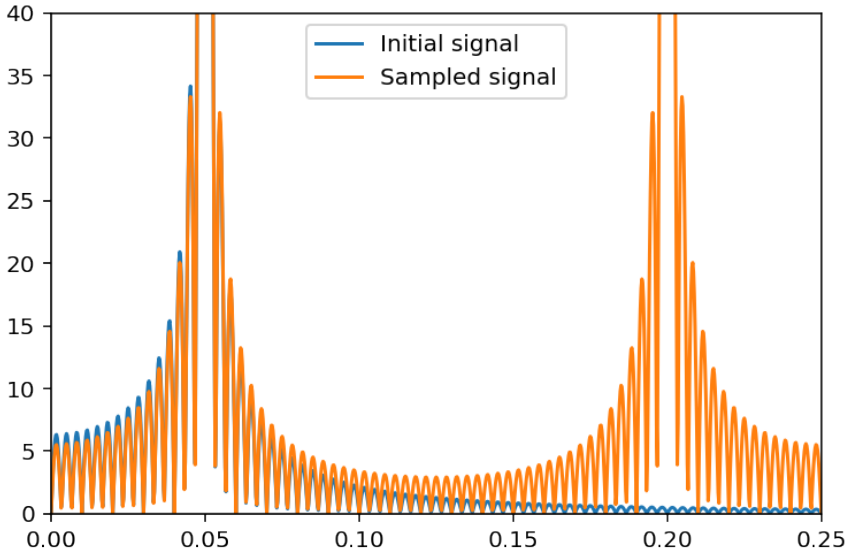
to be continued...

10.7 Lab on basics in image processing

10.7.1 Introduction

The objective of this lab is to show how the notions we discovered in the monodimensional case – that is for signals, can be extended to the two dimensional case. This also enable to have a new look at these notions and perhaps contribute to stenghten their understanding.


40
35
30
25
20
15
10
5
0
0.0
0.1
0.2
0.3
0.4
0.5


Initial signal
Sampled signal
40
35
30
25
20
15
10
5
0
0.00
0.05
0.10
0.15
0.20
0.25

In particular, we will look at the problems of representation and filtering, both in the direct (spatial) domain and in the transformed (spatial frequencies) domain. Next we will look at the problems of sampling and filtering.

Within Python, the modules `scipy.signal` and `scipy.ndimage` will be useful.

L'objectif de ce laboratoire est de montrer comment les notions que nous avons découvertes dans le cas monodimensionnel - c'est-à-dire pour les signaux, peuvent être étendues au cas bidimensionnel. Cela permet également d'avoir un nouveau regard sur ces notions et peut-être contribuer à renforcer leur compréhension.

En particulier, nous examinerons les problèmes de représentation et de filtrage, à la fois dans le domaine direct (spatial) et dans le domaine transformé (fréquences spatiales). Ensuite, nous examinerons les problèmes d'échantillonnage et de filtrage.

Dans Python, les modules `scipy.signal` et`scipy.ndimage` seront utiles.

In order to facilitate your learning and work, your servant has prepared a bunch of useful functions, namely:

```
rect2 -- returns a two dimensional centered rectangle
bandpass2d -- returns a 2D-bandpass filter (in the frequency domain)
showfft2 -- display of the 2D-Fourier transform, correcly centered and normalized
mesh -- display a ``3D'' representaion of an objet (à la Matlab (tm))
```

To read an image file, you will use the function `imread`.

To display an image in gray levels, you may use

```
imshow(S,cmap='gray',origin='upper')
```

You may either display your graphics inline (it is the default) or using external windows; for that call

11 Digital filters

11.0.1 Introduction

To be continued

11.0.2 The z-transform

To be continued

11.1 Pole-zero locations and transfer functions behavior

```
from zerospolesdisplay import ZerosPolesDisplay
```

Let $H(z)$ be a rational fraction

$$H(z) = \frac{N(z)}{D(z)},$$

where both $N(z)$ and $D(z)$ are polynomials in z, with $z \in \mathbb{C}$. The roots of both polynomials are very important in the behavior of $H(z)$.

Definition 3. *The roots of the numerator $N(z)$ are called the **zeros** of the transfer function. The roots of the denominator $N(z)$ are called the **poles** of the transfer function.*

Note that poles can occur at $z = \infty$. Recall that for $z = \exp(j2\pi f)$, the Z-transform reduces to the Fourier transform:

$$H(z = e^{j2\pi f}) = H(f).$$

Example 2. *Examples of rational fractions*

- *The rational fraction*

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

has a pole for $z = a$ and a zero for $z = 0$.

- *The rational fraction*

$$H(z) = \frac{1 - cz^{-1}}{(1 - az^{-1})(1 - bz^{-1})}$$

has two poles for $z = a$ and $z = b$ and two zeros, for $z = 0$ and $z = c$.

- *The rational fraction*

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

has $N-1$ zeros of the form $z = \exp(j2\pi k/N)$, for $k = 1..N$. Actually there are N roots for the numerator and one root for the denominator, but the common root $z = 1$ cancels.

Exercise 8. *Give the difference equations corresponding to the previous examples, and compute the inverse Z-transforms (impulse responses). In particular, show that for the last transfer function, the impulse response is* $\mathrm{rect}_N(n)$.

Property 2. *For any polynomial with real coefficients, the roots are either reals or appear by complex conjugate pairs.*

Proof. Let

$$P(z) = \sum_{k=0}^{L-1} p_k z^k.$$

If \$z_0=ρ&{jθ}\$*is a root of the polynom, then*

$$P(z_0) = \sum_{k=0}^{L-1} p_k \rho^k e^{-jk\theta},$$

and

$$P(z_0^*) = \sum_{k=0}^{L-1} p_k \rho^k e^{jk\theta}.$$

Putting $e^{-j(L-1)\theta}$ in factor, we get

$$P(z_0^*) = e^{-j(L-1)\theta} \sum_{k=0}^{L-1} p_k \rho^k e^{jk\theta} = e^{-j(L-1)\theta} P(z_0) = 0.$$

□

This shows that if the coefficients of the transfer function are real, then the zeros and poles are either real or appear in complex conjugate pairs. This is usually the case, since these coefficients are the coefficients of the underlying difference equation. For real filters, the difference equation has obviously real coefficients.

For real filters, the zeros and poles are either real or appear in complex conjugate pairs.

11.1.1 Analysis of no-pole transfer functions

Suppose that a transfer function $H(z) = N(z)$ has two conjugated zeros z_0 and z_0^* of the form $\rho e^{\pm j\theta}$. That is, $N(z)$ has the form

$$N(z) = (z - z_0)(z - z_0^*) = (z - \rho e^{j\theta})(z - \rho e^{-j\theta}) \tag{11.1}$$

$$= z^2 - 2\rho\cos(\theta)z + \rho^2 \tag{11.2}$$

For a single zero, we see that the transfer function, with $z = e^{j2\pi f}$, is minimum when $|z - z_0|$ is minimum. Since $|z - z_0|$ can be interpreted as the distance between points z and z_0 in the complex plane, this happens when z and z_0 have the same phase (frequency). When z_0 has a modulus one, that is is situated on the unit circle, then $z - z_0$ will be null for this frequency and the function transfer will present a null.

```
%matplotlib inline
poles=np.array([0])
zeros=np.array([0.85*np.exp(1j*2*pi*0.2)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a single zero", label="fig:singlezero")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

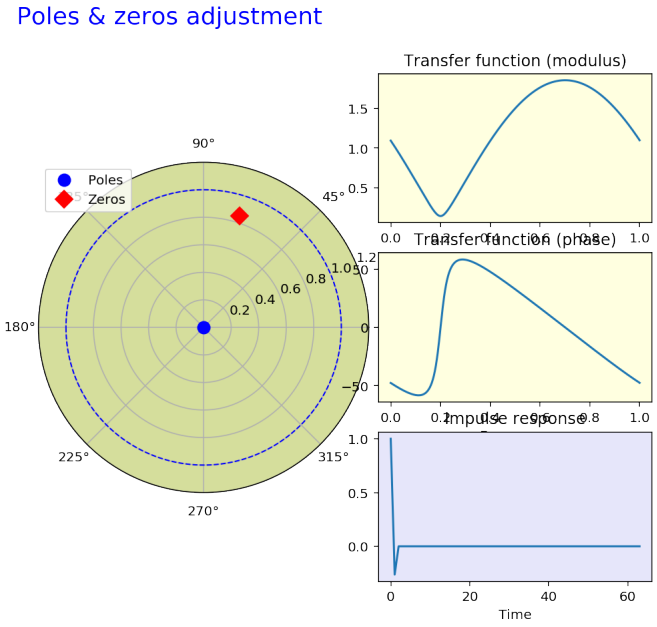


Figure 11.1: Poles-Zeros representation, Transfer function and Impulse response for a single zero

```
poles=np.array([0])
zeros=np.array([0.95*np.exp(1j*2*pi*0.4)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a single zero", label="fig:singlezero_2")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

With two or more zeros, the same kind of observations holds. However,because of the interactions berween the zeros, the minimum no more strictly occur for the frequencies of the zeros but for some close frequencies.

This is illustrated now in the case of two complex-conjugated zeros (which corresponds to a transfer function with real coefficients).

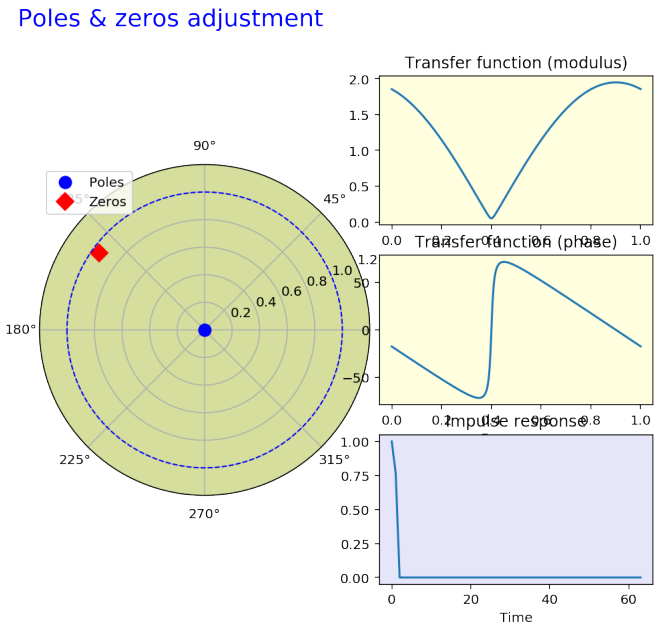


Figure 11.2: Poles-Zeros representation, Transfer function and Impulse response for a single zero

```
poles=np.array([0])
zeros=np.array([0.95*np.exp(1j*2*pi*0.2), 0.95*np.exp(-1j*2*pi*0.2)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a double zero", label="fig:doublezero")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

```
poles=np.array([0])
zeros=np.array([0.95*np.exp(1j*2*pi*0.2), 0.95*np.exp(-1j*2*pi*0.2), 0.97*np
    .exp(1j*2*pi*0.3), 0.97*np.exp(-1j*2*pi*0.3)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a 4 zeros", label="fig:doublezero2")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

11.1.2 Analysis of all-poles transfer functions

For an all-pole transfer function,

$$H(z) = \frac{1}{D(z)}$$

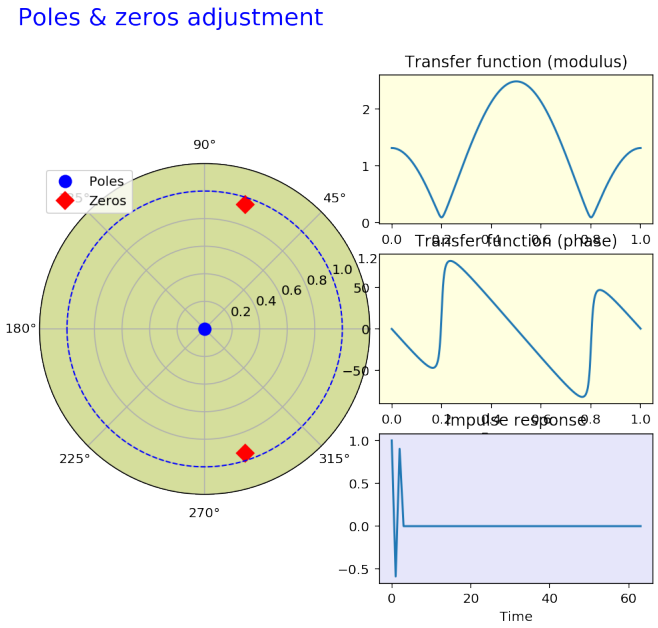


Figure 11.3: Poles-Zeros representation, Transfer function and Impulse response for a double zero

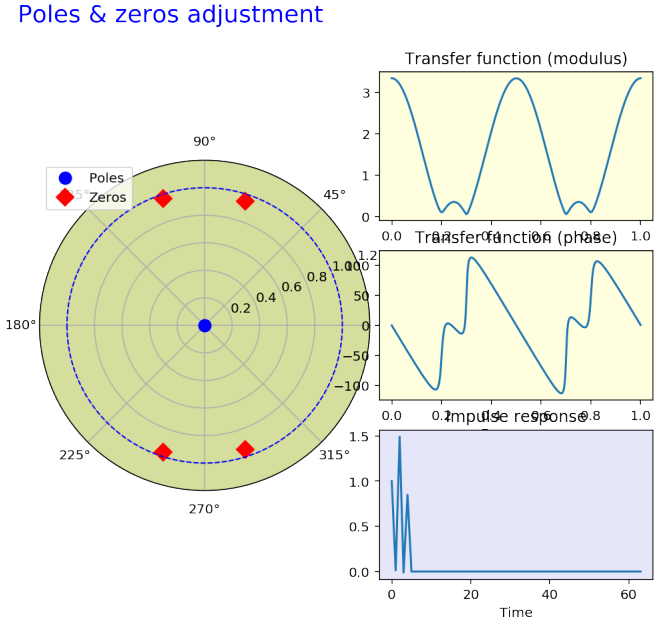


Figure 11.4: Poles-Zeros representation, Transfer function and Impulse response for a 4 zeros

we will obviously have the inverse behavior. Instead of an attenuation at a frequency close to the value given by the angle of the root, we will obtain a surtension. This is illustrated below, in the case of a single, the multiple poles.

```
zeros=np.array([0])
poles=np.array([0.85*np.exp(1j*2*pi*0.2)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a single pole", label="fig:singlepole")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

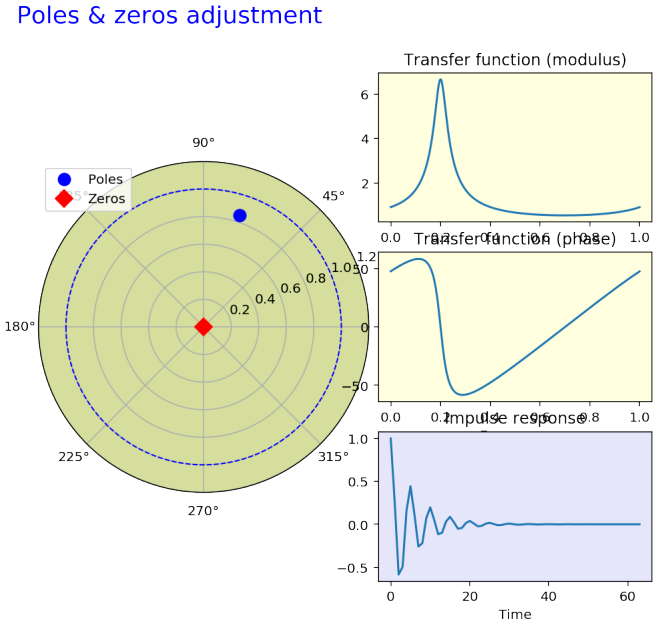


Figure 11.5: Poles-Zeros representation, Transfer function and Impulse response for a single pole

Furthermore, we see that

the closer the pole to the unit circle, the more important the surtension

```
zeros=np.array([0])
poles=np.array([0.97*np.exp(1j*2*pi*0.2)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a single pole", label="fig:singlepole_2")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

We can also remark that if the modulus of the pole becomes higher than one, then the impulse response diverges. The system is no more stable.

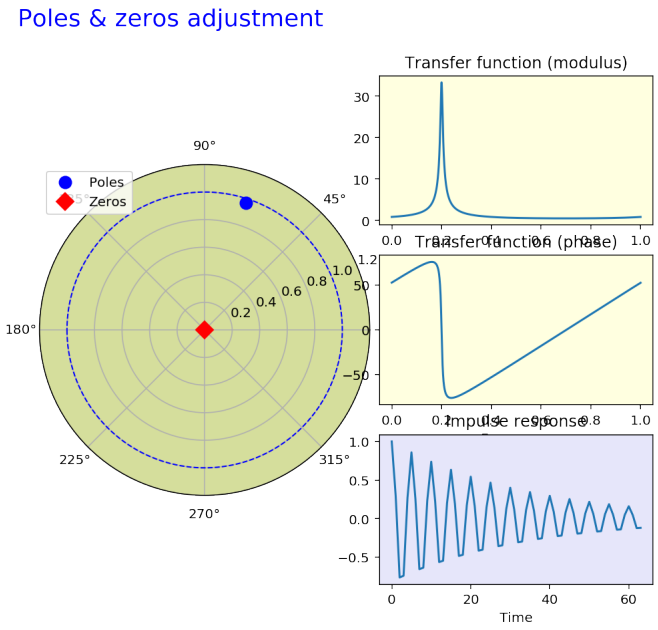


Figure 11.6: Poles-Zeros representation, Transfer function and Impulse response for a single pole

Poles with a modulus higher than one yields an instable system.

```
zeros=np.array([0])
poles=np.array([1.1*np.exp(1j*2*pi*0.2)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a single pole", label="fig:singlepole_3")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

For 4 poles, we get the following: two pairs of surtensions, for frequencies essentialy given by the arguments of the poles.

```
zeros=np.array([0])
poles=np.array([0.95*np.exp(1j*2*pi*0.2), 0.95*np.exp(-1j*2*pi*0.2), 0.97*np
    .exp(1j*2*pi*0.3), 0.97*np.exp(-1j*2*pi*0.3)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a 4 poles", label="fig:doublepoles2")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

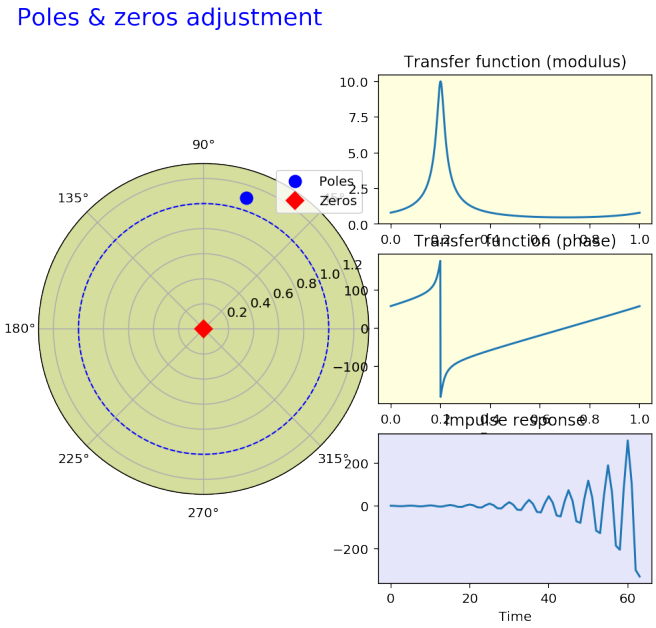


Figure 11.7: Poles-Zeros representation, Transfer function and Impulse response for a single pole

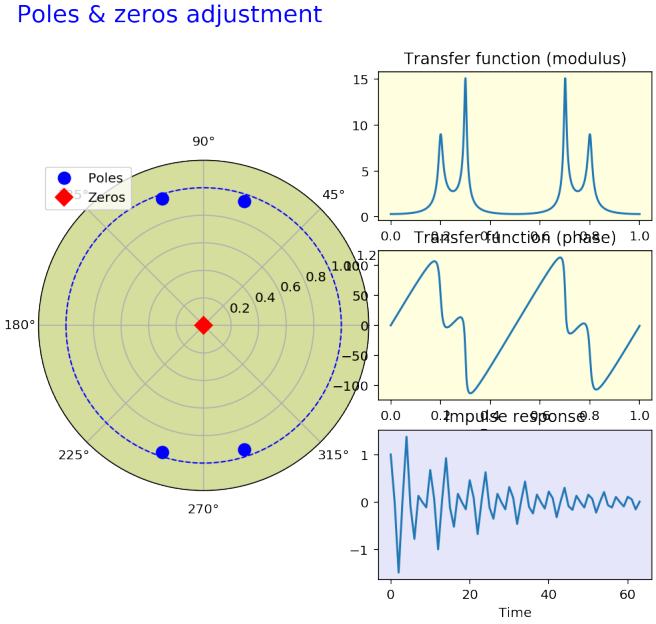


Figure 11.8: Poles-Zeros representation, Transfer function and Impulse response for a 4 poles

11.1.3 General transfer functions

For a general transfer function, we will get the combination of the two effects: attenuation or even nulls that are given by the zeros, and sutensions, maxima that are yied by the poles. Here is a simple example.

```
poles=np.array([0.85*np.exp(1j*2*pi*0.2), 0.85*np.exp(-1j*2*pi*0.2)])
zeros=np.array([1.1*np.exp(1j*2*pi*0.3), 1.1*np.exp(-1j*2*pi*0.3)])
A=ZerosPolesDisplay(poles,zeros)
figcaption("Poles-Zeros representation, Transfer function and Impulse
    response for a 2 poles and 2 zeros", label="fig:poleszero")
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```

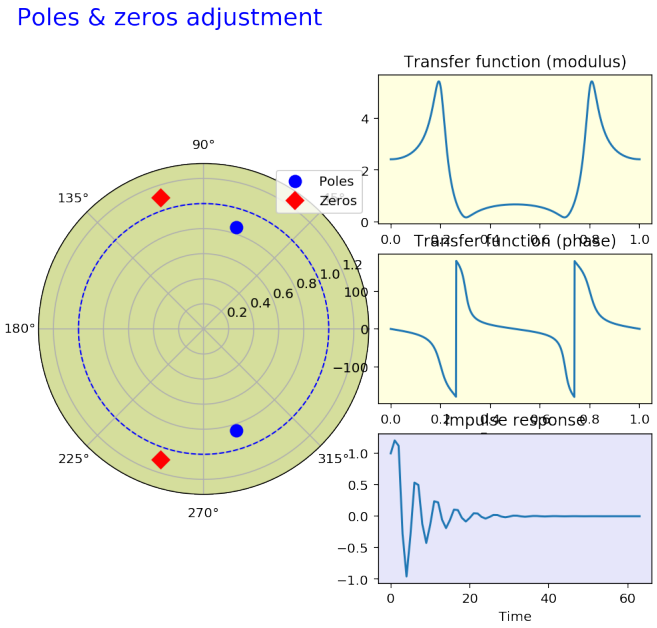


Figure 11.9: Poles-Zeros representation, Transfer function and Impulse response for a 2 poles and 2 zeros

Hence we see that it is possible to understand the behavior of transfer function by studying the location of their poles and zeros. It is even possible to design transfer function by optimizing the placement of their poles and zeros.

For instance, given the poles and zeros in the previous example, we immediately find the coefficients of the filter by computing the corresponding polynomials:

```
print("poles",A.poles)
print("zeros",A.zeros)

print("coeffs a:",np.poly(A.poles))
print("coeffs b:",np.poly(A.zeros))
```

```
poles [0.26266445+0.80839804j 0.26266445-0.80839804j]
zeros [-0.33991869+1.04616217j -0.33991869-1.04616217j]
```

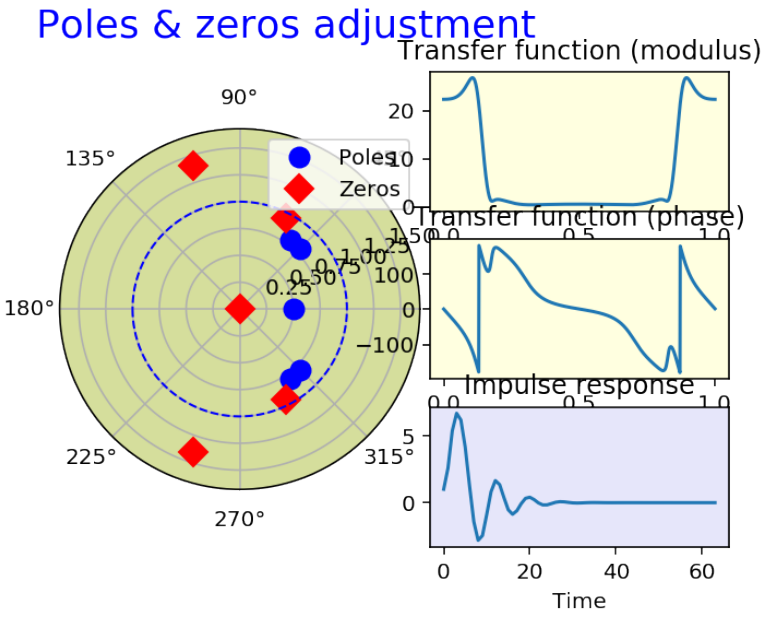
```
coeffs a: [ 1.          -0.52532889  0.7225     ]
coeffs b: [1.          0.67983739 1.21       ]
```

In order to further investigate these properties and experiment with the pole and zeros placement, your servant has prepared a ZerosPolesPlay class. **Enjoy!**

```
%matplotlib
%run zerospolesplay.py
```

```
Using matplotlib backend: TkAgg
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```



11.1.4 Appendix – listing of the class ZerosPolesPlay

```
# %load zerospolesplay.py
```

```
"""
Transfer function adjustment using zeros and poles drag and drop!
jfb 2015 – last update november 2018
"""
import numpy as np
import matplotlib.pyplot as plt
from numpy import pi
```

```
#line, = ax.plot(xs, ys, 'o', picker=5)  # 5 points tolerance


class ZerosPolesPlay():
    def __init__(self,
                 poles=np.array([0.7 * np.exp(1j * 2 * np.pi * 0.1)]),
                 zeros=np.array([1.27 * np.exp(1j * 2 * np.pi * 0.3)]),
                 N=1000,
                 response_real=True,
                 ymax=1.2,
                 Nir=64):

        if response_real:
            self.poles, self.poles_isreal = self.sym_comp(poles)
            self.zeros, self.zeros_isreal = self.sym_comp(zeros)
        else:
            self.poles = poles
            self.poles_isreal = (np.abs(np.imag(poles)) < 1e-12)
            self.zeros = zeros
            self.zeros_isreal = (np.abs(np.imag(zeros)) < 1e-12)

        self.ymax = np.max([
            ymax, 1.2 * np.max(np.concatenate((np.abs(poles), np.abs(zeros))
                ))
        ])
        self.poles_th = np.angle(self.poles)
        self.poles_r = np.abs(self.poles)
        self.zeros_th = np.angle(self.zeros)
        self.zeros_r = np.abs(self.zeros)
        self.N = N
        self.Nir = Nir
        self.response_real = response_real

        self.being_dragged = None
        self.nature_dragged = None
        self.poles_line = None
        self.zeros_line = None
        self.setup_main_screen()
        self.connect()
        self.update()

    def setup_main_screen(self):

        import matplotlib.gridspec as gridspec

        #Poles & zeros
        self.fig = plt.figure()
        gs = gridspec.GridSpec(3, 12)
        #self.ax = self.fig.add_axes([0.1, 0.1, 0.77, 0.77], polar=True,
            facecolor='#d5de9c')
        #self.ax=self.fig.add_subplot(221,polar=True, facecolor='#d5de9c')
        self.ax = plt.subplot(gs[0:, 0:6], polar=True, facecolor='#d5de9c')
        #self.ax = self.fig.add_subplot(111, polar=True)
        self.fig.suptitle(
            'Poles & zeros adjustment',
            fontsize=18,
            color='blue',
            x=0.1,
```

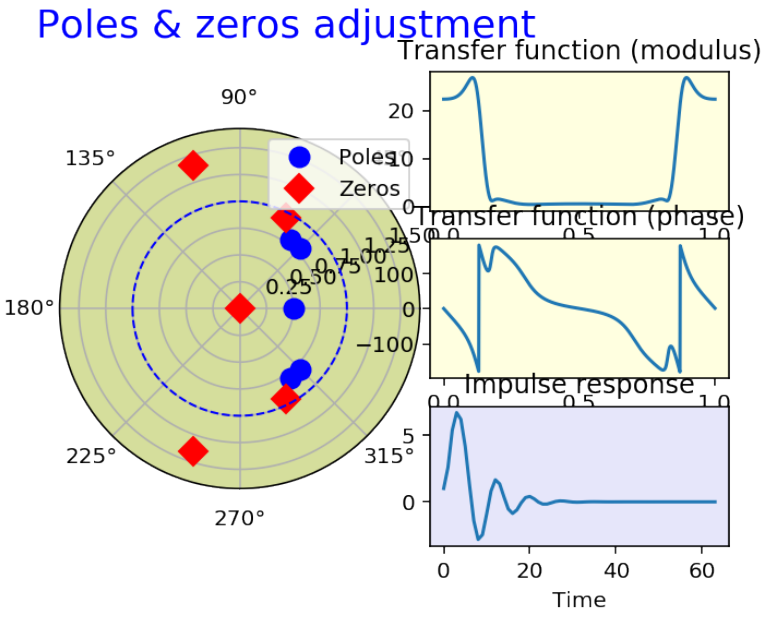
```
            y=0.98,
            horizontalalignment='left')
        #self.ax.set_title('Poles & zeros adjustment',fontsize=16, color='
            blue')
        self.ax.set_ylim([0, self.ymax])
        self.poles_line, = self.ax.plot(
            self.poles_th, self.poles_r, 'ob', ms=9, picker=5, label="Poles"
                )
        self.zeros_line, = self.ax.plot(
            self.zeros_th, self.zeros_r, 'Dr', ms=9, picker=5, label="Zeros"
                )
        self.ax.plot(
            np.linspace(-np.pi, np.pi, 500), np.ones(500), '--b', lw=1)
        self.ax.legend(loc=1)

        #Transfer function
        #self.figTF, self.axTF = plt.subplots(2, sharex=True)
        #self.axTF0=self.fig.add_subplot(222,facecolor='LightYellow')
        self.axTF0 = plt.subplot(gs[0, 6:11], facecolor='LightYellow')
        #self.axTF[0].set_axis_bgcolor('LightYellow')
        self.axTF0.set_title('Transfer function (modulus)')
        #self.axTF1=self.fig.add_subplot(224,facecolor='LightYellow')
        self.axTF1 = plt.subplot(gs[1, 6:11], facecolor='LightYellow')
        self.axTF1.set_title('Transfer function (phase)')
        self.axTF1.set_xlabel('Frequency')
        f = np.linspace(0, 1, self.N)
        self.TF = np.fft.fft(np.poly(self.zeros), self.N) / np.fft.fft(
            np.poly(self.poles), self.N)
        self.TF_m_line, = self.axTF0.plot(f, np.abs(self.TF))
        self.TF_p_line, = self.axTF1.plot(f, 180 / np.pi * np.angle(self.TF)
            )
        #self.figTF.canvas.draw()

        #Impulse response
        #self.figIR = plt.figure()
        #self.axIR = self.fig.add_subplot(223,facecolor='Lavender')
        self.axIR = plt.subplot(gs[2, 6:11], facecolor='Lavender')
        self.IR = self.impz(self.zeros, self.poles,
                            self.Nir)  #np.real(np.fft.ifft(self.TF))
        self.axIR.set_title('Impulse response')
        self.axIR.set_xlabel('Time')
        self.IR_m_line, = self.axIR.plot(self.IR)
        #self.figIR.canvas.draw()
        self.fig.canvas.draw()
        self.fig.tight_layout()

    def impz(self, zeros, poles, L):
        from scipy.signal import lfilter
        a = np.poly(poles)
        b = np.poly(zeros)
        d = np.zeros(L)
        d[0] = 1
        h = lfilter(b, a, d)
        return h

    def sym_comp(self, p):
        L = np.size(p)
        r = list()
        c = list()
```

```
        for z in p:
            if np.abs(np.imag(z)) < 1e-12:
                r.append(z)
            else:
                c.append(z)
        out = np.concatenate((c, r, np.conjugate(c[::-1])))
        isreal = (np.abs(np.imag(out)) < 1e-12)
        return out, isreal
#sym_comp([1+1j, 2, 3-2j])

    def connect(self):
        self.cidpick = self.fig.canvas.mpl_connect('pick_event', self.
            on_pick)
        self.cidrelease = self.fig.canvas.mpl_connect('button_release_event'
            ,
                                                      self.on_release)
        self.cidmotion = self.fig.canvas.mpl_connect('motion_notify_event',
                                                     self.on_motion)

    def update(self):

        #poles and zeros
        #self.fig.canvas.draw()

        #Transfer function & Impulse response
        if not (self.being_dragged is None):
            #print("Was released")

            f = np.linspace(0, 1, self.N)
            self.TF = np.fft.fft(np.poly(self.zeros), self.N) / np.fft.fft(
                np.poly(self.poles), self.N)
            self.TF_m_line.set_ydata(np.abs(self.TF))
            M = np.max(np.abs(self.TF))
            #update the yscale
            current_ylim = self.axTF0.get_ylim()[1]
            if M > current_ylim or M < 0.5 * current_ylim:
                self.axTF0.set_ylim([0, 1.2 * M])

            #phase
            self.TF_p_line.set_ydata(180 / np.pi * np.angle(self.TF))
            #self.figTF.canvas.draw()

            # Impulse response
            self.IR = self.impz(self.zeros, self.poles,
                                self.Nir)  #np.fft.ifft(self.TF)
            #print(self.IR)
            self.IR_m_line.set_ydata(self.IR)
            M = np.max(self.IR)
            Mm = np.min(self.IR)
            #update the yscale
            current_ylim = self.axIR.get_ylim()
            update_ylim = False
            if M > current_ylim[1] or M < 0.5 * current_ylim[1]:
                update_ylim = True
            if Mm < current_ylim[0] or np.abs(Mm) > 0.5 * np.abs(
                current_ylim[
                    0]):
                update_ylim = True
            if update_ylim: self.axIR.set_ylim([Mm, 1.2 * M])
```

```
            #self.figIR.canvas.draw()
        self.fig.canvas.draw()

    def on_pick(self, event):
        """When we click on the figure and hit either the line or the menu
            items this gets called."""
        if event.artist != self.poles_line and event.artist != self.
            zeros_line:
            return
        self.being_dragged = event.ind[0]
        self.nature_dragged = event.artist

    def on_motion(self, event):
        """Move the selected points and update the graphs."""
        if event.inaxes != self.ax: return
        if self.being_dragged is None: return
        p = self.being_dragged  #index of points on the line being dragged
        xd = event.xdata
        yd = event.ydata
        #print(yd)
        if self.nature_dragged == self.poles_line:
            x, y = self.poles_line.get_data()
            if not (self.poles_isreal[p]):
                x[p], y[p] = xd, yd
            else:
                if np.pi / 2 < xd < 3 * np.pi / 2:
                    x[p], y[p] = np.pi, yd
                else:
                    x[p], y[p] = 0, yd
            x[-p - 1], y[-p - 1] = -x[p], y[p]
            self.poles_line.set_data(x, y)  # then update the line
            #print(self.poles)
            self.poles[p] = y[p] * np.exp(1j * x[p])
            self.poles[-p - 1] = y[p] * np.exp(-1j * x[p])

        else:
            x, y = self.zeros_line.get_data()
            if not (self.zeros_isreal[p]):
                x[p], y[p] = xd, yd
            else:
                if np.pi / 2 < xd < 3 * np.pi / 2:
                    x[p], y[p] = np.pi, yd
                else:
                    x[p], y[p] = 0, yd
            x[-p - 1], y[-p - 1] = -x[p], y[p]
            self.zeros_line.set_data(x, y)  # then update the line
            self.zeros[p] = y[p] * np.exp(1j * x[p])  # then update the line
            self.zeros[-p - 1] = y[p] * np.exp(-1j * x[p])

        self.update()  #and the plot

    def on_release(self, event):
        """When we release the mouse, if we were dragging a point, recompute
             everything."""
        if self.being_dragged is None: return

        self.being_dragged = None
        self.nature_dragged = None
```

```
        self.update()


#case of complex poles and zeros
poles = np.array(
    [0.8 * np.exp(1j * 2 * pi * 0.125), 0.8 * np.exp(1j * 2 * pi * 0.15),
        0.5])
zeros = np.array(
    [0.95 * np.exp(1j * 2 * pi * 0.175), 1.4 * np.exp(1j * 2 * pi * 0.3),
        0])
A = ZerosPolesPlay(poles, zeros)
"""
#case of a single real pole
poles=np.array([0.5])
zeros=np.array([0])
A=ZerosPolesPlay(poles,zeros,response_real=False)
"""

plt.show()

#At the end, poles and zeros available as A.poles and A.zeros
```

```
/usr/local/lib/python3.5/site-packages/matplotlib/tight_layout.py:199: UserWarning:
  warnings.warn('Tight layout not applied. '
```



11.2 Synthesis of FIR filters

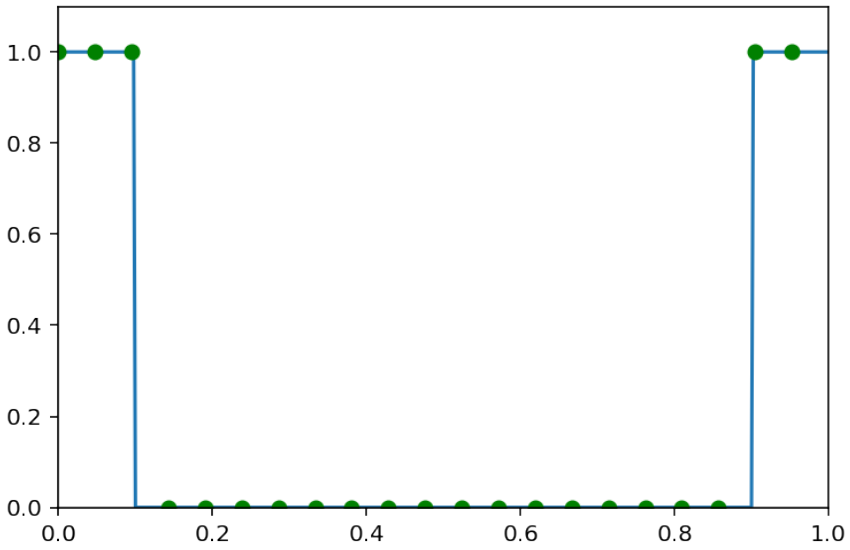
11.2.1 Synthesis by sampling in the frequency domain

The idea is compute the impulse response as the inverse Fourier transform of the transfer function. Since we look for an impulse response of finite length, the idea is to use the inverse Discrete Fourier Transform (DFT),

which links L samples in the frequency domain to L samples in the time domain. Hence, what we need to do is simply to sample the frequency response on the required number of samples, and then to compute the associated impulse response by inverse DFT. This is really simple.

```
%matplotlib inline
```

```
L = 21
#ideal filter
fc = 0.1
N = 20 * L
M = int(np.round(N * fc))
r = np.zeros(N)
r[0:M] = 1
r[-1:-M:-1] = 1
plt.plot(np.arange(0, N) / N, (r))
#sampling the ideal filter
# we want a total of L samples; then step=N//L (integer division)
step = N // L
rs = r[::step]
plt.plot(np.arange(0, N, step) / N, (rs), 'og')
_ = plt.ylim([0, 1.1])
_ = plt.xlim([0, 1])
plt.show()
```



The associated impulse response is given by the inverse DFT. It is represented on figure 11.10.

```
%precision 3
# The impulse response:
h = real(ifft(rs))
print("Impulse response h:", h)
plt.stem(h)
plt.title("Impulse response")
figcaption(
    "Impulse response obtained by frequency sampling", label="fig:h_sampfreq
        ")
```

```
Impulse response h: [ 0.238  0.217  0.161  0.086  0.013 -0.039 -0.059 -0.048 -0.015
  0.044  0.044  0.021 -0.015 -0.048 -0.059 -0.039  0.013  0.086  0.161
  0.217]
```

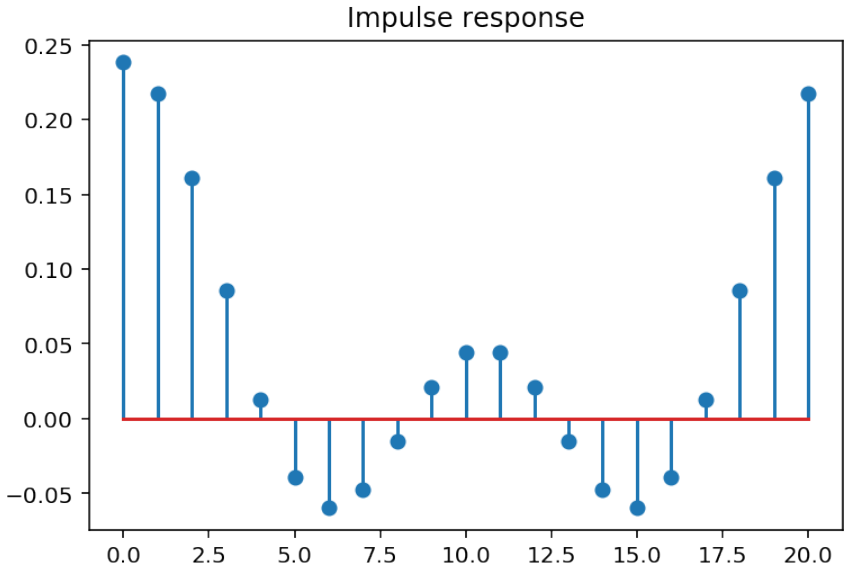


Figure 11.10: Impulse response obtained by frequency sampling

This impulse response is periodic, because of the implicit periodicity of sequences after use of a DFT operation. The "true" response is symmetric around $n = 0$. We can display it using a `fftshift`.

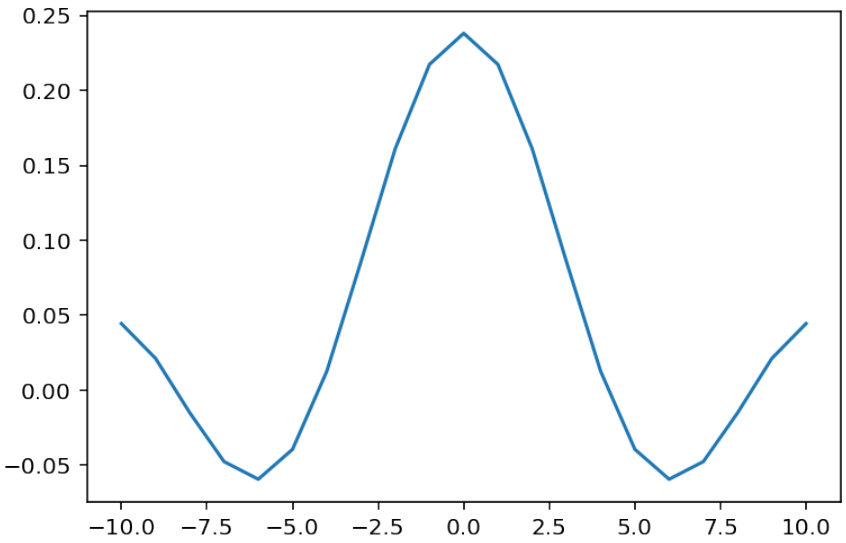
```
delay = (
    L - 1
) / 2 if L % 2 else L / 2  # delay of L/2 is L is even, (L-1)/2 otherwise
_ = plt.plot(np.arange(0, L) - delay, fftshift(h))
```

It is very instructive to look at the frequency response which is effectively realized. In other words we must look at what happens between the points. For that, we approximate the discrete time Fourier transform by zero-padding. At this point, it is really important to shift the impulse response because the zero-padding corresponds to an implicit truncation on L points of the periodic sequence, and we want to keep the true impulse response. This operation introduces a delay of $L/2$ is L is even and $(L-1)/2$ otherwise.

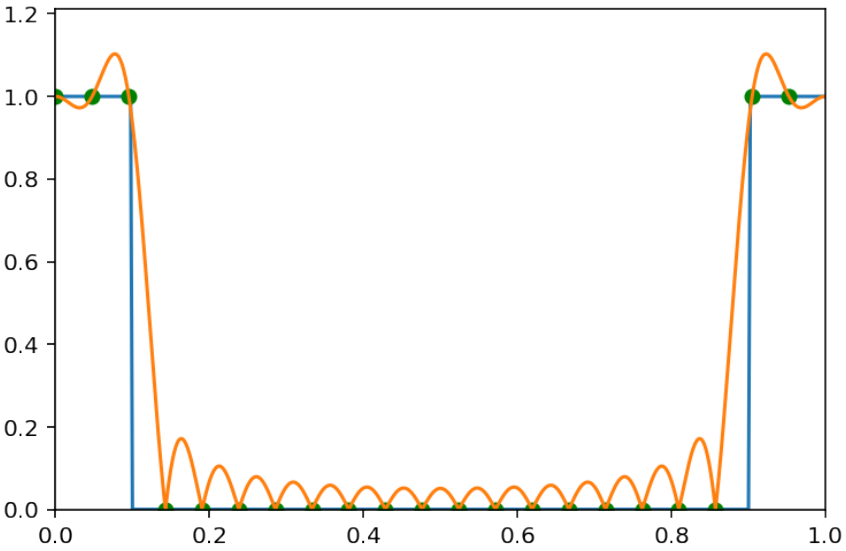
```
NN = 1000
H = fft(fftshift(h),
        NN)  ### <-- Here it is really important to introduce a fftshift
### otherwise, the sequence has large transitions
### on the boundaries
```

Then we display this frequency response and compare it to the ideal filter and to the frequency samples.

```
#ideal filter
plt.plot(np.arange(0, N) / N, (r))
#sampling the ideal filter
plt.plot(np.arange(0, N, step) / N, (rs), 'og')
_ = plt.ylim([0, 1.1])
_ = plt.xlim([0, 1])
```



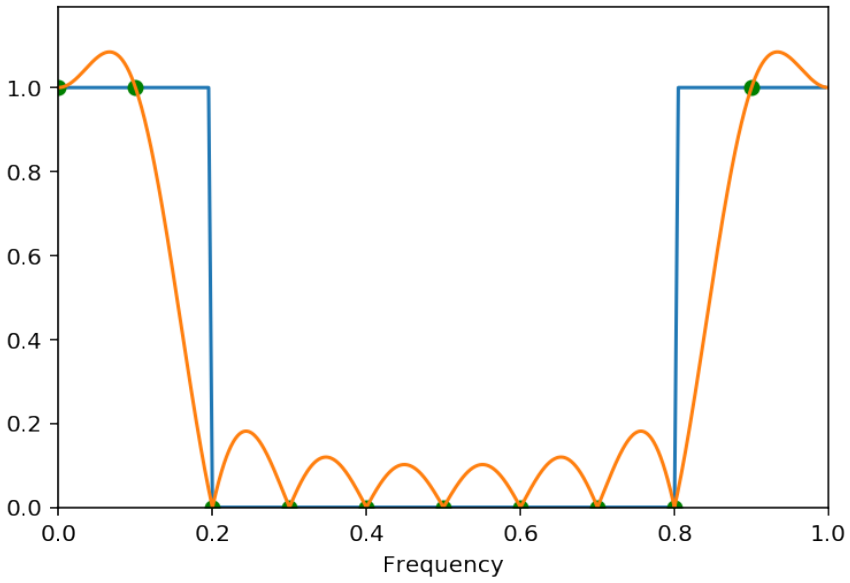

```
#realized filter
_ = plt.plot(np.arange(0, NN) / NN, np.abs(H))
_ = plt.ylim([0, 1.1 * np.max(np.abs(H))])
```



Once we have done all this, we can group all the code into a function and experiment with the parameters, using the interactive facilities of IPython notebook widgets.

```
#mpld3.disable_notebook()
def LP_synth_fsampling(fc=0.2, L=20,plot_impresp=False):

    #ideal filter
    N=20*L; M=int(np.round(N*fc))
    r=np.zeros(N); r[0:M]=1; r[-1:-M:-1]=1
    #sampling the ideal filter
```

```
    # we want a total of L samples; then step=N//L (integer division)
    step=N//L
    rs=r[::step]
    #clear_output(wait=True)

    # The impulse response:
    h=real(ifft(rs))
    if plot_impresp:
        plt.figure()
        %precision 3
        plt.plot(h)
        plt.title("Impulse response")

    plt.figure()
    NN=1000
    H=fft(fftshift(h),NN)

    #ideal filter
    plt.plot(np.arange(0,N)/N, (r))
    #sampling the ideal filter
    plt.plot(np.arange(0,N,step)/N, (rs), 'og')
    plt.xlabel("Frequency")
    _=plt.xlim([0, 1])
    #realized filter
    _=plt.plot(np.arange(0,NN)/NN, np.abs(H))
    _=plt.ylim([0, 1.1*np.max(np.abs(H))])

_=interact(LP_synth_fsampling, fc=widgets.FloatSlider(min=0, max=1, step
    =0.01,value=0.2),
        L=widgets.IntSlider(min=1,max=200,value=10), plot_impresp=False)
```



This is a variation on the interactive widgets example, where we do not use the interact function but rather directly the Jupyter widgets.

```
from ipywidgets import widgets
out = widgets.Output()
```

```
@out.capture(clear_output=True, wait=True)
def wLP_synth_fsampling():
    fc = fcw.value
    L = Lw.value
    plot_impresp = imprespw.value
    LP_synth_fsampling(fc, L, plot_impresp)
    plt.show()


fcw = widgets.FloatSlider(min=0, max=1, step=0.01, value=0.2, description="
    fc")
Lw = widgets.IntSlider(min=1, max=200, value=10, description="L")
imprespw = widgets.Checkbox(value=False, description="Show impulse response"
    )
fcw.on_trait_change(wLP_synth_fsampling, name="value")
Lw.on_trait_change(wLP_synth_fsampling, name="value")
imprespw.on_trait_change(wLP_synth_fsampling, name="value")

c = widgets.HBox(children=[fcw, Lw])
#d=widgets.VBox(children=[c,imprespw])
d = widgets.VBox(children=[fcw, Lw, imprespw])
d.align = "center"
d.box_style = "info"
d.layout = Layout(width='40%', align_items='baseline', border_radius=50)

display(widgets.VBox([d, out]))
```

```
Widget Javascript not detected.  It may not be installed or enabled properly.
```

11.2.2 Synthesis by the window method

The window method for filter design is an easy and robust method. It directly relies on the use of the convolution theorem and its performance are easily understood in terms of the same convolution theorem. Since what we need is an impulse response associated with an "ideal" transfer function, the first step consists in computing the discrete-time inverse Fourier transform of the ideal Fourier transform:

$$H(f) \to h(n).$$

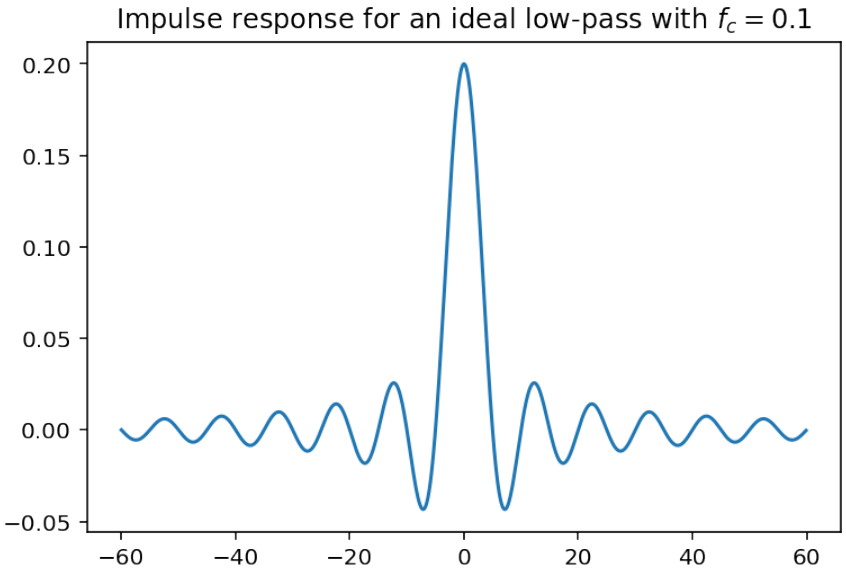
- Of course, this step would require by hand calculations, or a symbolic computation system. This leads to many exercises for students in traditional signal processing.
- In practice, one often begins with a precise numerical representation of the ideal filter and obtain the impulse response by IDFT. In this sense, the method is linked with synthesis by frequency sampling seen above.

If we begin with a transfer function which is only specified in magnitude, and if we choose to consider it as purely real, then the impulse response is even, thus non-causal. Furthermore, when the transfer function is band-limited, then its inverse transform has infinite duration. This is a consequence of the uncertainty principle for the Fourier transform. Hence, we face two problems: 1. the impulse response is non-causal, 2. it has infinite support.

A simple illustration is the following. If we consider an ideal low-pass filter, with cut-off frequency f_c, then its inverse Fourier transform is a cardinal sine

$$h(n) = 2f_c \text{sinc}\,(2\pi f_c n).$$

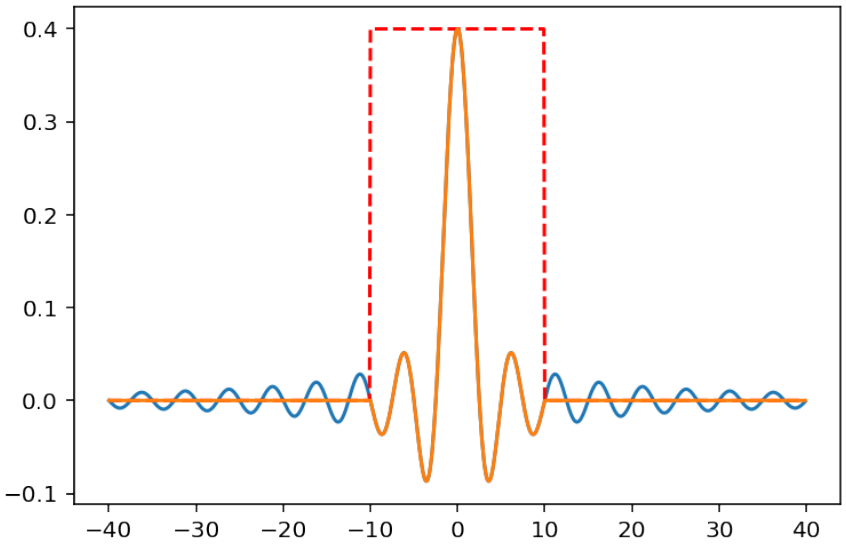
```
fc=0.1; N=60; n=np.arange(-N,N, 0.1)
plt.plot(n, 2*fc*np.sinc(2*fc*n))
_=plt.title("Impulse response for an ideal low-pass with $f_c={}$".format(fc
    ))
```



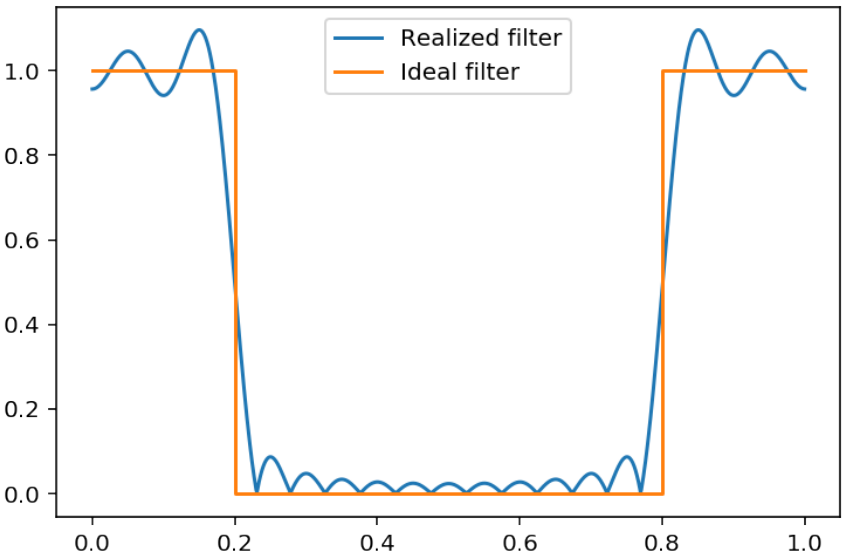
In order to get a finite number of points for our filter, we have no other solution but truncate the impulse response. Beware that one (you) need to keep both the positive and negative indexes. To get a causal system, it then suffices to shift the impulse response by the length of the non causal part. In the case of our ideal low-pass filter, this gives:

```
# L: number of points of the impulse response (odd)
L = 21
M = (L - 1) // 2
fc = 0.2
N = 40
step = 0.1
invstep = int(1 / step)
n = np.arange(-N, N, step)
h = 2 * fc * np.sinc(2 * fc * n)
plt.plot(n, h)
w = np.zeros(np.shape(n))
w[where(abs(n * invstep) < M * invstep)] = 1
plt.plot(n, 2 * fc * w, '--r')
ir_w = np.zeros(np.shape(n))
ir_w[where(abs(n * invstep) < M * invstep)] = h[where(
    abs(n * invstep) < M * invstep)]
#plt.figure();
_ = plt.plot(n, ir_w)
```

Then the realized transfer function can be computed and compared with the ideal filter.



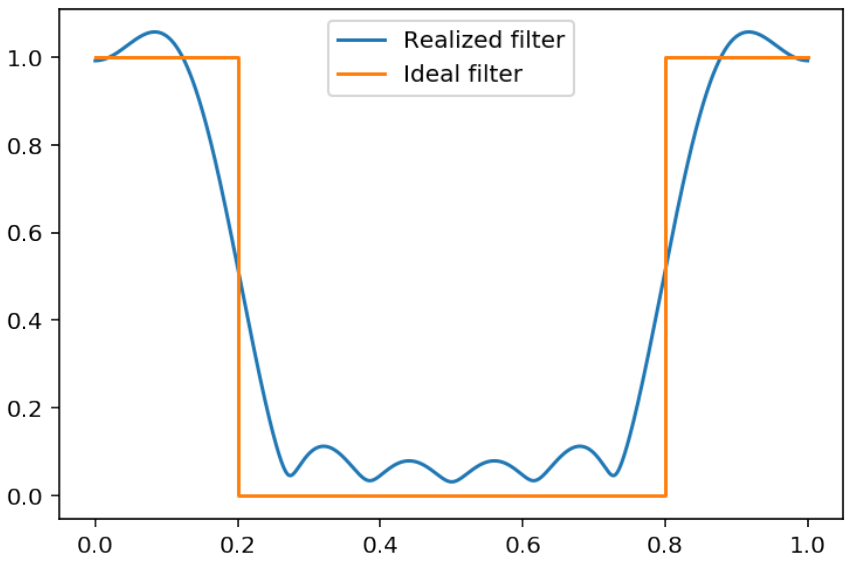

```
H_w = fft(ir_w[::invstep], 1000)
f = np.linspace(0, 1, 1000)
plt.plot(f, np.abs(H_w), label="Realized filter")
plt.plot(
    [0, fc, fc, 1 - fc, 1 - fc, 1], [1, 1, 0, 0, 1, 1], label="Ideal filter"
        )
_ = plt.legend(loc='best')
```



We observe that the frequency response presents ripples in both the band-pass and the stop-band. Besides, the transition bandwidth, from the band-pass to the stop-band is large. Again, we can put all the previous commands in the form of a function, and experiment interactively with the parameters.

```
def LP_synth_window(fc=0.2, L=21, plot_impresp=False):
```

```
    # L: number of points of the impulse response (odd)
    M = (L - 1) // 2
    step = 0.1
    invstep = int(1 / step)
    n = np.arange(-M - 5, M + 5, step)
    h = 2 * fc * np.sinc(2 * fc * n)
    w = np.zeros(np.shape(n))
    w[where(abs(n * invstep) < M * invstep)] = 1
    ir_w = np.zeros(np.shape(n))
    ir_w[where(abs(n * invstep) < M * invstep)] = h[where(
        abs(n * invstep) < M * invstep)]
    #plt.figure();
    if plot_impresp:
        plt.figure()
        plt.plot(n, w, '--r')
        _ = plt.plot(n, ir_w)
        plt.figure()

    H_w = fft(ir_w[::invstep], 1000)
    f = np.linspace(0, 1, 1000)
    plt.plot(f, np.abs(H_w), label="Realized filter")
    plt.plot(
        [0, fc, fc, 1 - fc, 1 - fc, 1], [1, 1, 0, 0, 1, 1],
        label="Ideal filter")
    plt.legend(loc='best')
    #return ir_w


_ = interact(
    LP_synth_window,
    fc=widgets.FloatSlider(
        min=0, max=0.49, step=0.01, value=0.2),
    L=widgets.IntSlider(
        min=1, max=200, value=10),
    plot_impresp=False)
```



We observe that the transition bandwidth varies with the number of points kept in the imulse response: and that the larger the number of points, the thinner the transition. We also observe that though the ripples oscillations have higher frequency, their amplitude do no change with the number of points.

There is a simple explanation for these observations, as well as directions for improvement. Instead of the rectangular truncating as above, it is possible to consider more general weight functions, say $w(n)$ of length N. The true impulse response is thus *apodized* (literal translation: "removing the foot") by multiplication with the window function:

$$h_w(n) = h(n)w(n).$$

By the Plancherel theorem, we immediately get that

$$H_w(h) = [H * W](f).$$

The resulting filter is thus the convolution of the ideal response with the Fourier transform of the window function.

In the example above, the window function is rectangular. As is now well know, its Fourier transform is a discrete cardinal sine (a ratio of two sine)

$$W(f) = \frac{\sin(\pi f(2L+1))}{(2L+1)\sin(\pi f)}.$$

Hence, the realized filter results from the convolution between the reactangle representing the ideal low-pass with a cardinal sine. This yields that the transition bandwidth is essentially given by the integral of the main lobe of the cardinal sine, and that the amplitude of the ripples are due to the integrals of the sidelobes. In order to improve the synthetiszed filter, we can adjust the number of taps of the impulse response and/or choose another weight window.

Many window functions have been proposed and are used for the design of FIR filters. These windows are also very useful in spectrum analysis where the same kind of problems – width of a main lobe, ripples due to the side-lobes, are encountered. A series of windows is presented in the following table. Many other windows exist, and entire books are devoted to their characterization.

Window	$\mathbf{w(n)}$	■f	PSLL
Rectangular (boxcar)	1	1	-13.252
Triangular (Bartlett)	$w(n) = 1-(n-(N-1)/2)/N$	2	-26.448
Hann($\cos^2$)	$0.5-0.5\cos\left(\frac{2\pi n}{M-1}\right)$	2	-31.67
Nuttal	$--$	2.98	-68.71
Hamming	$0.54-0.46\cos\left(\frac{2\pi n}{M-1}\right)$	3	-42.81
Bohman	$--$	3	-45.99
Blackman	$0.42-0.5\cos\left(\frac{2\pi n}{M-1}\right)+0.08\cos\left(\frac{4\pi n}{M-1}\right)$	3	-58.11
Flat-top	$--$	4.96	-86.52
Blackman-Harris	$--$	4	-92

This table present the characteristics of some popular windows. Δf represents the width of the main-lobe, normalized to $1/N$. PSLL is the abbreviation of Peal to Side-lobe leakage, and corresponds to the maximum leakage between the amplitude of the main-lobe and the amplitudes of the side-lobes. In the table, we have not reported too complicated expressions, defined on intervals or so. For example, the 4 terms Blackman-Harris window, which performs very well, has the expression

$$w(n) = \sum_{k=0}^{3} a_k \cos\left(\frac{2\pi kn}{M-1}\right)$$

with $[a_0, a_1, a_2, a_3] = [0.35875, 0.48829, 0.14128, 0.01168]$. The Kaiser-Bessel window function also performs very well. Its expression is

$$w(n) = I_0\left(\beta\sqrt{1-\frac{4n^2}{(M-1)^2}}\right)/I_0(\beta),$$

where I_0 is the modified zeroth-order Bessel function. The shape parameter β determines a trade-off between main-lobe width and side lobe level. As β gets large, the window narrows.

See the detailed table in the book "Window Functions and Their Applications in Signal Processing" by Prabhu (2013). We have designed a displaying and comparison tool for the window functions. The listing is provided in the appendix, but for now, readers of the IPython notebook version can experiment a bit by issuing the command `%run windows_disp.ipy`.

```
"""This is from scipy.signal.get_window() help
List of windows:
boxcar, triang, blackman, hamming, hann, bartlett, flattop,
        parzen, bohman, blackmanharris, nuttall, barthann,
        kaiser (needs beta), gaussian (needs std),
        general_gaussian (needs power, width),
        slepian (needs width), chebwin (needs attenuation)"""
windows=['boxcar', 'triang', 'blackman', 'hamming', 'hann', 'bartlett', '
    flattop',
        'parzen', 'bohman', 'blackmanharris', 'nuttall', 'barthann']
windows_1parameter=['kaiser', 'gaussian', 'slepian', 'chebwin']
windows_2parameter=['general_gaussian']
```

```
%run windows_disp_jup.ipy
```

```
Widget Javascript not detected.  It may not be installed or enabled properly.
```

The main observation is that with N fixed, we have a trade-off to find between the width of the main lobe, thus of the transition width, and the amplitude of the side-lobes. The choice is usually done on a case by case basis, which may also include other parameters. To sum it all up, the window method consists in:

- calculate (or approximate) the impulse response associated with an ideal impulse response,
- choose a number of samples, and a window function, and apodize the impulse response. The choice of the number of points an window function can also be motivated by maximum level of ripples in the band pass and/or in the stop band.
- shift the resulting impulse response by half the number of samples in order to obtain a causal filter.

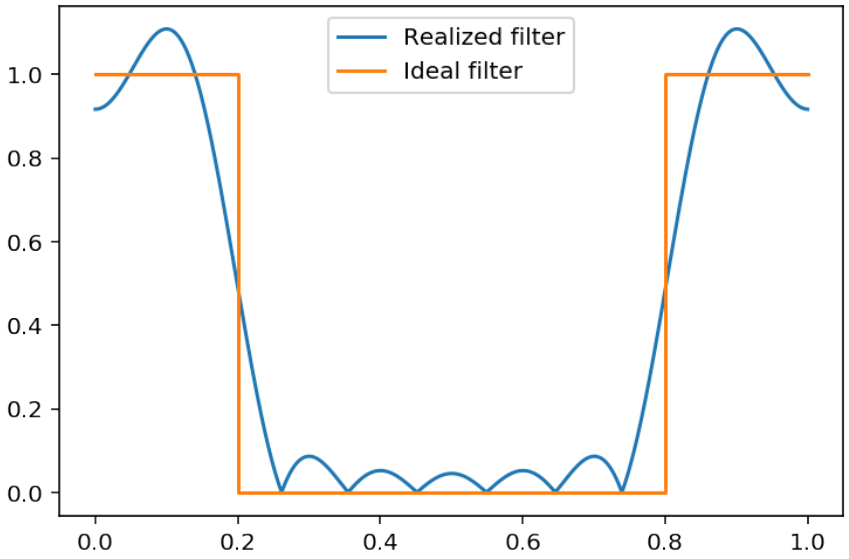
It is quite simple to adapt the previous script with the rectangular window to accept more general windows. This is done by adding a parameter `window`.

```
def LP_synth_genwindow(fc=0.2,
                       L=21,
                       window='boxcar',
                       plot_impresp=False,
                       plot_transferfunc=True):

    # L: number of points of the impulse response (odd)
```

```
    M = (L - 1) // 2
    step = 1
    invstep = int(1 / step)
    n = np.arange(-M, M + 1, step)
    h = 2 * fc * np.sinc(2 * fc * n)
    w = sig.get_window(window, 2 * M + 1)
    ir_w = w * h
    #plt.figure();
    if plot_impresp:
        plt.figure()
        plt.plot(n, w, '--r', label="Window function")
        _ = plt.plot(n, h, label="Initial impulse response")
        _ = plt.plot(n, ir_w, label="Windowed impulse response")
        plt.legend()

    H_w = fft(ir_w[::invstep], 1000)
    if plot_transferfunc:
        plt.figure()
        f = np.linspace(0, 1, 1000)
        plt.plot(f, np.abs(H_w), label="Realized filter")
        plt.plot(
            [0, fc, fc, 1 - fc, 1 - fc, 1], [1, 1, 0, 0, 1, 1],
            label="Ideal filter")
        plt.legend(loc='best')

    return ir_w


w = interactive(
    LP_synth_genwindow,
    fc=widgets.FloatSlider(
        min=0, max=0.49, step=0.01, value=0.2),
    L=widgets.IntSlider(
        min=1, max=200, value=10),
    window=widgets.Dropdown(options=windows),
    plot_impresp=False,
    plot_transferfunc=True)
w
```

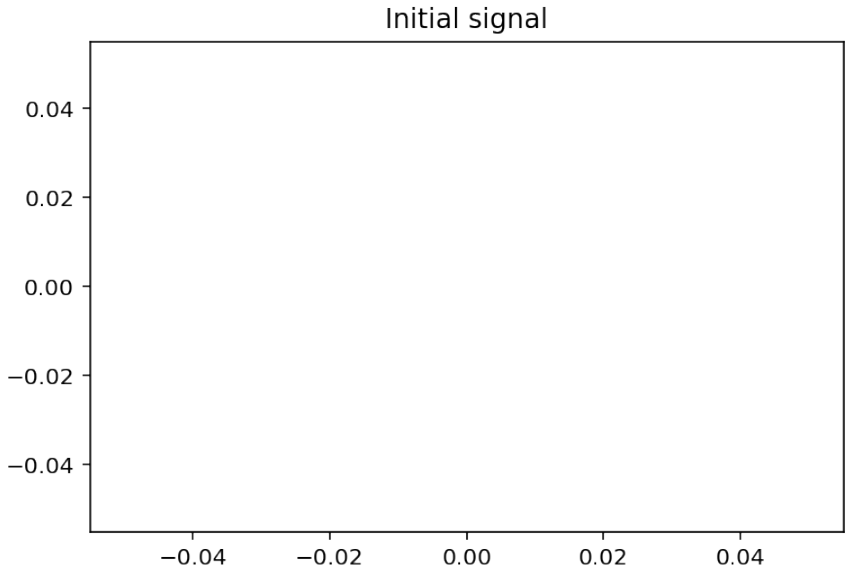
Exercise 9. *The function* `LP_synth_genwindow` *returns the impulse response of the synthetized filter. Create a signal* $x_{test} = \sin(2\pi f_0 n) + \sin(2\pi f_1 n) + \sin(2\pi f_2 n)$*, with* $f_0 = 0.14$*,* $f_1 = 0.24$*,* $f_2 = 0.34$ *and filter this signal with the synthetized filter, for* $f_c = 0.2$*,* $L = 50$*, and for a* `hamming` *window. Comment on the results.*

```
# define constants
n = np.arange(0, 100)
f0, f1, f2 = 0.14, 0.24, 0.34
# the test signal
xtest = 0  # Complete here
plt.plot(xtest)
plt.title("Initial signal")
# compute the filter
#h1 = LP_synth_genwindow(
# Complete here
#)
# then filter the signal
y1 = 0  # Complete here
#and display it
```



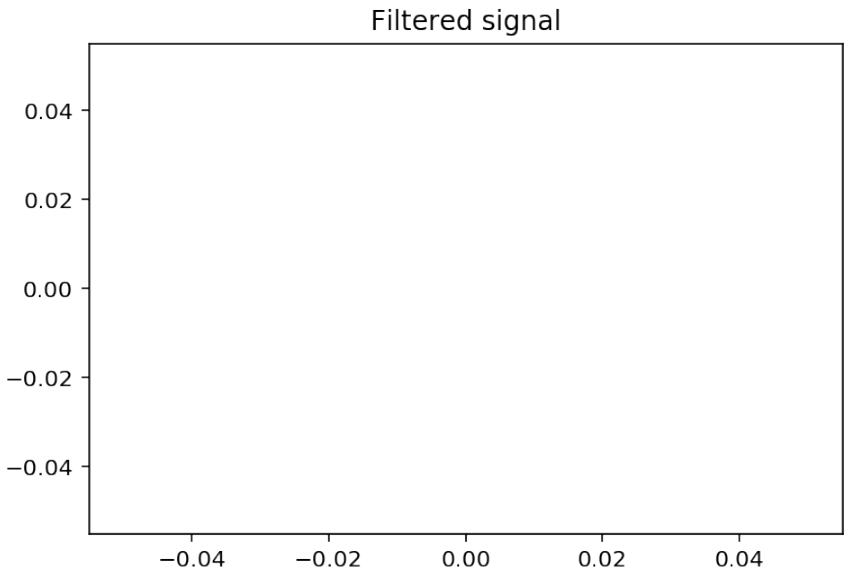

```
plt.figure()
plt.plot(y1)
plt.title("Filtered signal")
```

```
Text(0.5, 1.0, 'Filtered signal')
```



Solution

```
# define constants
n = np.arange(0, 100)
f0, f1, f2 = 0.14, 0.24, 0.34
```



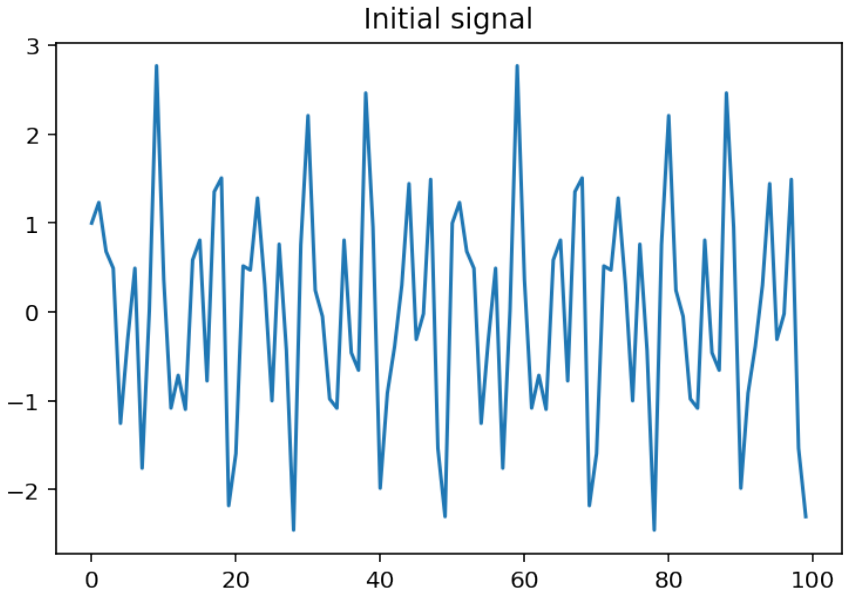

```
# the test signal
xtest = sin(2 * pi * f0 * n) + sin(2 * pi * f1 * n) + cos(2 * pi * f2 * n)
plt.plot(xtest)
plt.title("Initial signal")
# compute the filter
h1 = LP_synth_genwindow(
    fc=0.2,
    L=50,
    window='hamming',
    plot_impresp=False,
    plot_transferfunc=False)
# then filter the signal
y1 = sig.lfilter(h1, [1], xtest)
#and display it
plt.figure()
plt.plot(y1)
plt.title("Filtered signal")
```

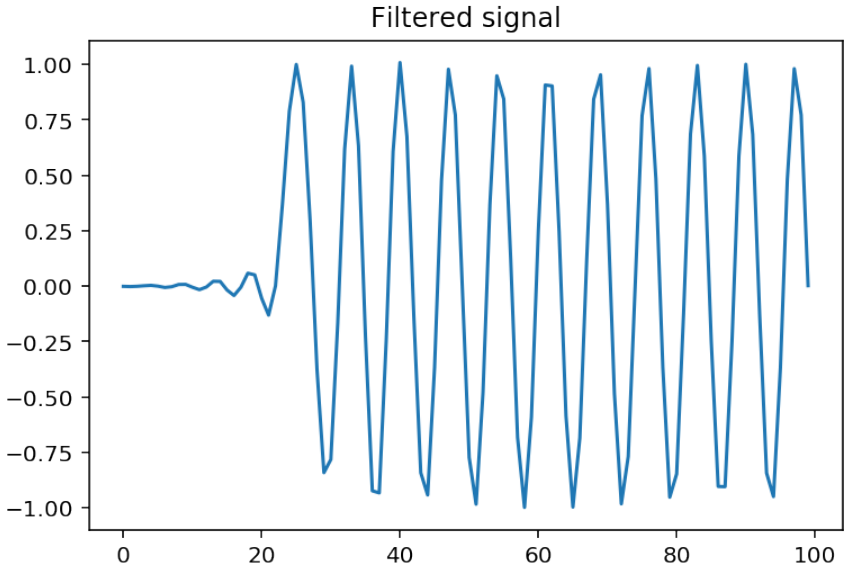
```
Text(0.5, 1.0, 'Filtered signal')
```

The whole synthesis workflow for the window method is available in two specialized functions of the scipy library. Nowadays, it is really useless to redevelop existing programs. It is much more interesting to gain insights on what is really done and how things work. This is actually the goal of this lecture. The two functions avalaible in `scipy.signal` are `firwin` and `firwin2`.

Exercise 10. *Use one of these functions to design a high-pass filter with cut-off frequency at* $f_c = 0.3$. *Filter the preceding signal* x_{test} *and display the results.*

```
# define constants
n = np.arange(0, 200)
f0, f1, f2 = 0.14, 0.2, 0.34
# the test signal
```


Initial signal
3
2
1
0
−1
−2
0
20
40
60
80
100


Filtered signal
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
20
40
60
80
100

```
xtest = sin(2 * pi * f0 * n) + sin(2 * pi * f1 * n) + sin(2 * pi * f2 * n)
plt.plot(xtest)
plt.title("Initial signal")
figcaption("Initial signal")
# compute the filter
h1 = sig.firwin(
    101,
    0.3,
    width=None,
    window='hamming',
    pass_zero=False,
    scale=True,
    nyq=0.5)
plt.figure()
plt.plot(np.linspace(0, 1, 1000), abs(fft(h1, 1000)))
plt.xlabel("Frequency")
plt.title("Transfer function")
figcaption("Transfer function")
# then filter the signal
y1 = sig.lfilter(h1, [1], xtest)
#and display it
plt.figure()
plt.plot(y1)
plt.title("Filtered signal")
figcaption("Filtered signal")
```

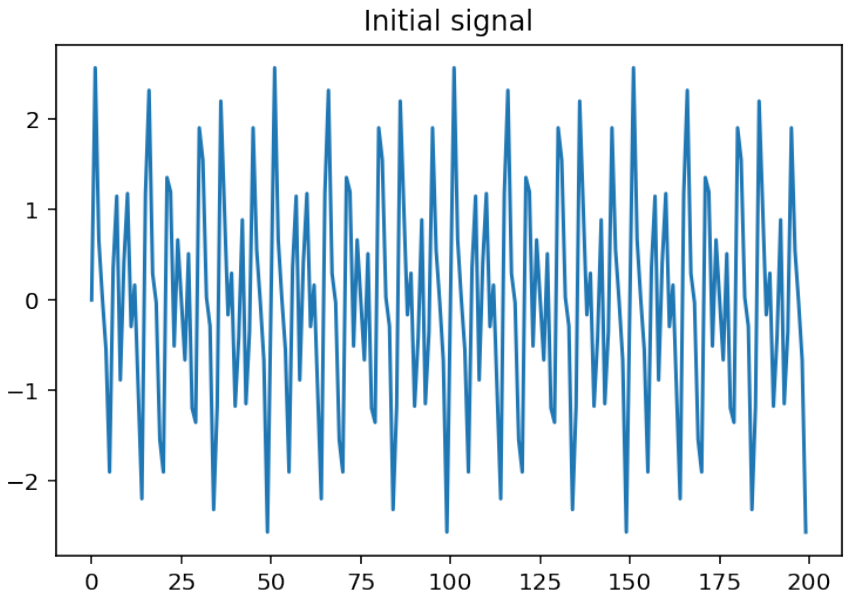


Figure 11.11: Filtered signal

11.3 Synthesis of IIR filters by the bilinear transformation method

A simple and effective method for designing IIR digital filters with prescribed magnitude response specifications is the bilinear transformation method. The point is that already exists well-known optimal methods

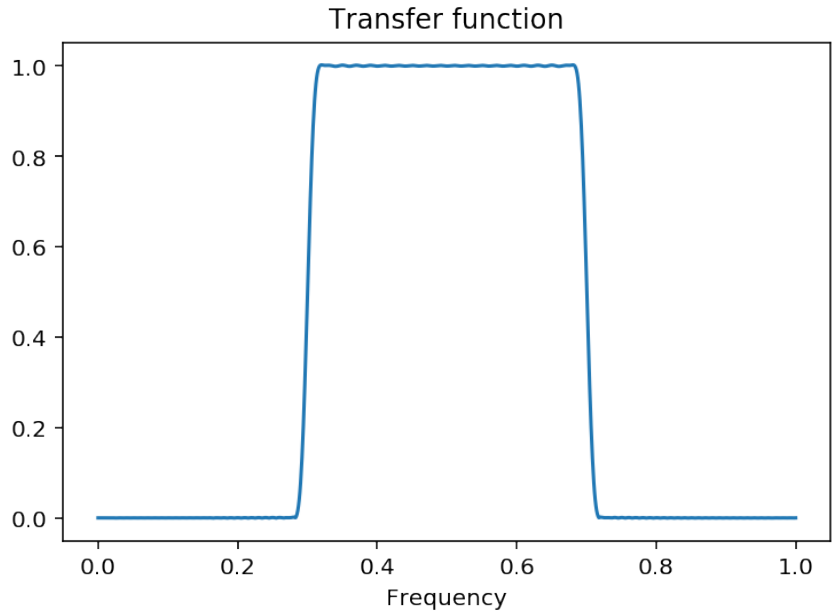


Figure 11.12: Transfer function

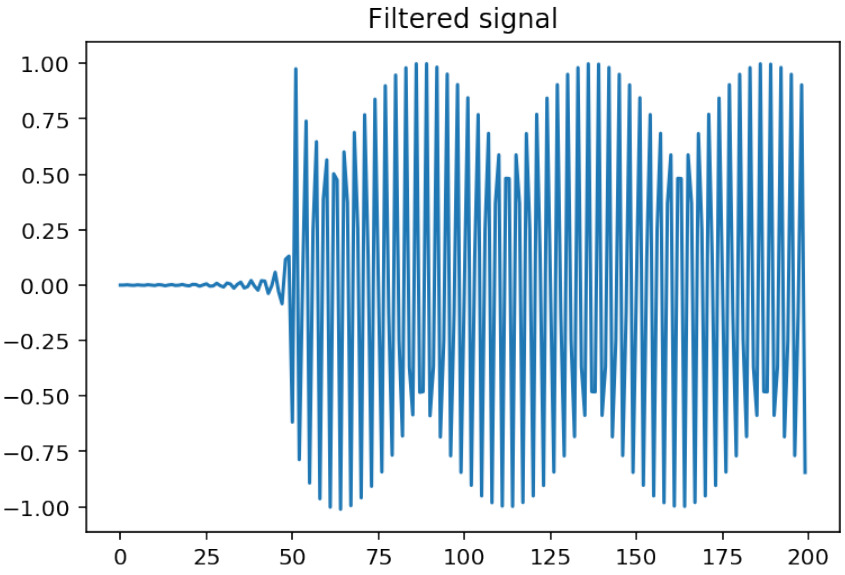


Figure 11.13: Initial signal

for the design of analog filters, such as Butterworth, Chebyshev, or elliptic filter designs. Then, the idea is to map the digital filter into an equivalent analog filter, which can be designed optimally, and map back the design to the digital domain. The key for this procedure is to dispose of a reversible mapping from the analog domain to the digital domain.

11.3.1 The bilinear transform

Recall that in the analog domain, the equivalent of the Z-transform is the Laplace transform which associates a signal $s(t)$ with a function $S(p)$ of the complex variable p. When p lives on the imaginary axis of the complex plane, then the Laplace transform reduces to the Fourier transform (for causal signals). For transfer functions, stability is linked to the positions of poles in the complex plane. They must have a negative real part (that is belong to the left half plane) to ensure the stability of the underlying system.

The formula for the bilinear transform comes from a formal analogy between the derivation operator in the Laplace and Z domains.

The bilinear transform makes an association between analog and digital frequencies, as follows:

$$p = k\frac{1-z^{-1}}{1+z^{-1}}, \tag{11.3}$$

where k is an arbitrary constant. The usual derivation leads to $k = 2/T_s$, where T_s is the sampling period. However, using a general parameter k does not change the methodology and offers a free parameter that enables to simplify the procedure.

The point is that this transform presents some interesting and useful features:

1. It preserves stability and minimum phase property (the zeros of the transfer function are with negative real part (analog case) or are inside the unit circle (discrete case).

2. It maps the infinite analog axis into a periodic frequency axis in the frequency domain for discrete signals. That mapping is highly non linear and warp the frequency components, but it recovers the well-known property of periodicity of the Fourier transform of discrete signals.

The corresponding mapping of frequencies is obtained as follows. Letting $p=j\omega_a = j2\pi f_a$ and $z = \exp(\omega_d) = \exp(j2\pi f_d)$. Plugging this in (11.3), we readily obtain

$$\omega_a = k\tan\left(\frac{\omega_d}{2}\right), \tag{11.4}$$

or

$$\omega_d = 2\arctan\left(\frac{\omega_a}{k}\right). \tag{11.5}$$

The transformation (11.4) corresponds to the initial transform of the specifications in the digital domain into analog domain specifications. It is often called a *pre-warping* . Figure 11.14 shows the mapping of pulsations from one domain into the other one.

```
k = 2
xmin = -5 * pi
xmax = -xmin
omegaa = np.arange(xmin, xmax, 0.1)
omegad = 2 * np.arctan(omegaa / k)
plt.plot(omegaa, omegad)
plt.plot([xmin, xmax], [-pi, -pi], '--', color='lightblue')
plt.plot([xmin, xmax], [pi, pi], '--', color='lightblue')
```

```
#plt.text(-3.7,0.4,'Fs/2', color='blue',fontsize=14)
plt.xlabel("Analog pulsations $\omega_a$")
plt.ylabel("Digital pulsations $\omega_d$")
_ = plt.xlim([xmin, xmax])
plt.title("Frequency mapping of the bilinear transform")
figcaption("Frequency mapping of the bilinear transform", label="fig:BLT")
```

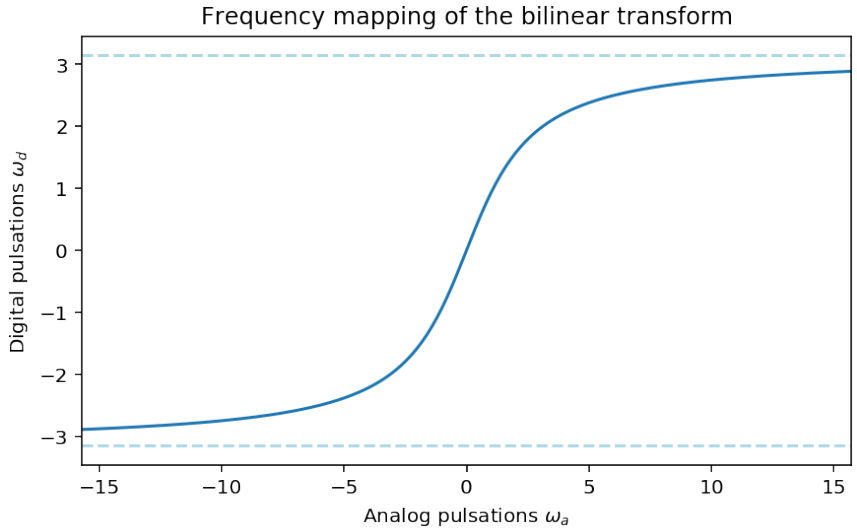


Figure 11.14: Frequency mapping of the bilinear transform

When designing a digital filter using an analog approximation and the bilinear transform, we follow these steps: Pre-warp the cutoff frequencies Design the necessary analog filter apply the bilinear transform to the transfer function Normalize the resultant transfer function to be monotonic and have a unity passband gain (0dB).

11.3.2 Synthesis of low-pass filters – procedure

Let ω_p and ω_s denote the edges of the pass-band and of the stop-band.

1. For the synthesis of the analog filter, it is convenient to work with a normalized filter such that $\Omega_p = 1$. Therefore, as a first step, we set
$$k = \arctan(\omega_p/2)$$
which ensures that $\Omega_p = 1$. Then, we compute $\Omega_s = 2\arctan(\omega_s/k)$.

2. Synthetize the optimum filter in the analog domain, given the type of filter, the frequency and gain constraints. This usually consists in determining the order of the filter such that the gain constraints (ripples, minimum attenuation, etc) are satisfied, and then select the corresponding polynomial. This yields a transfer function $H_a(p)$.

3. Map back the results to the digital domain, using the bilinear transform (11.3), that is compute
$$\boxed{H(z) = H_a(p)\big|_{p=k\frac{1-z^{-1}}{1+z^{-1}}}}$$

Exercise 11. *We want to synthetize a digital filter with $f_p = 6kHz$, with a maximum attenuation of -3dB, and a stop-band frequency of $f_s = 9kHz$, with a minimum attenuation of -9dB. The Nyquist rate (sampling frequency) is $F_s = 36kHz$.*

- *Represent the template for this sythesis,*
- *Compute the pulsations in the analog domain (fix $\Omega_p = 1$).*
- *if we choose a Butterworth filter, the best order is $n = 2$ and the corresponding polynomial is $D(p) = p^2 + \sqrt{2}p + 1$, and the transfer function is $H_a(p) = 1/D(p)$. Compute $H(z)$.*
- *Plot $H(f)$. Use `sig.freqz` for computing the transfer function. We also provide a function `plt_LPtemplate(omega, A, Abounds=None)` which displays the template of the filter. Import it using `from plt_LPtemplate import *`.*

Elements of solution

- $k = 1/\tan(\pi/6) = \sqrt{3}$
- $\Omega_s = k\tan(\pi/4) = \sqrt{3}$
- $$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{(4+\sqrt{6}) - 4z^{-1} + (4-\sqrt{6})z^{-2}}$$

```
# compute the transfer function using freqz
w, H = sig.freqz([1, 2, 1], [4 + sqrt(6), -4, 4 - sqrt(6)], whole=True)
# plot the result --w-pi corresponds to a shift of the pulsation
#axis associated with the fftshift of the transfer function.
plt.plot(w - pi, 20 * np.log10(fftshift(np.abs(H))))
# plot the filter template
from plt_LPtemplate import *
plt_LPtemplate([pi / 3, pi / 2], [-3, -9], Abounds=[5, -35])
plt.title("Realized filter and its template")
figcaption("Realized filter and its template")
```

```
/home/bercherj/.local/lib/python3.5/site-packages/ipykernel_launcher.py:5: RuntimeW
  """
```

In practice, the transfer function can be generated by transforming the poles and zeros of the analog filter into the poles and zeros of the digital filter. This is simply done using the transformation

$$z = \frac{1 + p/k}{1 - p/k}.$$

It remains a global gain that can be fixed using a value at $H(1)$ ($\omega = 0$).This dramatically simplifies the synthesis in practice.

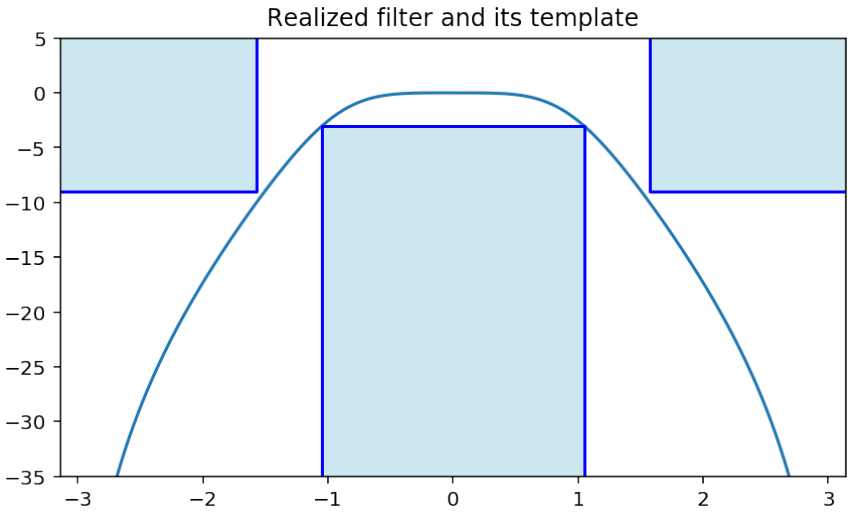


Figure 11.15: Realized filter and its template

11.3.3 Synthesis of other type of filters

For other kind of filters (namely band-pass, high-pass, band-stop), we can either: - use the bilinear transform and the pre-warping to obtain a filter of the same type in the analog domain; then transferring the problem to the analog designer. - use an auxiliary transform that converts a digital low-pass filter into another type of filter. Using the second option, we see that the low-pass synthesis procedure has to be completed with

0. Transform the specifications of the digital filter into specifications for a low-pass digital filter.

Transposition from a low-pass filter (ω_p) to another type

Let ω_1 and ω_2 denote the low and high cut-off frequencies (only ω_1 for the transposition of a low-pass into a high-pass). These transformations preserve the unit circle. That is, $z = e^{j\theta}$ is transformed into $z = e^{j\theta'}$. There is an additional frequency warping, but the notion of frequency is preserved. 1. **low-pass ω_p– low-pass ω_1**

$$z^{-1} \rightarrow \frac{z^{-1} - \alpha}{1 - \alpha z^{-1}}$$

with

$$\alpha = \frac{\sin\left(\frac{\omega_p - \omega_1}{2}\right)}{\sin\left(\frac{\omega_p + \omega_1}{2}\right)}$$

3. **low-pass ω_p– band-pass $\omega_1, \omega_2$$**$z^{-1} \rightarrow -\frac{z^{-2} - \frac{2\alpha\beta}{\beta+1} z^{-1} + \frac{\beta-1}{\beta+1}}{\frac{\beta-1}{\beta+1} z^{-2} - \frac{2\alpha\beta}{\beta+1} z^{-1} + 1}$ with

$$\alpha = \frac{\cos\left(\frac{\omega_p + \omega_1}{2}\right)}{\cos\left(\frac{\omega_p - \omega_1}{2}\right)}$$

and

$$\beta = \tan\left(\frac{\omega_p}{2}\right) \tan\left(\frac{\omega_2 - \omega_1}{2}\right)$$

Exercise 12. *We want to synthetize an high-pass digital filter with edge frequencies* $2,4,12$ *and* 14 *kHz, with a maximum attenuation of 3dB in the band-pass, and a minimum attenuation of -12dB in the stop-band. The Nyquist rate (sampling frequency) is* $F_s = 32kHz$.

- *Represent the template for this sythesis,*
- *Compute the pulsations in the analog domain (fix* $\Omega_p = 1$*).*
- *if we choose a Butterworth filter, the best order is* $n = 2$ *and the corresponding polynomial is* $D(p) = p^2 + \sqrt{2}p + 1$*, and the transfer function is* $H_a(p) = 1/D(p)$*. Compute* $H(z)$*.*
- *Plot* $H(f)$*. Use* `sig.freqz` *for computing the transfer function.*

** Sketch of solution** - normalized pulsations: $\omega_0 = \frac{\pi}{8}, \omega_1 = \frac{\pi}{4}, \omega_2 = \frac{3\pi}{4}, \omega_3 = \frac{7\pi}{8}$ - transposition into a digital low-pass $\alpha = 0, \beta = \tan(\omega_p/2)$. Choosing $\beta = 1$, we get $\omega_p = \pi/2$ - the transform is thus $z^{-1} \to -z^{-2}$ - In the BLT, we take $k = 1$; thus $\omega_s = 1$ and $\omega_s = \tan(3\pi/8)$ - Finally, we obtain

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

for the digital low-pass, which after the transform $z^{-1} \to -z^{-2}$ gives

$$H(z) = \frac{1 - 2z^{-2} + z^{-4}}{(2+\sqrt{2}) + (2-\sqrt{2})z^{-4}}$$

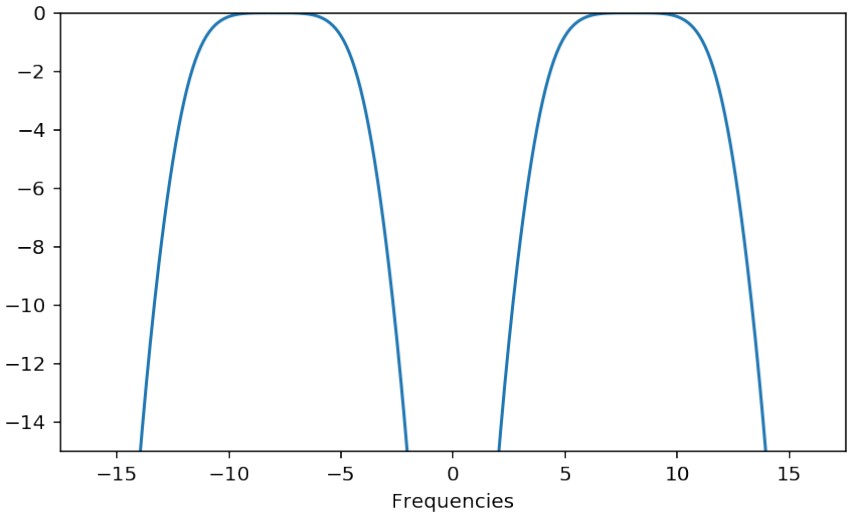
```
# compute the transfer function using freqz
w, H = sig.freqz(
    [1, 0, -2, 0, 1], [2 + sqrt(2), 0, 0, 0, 2 - sqrt(2)], whole=True)
# plot the result --w-pi corresponds to a shift of the pulsation
#axis associated with the fftshift of the transfer function.
plt.plot((w - pi) / (2 * pi) * 32, 20 * np.log10(fftshift(np.abs(H))))
plt.xlabel("Frequencies")
_ = plt.ylim([-15, 0])
```

```
/home/bercherj/.local/lib/python3.5/site-packages/ipykernel_launcher.py:6: RuntimeW
```

11.3.4 Numerical results

Finally, we point out that these procedures have been systematized into computer programs. Two functions are available in scipy to design an IIR filter: `sig.iirfilter` that computes the coefficients of a filter for a given order, and `sig.iirdesign` that even determine a correct order given constraint on maximum and/or minimum attenuation. It is instructive to consult the help of these functions (e.g. `help(sig.iirdesign)`) and try to reproduce the results we obtained analytically above. Possible solutions are provided below.

```
b, a = sig.iirfilter(
    2, [1 / (pi)],
    rp=None,
    rs=None,
    btype='lowpass',
```




```
    analog=False,
    ftype='butter',
    output='ba')
# compute the ttransfer function using freqz
w, H = sig.freqz(b, a, whole=True)
# plot the result --w-pi corresponds to a shift of the pulsation
#axis associated with the fftshift of the transfer function.
plt.plot(w - pi, 20 * np.log10(fftshift(np.abs(H))))
# plot the filter template
from plt_LPtemplate import *
plt_LPtemplate([pi / 3, pi / 2], [-3, -9], Abounds=[12, -35])
plt.title("Realized filter and its template")
figcaption("Realized filter and its template")
```

```
/home/bercherj/.local/lib/python3.5/site-packages/ipykernel_launcher.py:13: Runtime
  del sys.path[0]
```

```
b, a = sig.iirdesign(
    [4 / 16, 12 / 16], [2 / 16, 14 / 16],
    3,
    12,
    analog=False,
    ftype='butter',
    output='ba')
# compute the ttransfer function using freqz
w, H = sig.freqz(b, a, whole=True)
# plot the result --w-pi corresponds to a shift of the pulsation
#axis associated with the fftshift of the transfer function.
plt.plot((w - pi) / (2 * pi) * 32, 20 * np.log10(fftshift(np.abs(H))))
_ = plt.ylim([-20, 0])
```

```
/home/bercherj/.local/lib/python3.5/site-packages/ipykernel_launcher.py:12: Runtime
  if sys.path[0] == '':
```

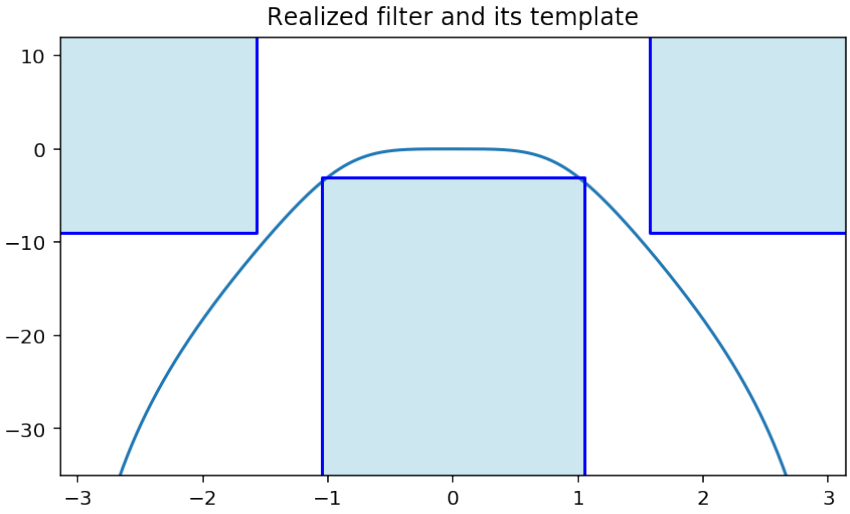
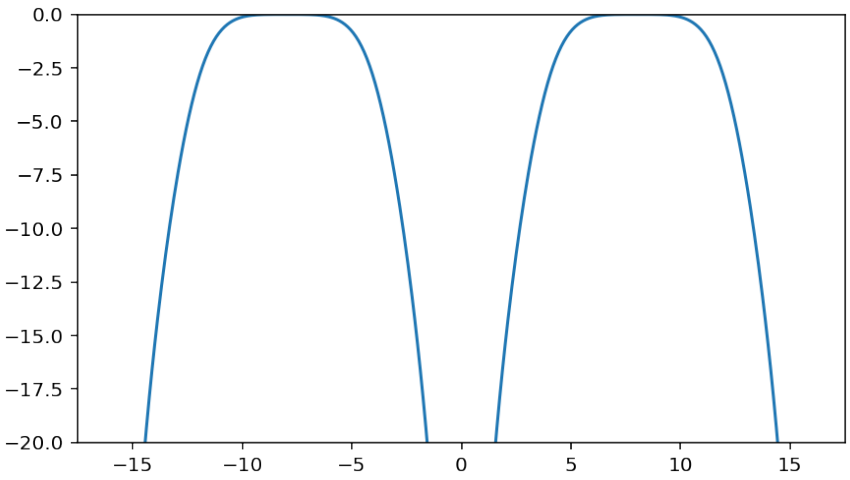


Figure 11.16: Realized filter and its template



11.4 Lab – Basic Filtering

Author: J.-F. Bercher date: november 19, 2013 Update: february 25, 2014 Last update: december 08, 2014

The goal of this lab is to study and apply several digital filters to a periodic signal with fundamental frequency f_0=200 Hz, sampled at frequency F_s=8000 Hz. This signal is corrupted by a low drift, and that is a common problem with sensor measurements. A first filter will be designed in order to remove this drift. In a second step, we will boost a frequency range withing the components of this signal. Finally, we will consider the design of a simple low-pass filter using the window method, which leads to a linear-phase filter.

This signal is contained into the vector x stored in the file `sig1.npz`. It is possible to load it via the instruction `f=np.load(sig1.npz)`

```
Fs=8000
Ts=1/Fs
```

First load all useful modules:

```
import numpy as np
from numpy import ones, zeros, abs, exp, pi, sin, real, imag
import matplotlib.pyplot as plt
import scipy.io
from scipy.signal import lfilter
from numpy.fft import fft, ifft, fftshift

%matplotlib inline
```

11.4.1 Analysis of the data

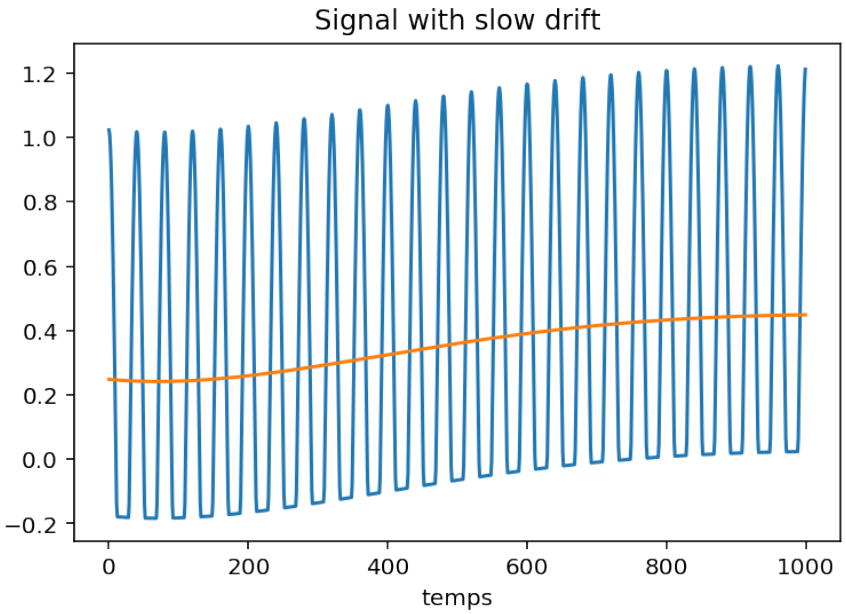
```
# utilitary function
def freq(N,Fs=1):
    """ Returns a vector of size N  of normalized frequencies
    between -Fs/2 and Fs/2 """
    return np.linspace(-0.5,0.5,N)*Fs
```

```
# To load the signal
sig1=np.load('sig1.npz')
#sig1 is a dictionnary. One can look at the keys by: sig1.keys()
m=sig1['m']
x=sig1['x']
```

```
# Time
plt.figure(1)
plt.plot(x)
plt.plot(m)
plt.title('Signal with slow drift')
plt.xlabel("temps")
```

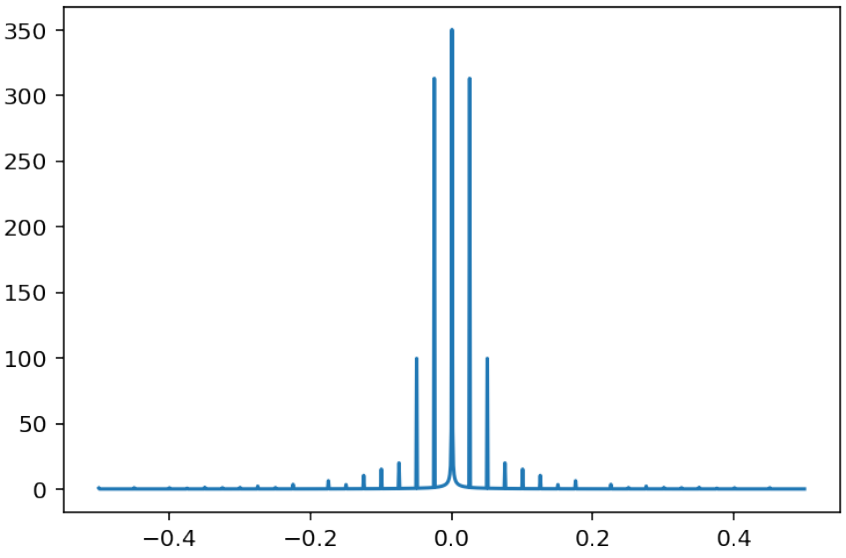
```
Text(0.5, 0, 'temps')
```

```
import mpld3
mpld3.enable_notebook()
```




```
%matplotlib inline
# Frequency representation
N=len(x)
f=freq(N)
plt.plot(f,abs(fftshift(fft(x))))
#plt.title('Fourier transform of the signal (modulus)')
```

```
[<matplotlib.lines.Line2D at 0x7fd29f447828>]
```



11.4.2 Filtering

We wish now to modify the spectral content of x using different digital filters with transfer function $H(z) = B(z)/A(z)$. A standard Python function will be particularly useful:

- *lfilter* implements the associated difference equation. This function computes the output vector y of the digital filter specified by
- the vector B (containing the coefficients of the numerator $B(z)$,
- and by the vector A of the denominator's coefficients $A(z)$, for an input vector x: `y=lfilter(B,A,x)`
- *freqz* computes the frequency response $H(e^{j2\pi f/Fs})$ in modulus and phase, for a filter described by the two vectors B and A: `freqz(B,A)`

11.4.3 Design and implementation of the lowpass averaging filter

The signal is corrupted by a slow drift of its mean value. We look for a way to extract and then remove this drift. We will denote $M(n)$ the drift, and $x_c(n)$ the centered (corrected) signal.

Theoretical part:

What analytical expression enable to compute the signal's mean on a period?

From that, deduce a filter with impulse response $g(n)$ which computes this mean $M(n)$.

Find another filter, with impulse response $h(n)$, removes this mean: $x_c(n) = x(n) - M(n) = x(n) * h(n)$. Give the expression of $h(n)$.

Also give the analytical expressions of $G(z)$ and $H(z)$.

Practical part

For the averaging filter and then for the subtracting filter:

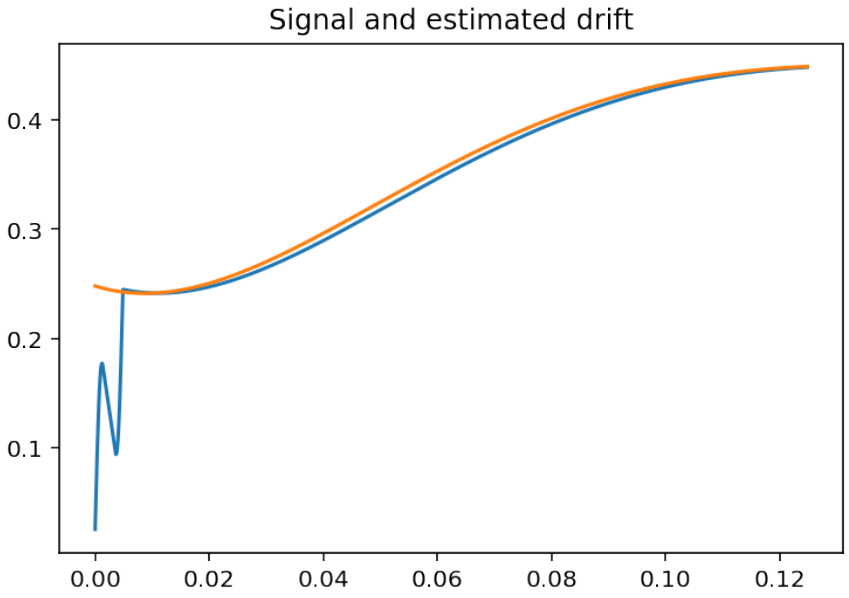
- Compute and plt.plot the two impulse responses (you may use the instruction `ones(L)` which returns a vector of L ones.
- plt.plot the frequency responses of these two filters. You may use the function `fft` which returns the Fourier transform, and plt.plot the modulus `abs` of the result.
- Filter x by these two filters. plt.plot the output signals, in the time and frequency domain. Conclude.

```
# Averaging filter
#-----------------------------------------------------------
# Filter g which computes the mean on a period of 40 samples
L=40
N=len(x)
t=np.arange(N)/Fs
h= ones(L)/L
m_estimated=lfilter(h,[1],x)
# ...
plt.plot(t,m_estimated,t,m)
plt.title('Signal and estimated drift')
```

```
#
# We check G(f)
plt.figure()
H=fft(h,1000)
plt.plot(f,350*fftshift(abs(H)))
plt.plot(f,fftshift(abs(fft(x))))

plt.xlabel('Normalized fequencies')
plt.title('Transfer Function of the Averaging Filter')

plt.figure()
plt.plot(f,abs(fftshift(fft(m_estimated))))
```

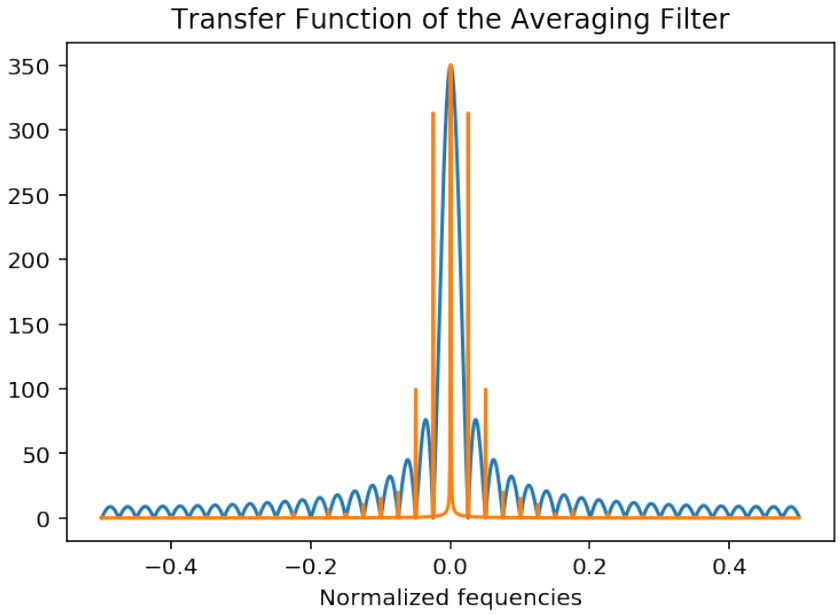
```
/usr/local/lib/python3.5/site-packages/scipy/signal/signaltools.py:1344: FutureWarn
  out = out_full[ind]
```

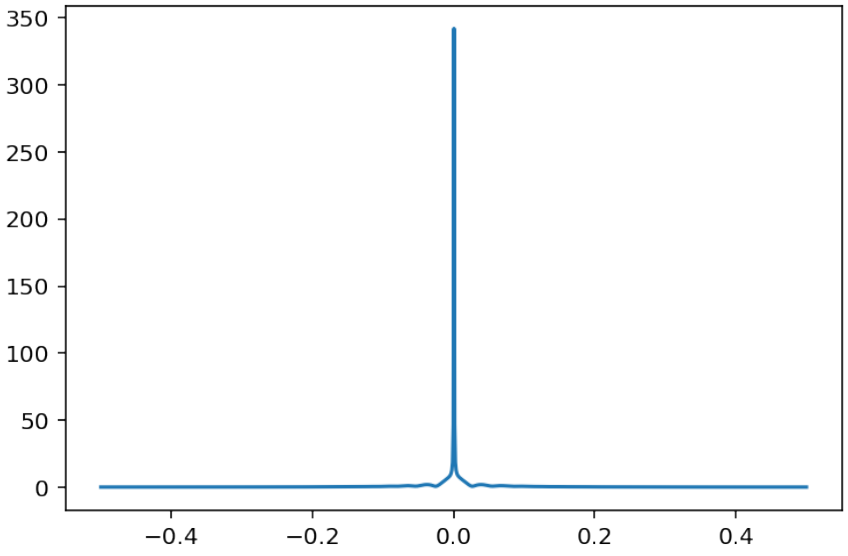
```
[<matplotlib.lines.Line2D at 0x7fd29d710320>]
```



Computation of the subtracting filter

```
# Mean subtracting filter
#-------------------------------------------------------
# The filter h subtract the mean computed over a sliding window of 40
    samples
# h may be defined as
d=zeros(L); d[0]=1
g=d-h
xc=lfilter(g, [1], x)
plt.plot(t,xc)
```

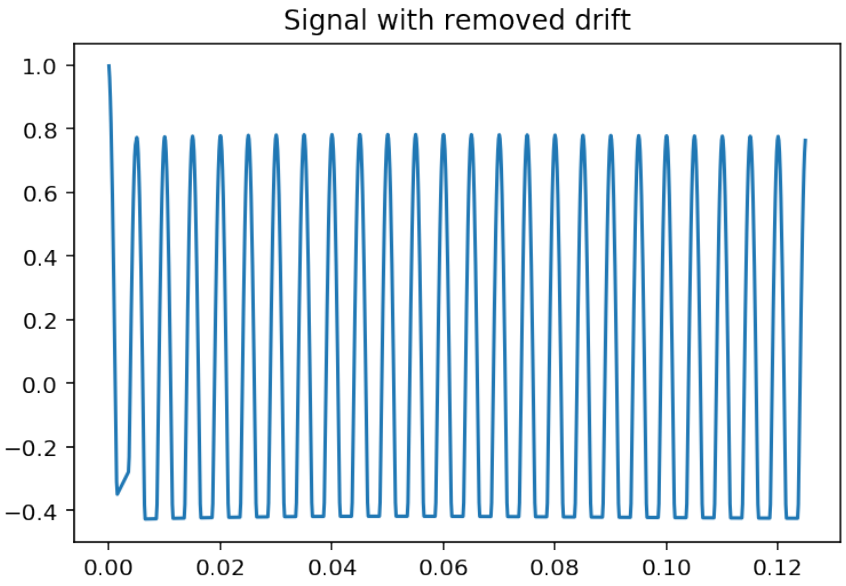

Transfer Function of the Averaging Filter
350
300
250
200
150
100
50
0
−0.4
−0.2
0.0
0.2
0.4
Normalized fequencies


350
300
250
200
150
100
50
0
−0.4
−0.2
0.0
0.2
0.4

```
plt.title('Signal with removed drift')
#plt.show()

#
plt.figure()
plt.plot(f, fftshift(abs(fft(xc))))
plt.xlabel('Frequencies')
plt.xlim([-0.5, 0.5])
plt.title('Fourier transform of the signal with removed drift')


#We check H(f)
plt.figure()
G=fft(g, 1000)
plt.plot(f,abs(fftshift(G)))
plt.xlabel('Normalized fequencies')
plt.title('Transfer Function of the Subtracting Filter')
```

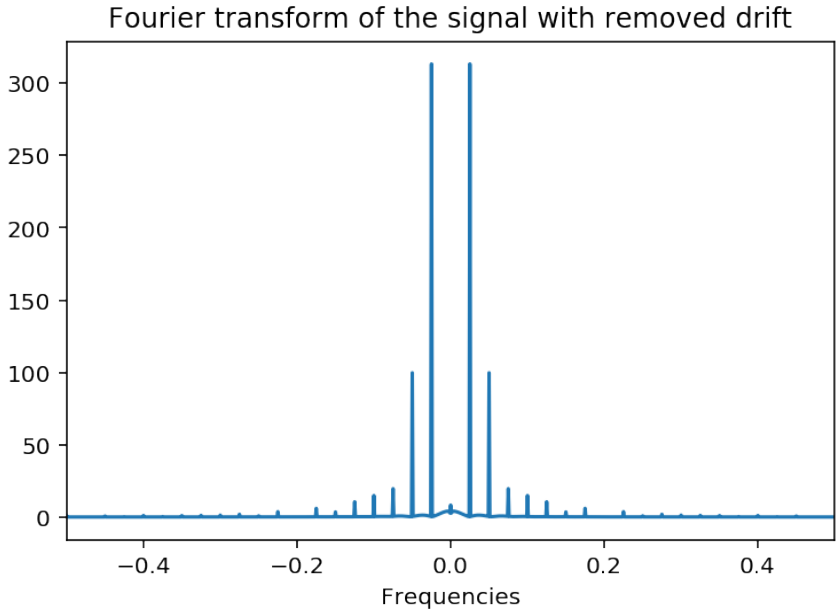
```
/usr/local/lib/python3.5/site-packages/scipy/signal/signaltools.py:1344: FutureWarn
  out = out_full[ind]
```

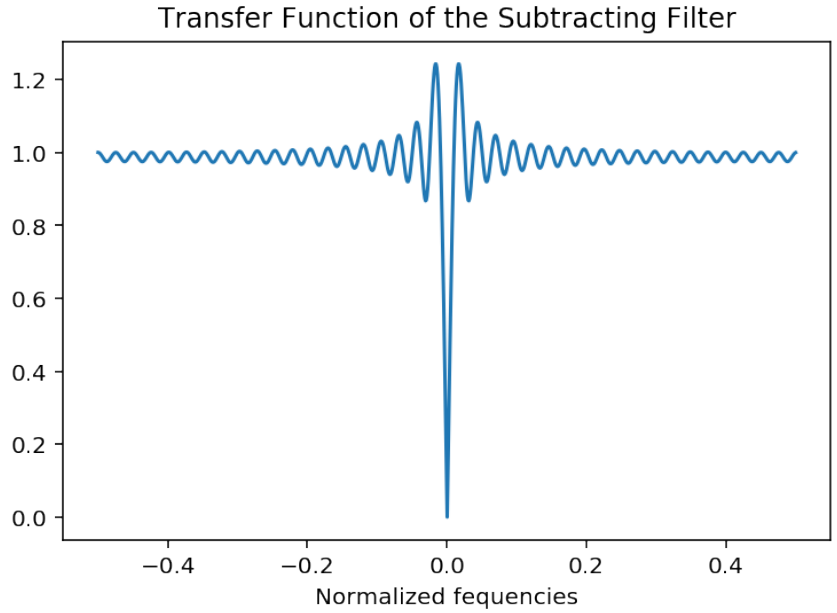
```
Text(0.5, 1.0, 'Transfer Function of the Subtracting Filter')
```



11.4.4 Second part: Boost of a frequency band

We wish now to boost a range of freqencies aound 1000 Hz on the initial signal.


Fourier transform of the signal with removed drift
300
250
200
150
100
50
0
−0.4
−0.2
0.0
0.2
0.4
Frequencies


Transfer Function of the Subtracting Filter
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.4
−0.2
0.0
0.2
0.4
Normalized fequencies

11.5 Theoretical Part

After a (possible) recall of the lecturer on rational filters, compute the poles p_1 and p_2 of a filter in order to perform this accentuation. Compute the transfer function $H(z)$ and the associated impulse response $h(n)$.

Pratical part

- The vector of denominator's $A(z)$ coefficients will be computed according to `A=poly([p1,p2])`, and you will check that you recover the hand-calculated coefficients.
- plt.plot the frequency response
- Compute the impulse response, according to # computing the IR d=zeros(300) d[1]=1 h_accentued=lfilter([1],a,d) (output to a Dirac impulse on 300 point). plot it.
- Compute and plot the impulse response obtained using the theoretical formula. Compare it to the simulation.
- Compute and plot the output of the filter with input x_c, both in the time and frequency domain. Conclude.

```
# ...

# Compute the IR
# ...
#plt.plot(h_accentued)
#plt.title('Impulse response of the boost filter')

# in frequency
# ...
#plt.xlabel('Normalized frequencies')
#plt.xlim([-0.5, 0.5])
#plt.title('Transfer Function of the Boost Filter')
```

```
# Filtering
#sig_accentuated=...
# ...
#plt.xlabel('Time')
#plt.xlim([0, len(x)*Ts])
#plt.title('Signal with boosted 1000 Hz')

# In the frequency domain
# ...
#plt.xlabel('Normalized frequencies')
#plt.xlim([-Fs/2, Fs/2])
#plt.title('Fourier Transform of Boosted Signal')
```

- How can we simultaneously boost around 1000 Hz and remove the drift? Propose a filter that performs the two operations.

```
# both filterings:
# ...

#plt.xlabel('Time')
#plt.xlim([0, len(x)*Ts])
#plt.title('Centered Signal with Boosted 1000 Hz')
```

11.5.1 Lowpass [0- 250 Hz] filtering by the window method

We want now to only keep the low-frequencies components (0 à 250 Hz) of x_c by filtering with a lowpass FIR filter with N=101 coefficients.

Theoretical Part

We consider the ideal lowpass filter whose transfer function $H(f)$ (in modulus) is a rectangular function. Compute the (infinite support) impulse response of the associated digital filter.

Practical Part

a. We want to limit the number of coefficients to L (FIR). We thus have to clip-off the initial impulse response. Compute the vector h with L coefficients corresponding to the initial response, windowed by a rectangular window $\text{rect}_T(t)$, where $T = L * Ts$.
b. plt.plot the frequency response.
c. Comput and plt.plot the output of this filter subject to the input x_c.
d. Observe the group delay of the frequency response: `plt.plot(f,grpdelay(B,A,N))`. Comment.

Theoretical Part

Practical part

```
B=250
Fs=8000
B=B/Fs # Band in normalized fequencies
n=np.arange(-150,150)

def sinc(x):
    x=np.array(x)
    z=[sin(n)/n if n!=0 else 1 for n in x]
    return np.array(z)

# ...
#plt.xlabel('n')
#plt.title('Impulse response')
```

```
# ...

#plt.title("Frequency Response")
#plt.xlim([-1000, 1000])
#plt.grid(True)
```

Output of the lowpass filter

Group Delay

The group delay is computed as indicated here, cf https://ccrma.stanford.edu/~jos/fp/Numerical_Computation_Group_Delay.html

```
def grpdelay(h):
    N=len(h)
    NN=1000
    hn=h*np.arange(N)
    num=fft(hn.flatten(),NN)
```

```
    den=fft(h.flatten(),NN)
    Mden=max(abs(den))
    #den[abs(den)<Mden/100]=1
    Td=real(num/den)
    Td[abs(den)<Mden/10]=0
    return num,den,Td
hh=zeros(200)
#hh[20:25]=array([1, -2, 70, -2, 1])
hh[24]=1
#plt.plot(grpdelay(hh))
num,den,Td=grpdelay(h_tronq)
plt.figure(3)
plt.plot(Td)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-16-2696f2851cc3> in <module>()
     14 hh[24]=1
     15 #plt.plot(grpdelay(hh))
---> 16 num,den,Td=grpdelay(h_tronq)
     17 plt.figure(3)
     18 plt.plot(Td)


NameError: name 'h_tronq' is not defined
```

Thus we see that we have a group delay of ...
END.

12 Random Signals

```
#Some specific imports for plotting

from plot_rea import *
from plot_sighisto import *
%matplotlib inline
```

12.1 Introduction to Random Signals

Just as a random variable is a set of values associated with a probability distribution, a random signal, also callled a random process, is a set of *functions* associated with a probability distribution. In addition to properties similar to those of random variables, the study of random processes include characterizations of dependences *in* the random process (namely notions of *correlation*), the study the behaviour of the function under transformations (*filtering*) and the design of some optimal transformations.

Notations

We denote by $X(n,\omega)$ a random signal X. It is a set of functions of n, the set being indexed by ω. A random signal is thus a bivariate quantity. When $\omega = \omega_i$ is fixed, we get a *realization* of the random process, denoted $X(n,\omega_i)$ or, more simply $x_i(n)$. When n is fixed, the random process reduces to a simple random variable. Considering the process for $n = n_i$, we obtain a random variable $X(n_i,\omega)$, denoted $X_i(\omega)$, or X_i. Finally, we will denote x_i the values taken by the random variable X_i.

12.2 Fundamental properties

12.2.1 Stationnarity

Definition 4. *A random signal is said stationnary if its statistical properties are invariant by time translation.*

This means that the joint probability density function

$$p_{X(n_1),X(n_2),\ldots,X(n_k)} = p_{X(n_1-\tau),X(n_2-\tau),\ldots,X(n_k-\tau)},$$

and if $\tau = n_k$

$$p_{X(n_1),X(n_2),\ldots,X(n_k)} = p_{X(n_1-n_k),X(n_2-n_k),\ldots,X(0)}.$$

Therefore, the joint distribution only depends on $k-1$ parameters, instead of the k initial parameters.

As a consequence, we have that

- $\mathbb{E}[X(n)] = \mu$ is constant and does not depend on the particular time n
- $\mathbb{E}[X(n)X(n-\tau)^*] = R_X(\tau)$ only depends on the delay between the two instants. In such a case, the resulting function $R_X(\tau)$ is called a **correlation function**.

12.2.2 Ergodism

Definition

The time average, taken on realization ω is

$$\langle X(n,\omega)^n \rangle = \lim_{N\to+\infty} \frac{1}{N} \sum_{[N]} X(n,\omega)^n.$$

Of course, in the general case, this time average **is** a random varaible, since it depends on ω.

Definition A random signal is said ergodic if its time averages are deterministic, i.e. non random, variables.

Important consequence

A really important consequence is that if a signal is both stationnary and ergodic, then the statistical means and the time averages are equal.

$$\boxed{\mathbb{E}[\bullet] = \langle \bullet \rangle}$$

Exercise > - Check that

- (moments) Check that if the signal is both stationnary and ergodic, then

$$\mathbb{E}\left[X(n,\omega)^k\right] = \lim_{N\to+\infty} \frac{1}{N} \sum_{[N]} X(n,\omega)^k,$$

- (covariance) Similarly, check that

$$R_X(\tau) = \mathbb{E}[X(n,\omega)X(t-\tau,\omega)] = \lim_{N\to+\infty} \frac{1}{N} \sum_{[N]} X(n,\omega)X(n-\tau,\omega).$$

12.2.3 Examples of random signals

1. Let us first consider the noisy sine wave $X(n,\omega) = A\sin(2\pi f_0 n) + B(n,\omega)$. Function `plot_rea` plots some realizations of this signal and plots the ensemble and time averages. You will also need `sig_histo` which plots the histogram, together with the time series.

```
from plot_rea import *
from plot_sighisto import *
```

Experiment with the parameters (amplitude, number of samples). Is the signal stationary, ergodic, etc?

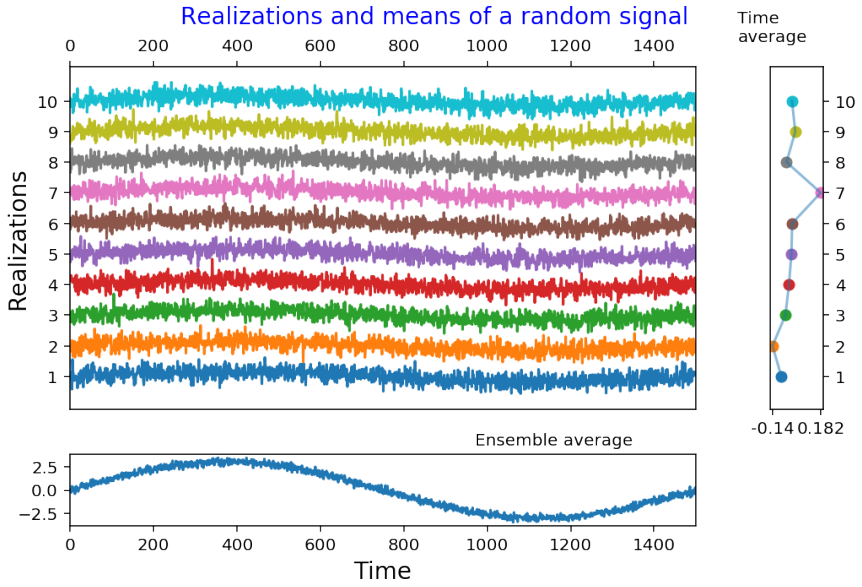
```
import scipy.stats as stats

M = 10
# number of bins in histograms
N = 1500  # Number of samples per realization
K = 200  # Total number of realizations

XGauss = stats.norm(loc=0, scale=1)

#Sine wave plus noise
X = 3 * XGauss.rvs(size=(K, N)) + 3 * np.outer(
    np.ones((K, 1)), np.sin(2 * np.pi * np.arange(N) / N))
print("Standard deviation of time averages: ", np.std(np.mean(X, axis=1)))
#pylab.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['figure.figsize'] = (8, 5)
plot_rea(X, nb=10, fig=1)
```

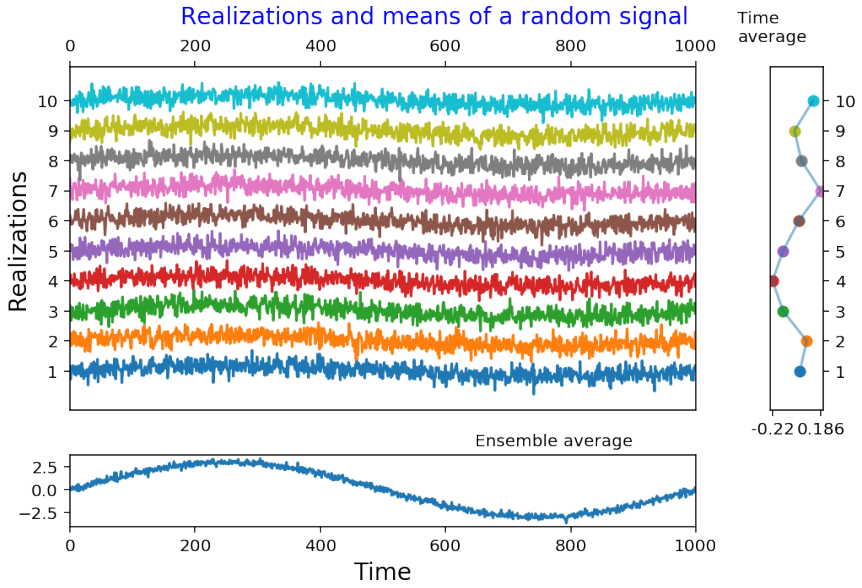
```
Standard deviation of time averages:  0.07627467118754734
```

By varying the number of samples N, we see that the time average converges to zero, for each realization. Thus we could say that this process is ergodic. However, the ensemble average converges to the sine wave and is dependent if time: the process *is not stationary*.

```
XGauss = stats.norm(loc=0, scale=1)
#pylab.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['figure.figsize'] = (8, 5)

def q1_experiment(N):
    K = 200
    #Sine wave plus noise
    X = 3 * XGauss.rvs(size=(K, N)) + 3 * np.outer(
        np.ones((K, 1)), np.sin(2 * np.pi * np.arange(N) / N))
    print("Standard deviation of time averages: ", np.std(np.mean(X, axis=1)
        ))
    plot_rea(X, nb=10, fig=1)


_ = interact(q1_experiment, N=(0, 2000, 10))
```

```
Standard deviation of time averages:  0.09498395926986315
```

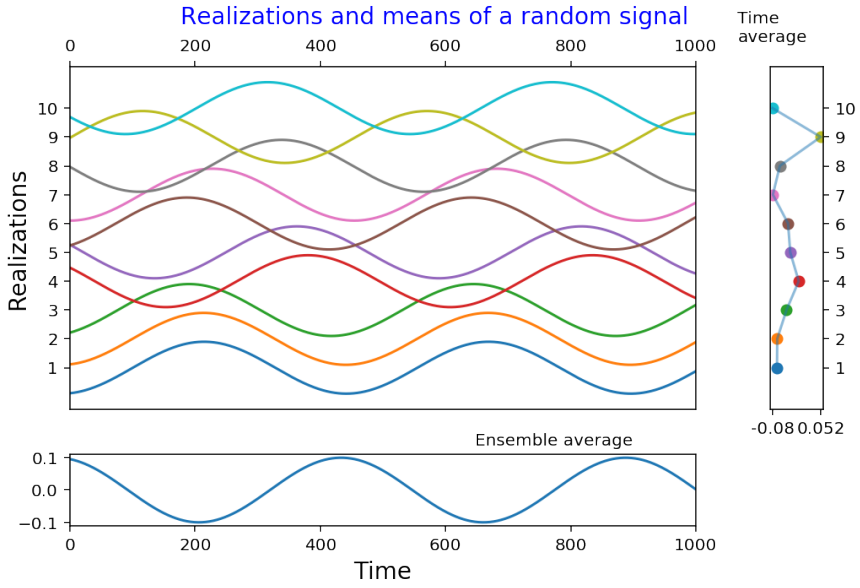

Realizations and means of a random signal
Time average
Realizations
Ensemble average
Time
-0.14 0.182


Realizations and means of a random signal
Time average
Realizations
Ensemble average
Time
-0.22 0.186

2- Consider now a sine wave with a random phase $X(n,\omega) = A\sin(2\pi f_0 n + \phi(\omega))$.

Experiment with the parameters (amplitude, number of samples). Is the signal stationary, ergodic, etc? Also change the value of the frequency, and replace function `sin` by `square` which generates a pulse train instead of a sine wave.

```
from pylab import *
K = 100
N = 1000
fo = 2.2 / N
S = zeros((K, N))
for r in range(K):
    S[r, :] = 1.1 * sin(2 * pi * fo * arange(N) + 2 * pi * rand(1, 1))
plot_rea(S, fig=2)
```



This example shows that a random signal is not necessarily noisy and irregular. Here we have a random signal which is 'smooth'. The random character is introduced by the random phase, which simply reflects that we do not know the time origin of this sine wave.

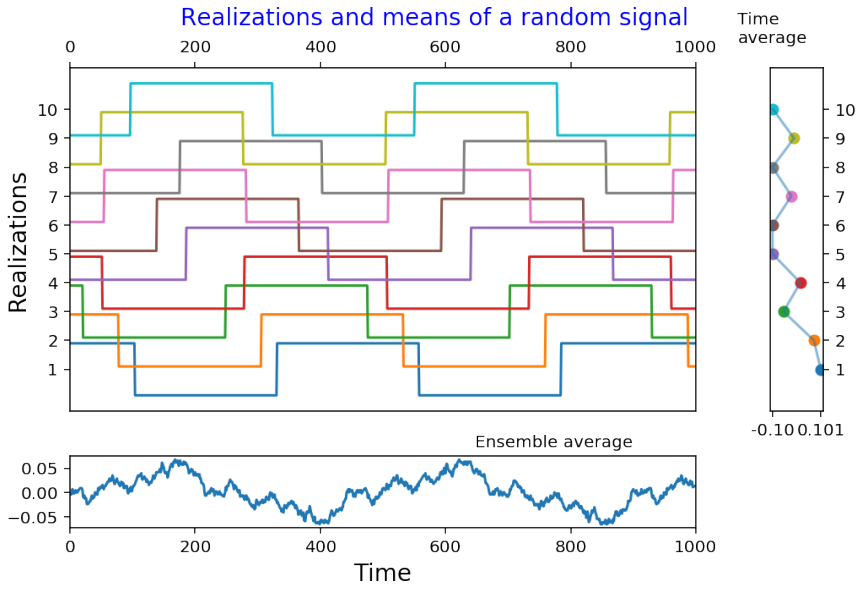
Here, we see that both the time average and the ensemble average converge to zero. Therefore, we can conclude that this signal is stationary and ergodic.

Let us now define a square wave:

```
def square(x):
    """square(x): \n
    Returns a pulse train with period :math:'2\pi'
    """
    return sign(sin(x))
```

Then generate a random square wave as follows

```
K = 1000
N = 1000
S = zeros((K, N))
for r in range(K):
    S[r, :] = 1.1 * square(2 * pi * fo * arange(N) + 2 * pi * rand(1, 1))
plot_rea(S, fig=2)
```



Again, we see that both means tend to zero, a constant, which means that the signal is stationary (its ensemble average does not depend of time) and ergodic (its time average does not depend on the actual realization).

12.2.4 White noise

For discrete signals, a white noise is simply a sequence of independent variables (often the variables will be also identically distributed). An independent and identically distributed signal is denoted iid. Since the components of a white noise are all independent, there will be no correlation between them. We will see later that the spectral representation of a white noise is *flat*, thus coining the name of white noise by analogy with the white light.

The notion of white noise is more involved in the case of a time-continuous signal. The white noise is in such case a limit processe with "microscopic dependences".

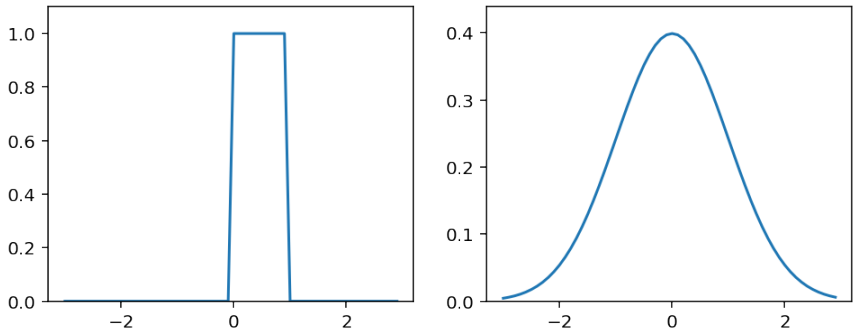
We consider now two kinds of random noises: the first one is a sequence of independent and identically distributed variables (iid variables), according to a uniform distribution. The second one is an iid sequence, Gaussian distributed. Plot the two probability density functions, plot the histograms (with `sig_histo`) and compare the time series.

> 3- Compute and analyze the histograms of two white noises, respectively with a uniform and a Gaussian probability density function, using the lines in script `q1c`. Do this for several realizations (launch the program again and again) and change the number of points and of bins. Compare the two signals. What do you think of the relation between whiteness and gaussianity.

```
# An object "uniform random variable" with fixed bounds [0,1]
x_uni = stats.uniform(loc=0, scale=1)
# An object "gaussian random variable" with zero mean and scale 1
x_gauss = stats.norm(loc=0, scale=1)
#plt.rcParams['figure.figsize'] = (8,5)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
x = arange(-3, 3, 0.1)
```

```
ax1.plot(x, x_uni.pdf(x))
ax1.set_ylim([0, 1.1 * max(x_uni.pdf(x))])
ax2.plot(x, x_gauss.pdf(x))
ax2.set_ylim([0, 1.1 * max(x_gauss.pdf(x))])
```

```
(0, 0.438836508441576)
```




```
(m, v) = x_uni.stats(moments='mv')
print("Uniform distribution: ",
      "Value of the mean : {0:2.3f} and of the variance {1:2.3f}".format(
          float(m), float(v)))
(m, v) = x_gauss.stats(moments='mv')
print("Gauss distribution: ",
      "Value of the mean : {0:2.3f} and of the variance {1:2.3f}".format(
          float(m), float(v)))
```

```
Uniform distribution:  Value of the mean : 0.500 and of the variance 0.083
Gauss distribution:  Value of the mean : 0.000 and of the variance 1.000
```

We can compare the two signals

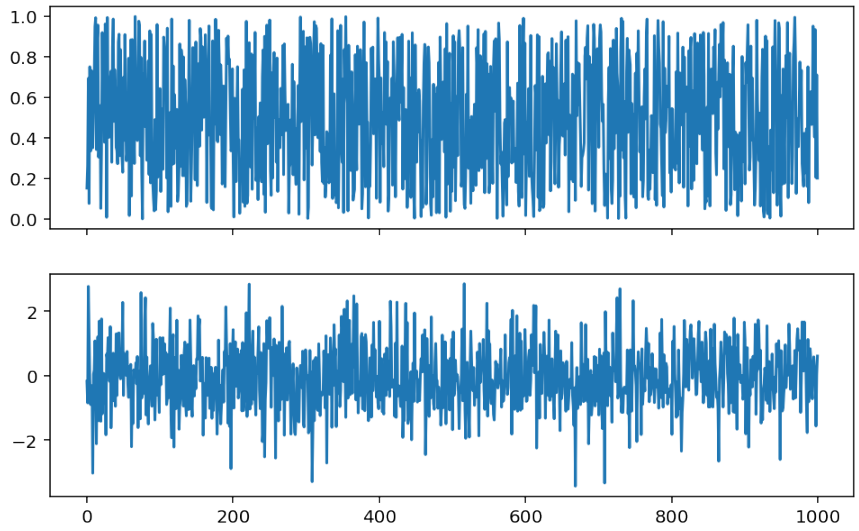
```
fig, (ax1, ax2) = subplots(2, 1, sharex=True)
ax1.plot(x_uni.rvs(size=N))
ax2.plot(x_gauss.rvs(size=N))
```

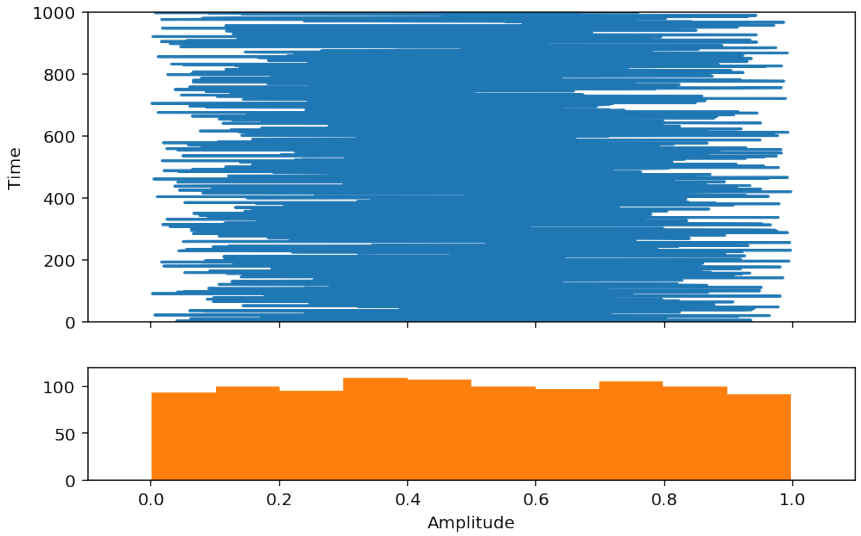
```
[<matplotlib.lines.Line2D at 0x7f0a836bc518>]
```

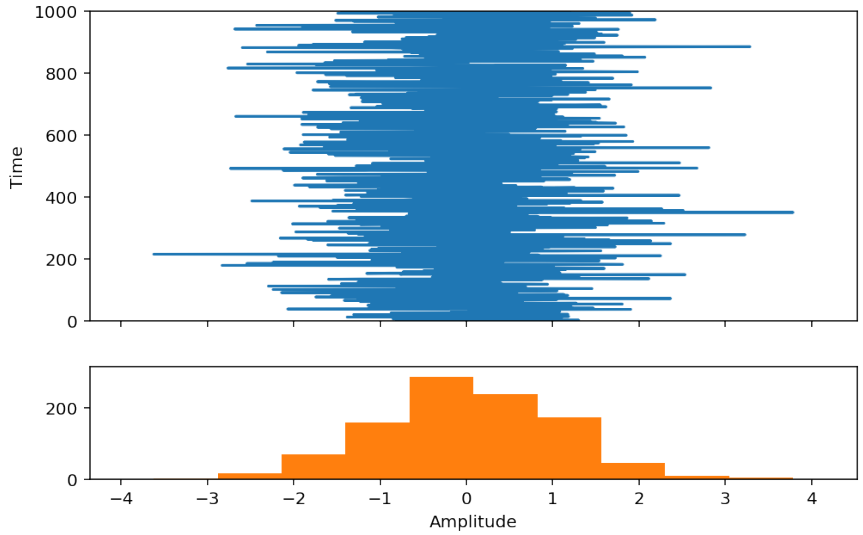
```
from plot_sighisto import *
plot_sighisto(x_uni.rvs(size=N), fig=1)
plot_sighisto(x_gauss.rvs(size=N), fig=2)
```

We see that the Gaussian noise is more concentrated on its mean 0, and exhibits more important values, while the uniform noise is confined into the interval [0,1].

Concerning the question on the relation between whiteness and Gaussianity, actually, there is no relation between these two concepts. A white noise can be distributed according to any distribution, and a Gaussian sequence is not necessarily iid (white).


1.0
0.8
0.6
0.4
0.2
0.0
2
0
−2
0
200
400
600
800
1000


1000
800
600
400
200
0
Time
100
50
0
0.0
0.2
0.4
0.6
0.8
1.0
Amplitude



12.3 Second order analysis

12.3.1 Correlation functions

Definition

If $X(n,\omega)$ and $Y(n,\omega)$ sare two jointly stationnary random processes, the intercorrelation and autocorrelation functions are defined by

$$R_{XY}(k) \stackrel{\triangle}{=} \mathbb{E}\left[X(n,\omega)Y^*(n-k,\omega)\right] = \lim_{\text{erg}\, N\to+\infty} \frac{1}{N}\sum_{n=0}^{N} X(n,\omega)Y^*(n-k,\omega),$$
$$R_{XX}(k) \stackrel{\triangle}{=} \mathbb{E}\left[X(n,\omega)X^*(n-k,\omega)\right] = \lim_{\text{erg}\, N\to+\infty} \frac{1}{N}\sum_{n=0}^{N} X(n,\omega)X^*(n-k,\omega).$$

Main properties

1. (Hermitian symmetry)

$$R_{YX}(\tau) = \mathbb{E}\left[Y(n,\omega)X^\star(n-\tau,\omega)\right] = \mathbb{E}\left[Y(n+\tau,\omega)X^\star(n,\omega)\right] = \mathbb{E}\left[X(n,\omega)Y^\star(n+\tau,\omega)\right]^\star = R^\star_{XY}(-\tau).$$

2. (Symmetry for the autocorrelation). In the case of the autocorrelation function, the hermitian symmetry reduces to
$$R_{XX}(\tau) = R^\star_{XX}(-\tau).$$

3. (Centering). If $X_c(n,\omega) = X(n,\omega) - m_X$ is the centered signal, then
$$R_{XX}(\tau) = R_{X_cX_c}(\tau) + m_X^2.$$

4. (Autocorrelation and power). For a delay $\tau = 0$, we have
$$R_{XX}(0) = \mathbb{E}\left[|X(n,\omega)|^2\right] = \lim_{\text{erg}\, N\to+\infty} \frac{1}{N}\sum_{[N]} |X(n,\omega)|^2 = P_X.$$

This shows that $R_{XX}(0)$ is nothing but the power of the signal under study. Observe that necessarily $R_{XX}(0) > 0$.

5. (Maximum). Beginning with the *Schwarz inequality*,

$$| < x,y > |^2 \leq < x,x >< y,y >,$$

and using the scalar product $ <x_1,x_2> = E[X_1(n)X_2^*(n)] $, *weget*

6. $|R_{YX}(\tau)|^2 \leq R_{XX}(0)R_{YY}(0), \quad \forall \tau,$

7. $|R_{XX}(\tau)| \leq R_{XX}(0), \quad \forall \tau,$

(Non negativity) The autocorrelation function is non negative definite

$$\sum_i \sum_j \lambda_i R_{XX}(\tau_i - \tau_j)\lambda_j \geq 0, \quad \forall i,j.$$

proof: develop $\mathbb{E}\left[|\sum_i \lambda_i X(\tau_i)|^2\right] \geq 0$

(Correlation coefficient). By the maximum property, the correlation coefficient

$$\rho_{XY}(\tau) = \frac{R_{YX}(\tau)}{\sqrt{R_{XX}(0)R_{YY}(0)}}$$

is such that $\rho_{XY}(\tau) \leq 1$.

(Memory). If the correlation coefficient is zero after a certain time t_c then the process is said to have a finite memory and t_c is called the *correlation time*.

Exercises

1. Developing $\mathbb{E}\left[|X + \lambda Y|^2\right]$ into a polynom of λ and observing that this polynom in always nonegative, prove the Schwarz inequality.

2. Consider a random signal $U(n,\omega)$ defined on the interval $[0,N]$. Define the periodic signal

$$X(n,\omega) = \text{Rep}_N[U(n,\omega)] = \sum_k U(t - kN, \omega).$$

   - Show that $R_{UU}(\tau) = 0$ for $\tau \notin [-N,N]$.
   - Show that $R_{XX}(\tau)$ is a periodic function with period N and express $R_{XX}(\tau)$ as a function of $R_{UU}(\tau)$.

3. Consider a random signal $X(n,\omega)$ with autocorrelation $R_{XX}(k)$ and define

$$Z(n,\omega) = X(n,\omega) + aX(n - n_0, \omega).$$

   Compute the autocorrelation function of $Z(n,\omega)$.

Estimation of correlation functions

By stationnarity and ergodism, we have that

$$R_{XX}(k) = \lim_{\text{erg}\, N\to+\infty} \frac{1}{N}\sum_{n=0}^{N} X(n,\omega)X^*(n-k,\omega).$$

Given a finite number of points N, with data known from $n = 0..N-1$, it is thus possible to approximate the correlation function by a formula like

$$R_{XX}(k) = \frac{1}{N}\sum_{n=0}^{N-1} X(n,\omega)X^*(n-k,\omega).$$

If we take $k \geq 0$, we see that $X^*(n-k,\omega)$ is unavailable for $k > n$. Consquently, the sum must go from $n = k$ to $N-1$. At this point, people define two possible estimators. The first one is said "unbiased" while the second is "biased" (check this by computing the expectation $\mathbb{E}[\bullet]$ of the two estimators).

$$\hat{R}_{XX}^{(unbiased)}(k) = \frac{1}{N-k}\sum_{n=k}^{N-1} X(n,\omega)X^*(n-k,\omega) \tag{12.1}$$

$$\hat{R}_{XX}^{(biased)}(k) = \frac{1}{N}\sum_{n=k}^{N-1} X(n,\omega)X^*(n-k,\omega). \tag{12.2}$$

For the biased estimator, it can be shown (Bartlett) that the variance has the form

$$Var\left[\hat{R}_{XX}^{(biased)}(k)\right] \approx \frac{1}{N}\sum_{m=-\infty}^{+\infty} \rho(m)^2 + \rho(m+k)\rho(m-k) - 4\rho(m)\rho(k)\rho(m-k) + 2\rho(m)^2\rho(k)^2,$$

that is, essentially a constant over N. As far the unbiased estimator is concerned, we will have a factor $N/(N-k)$, and we see that this time the variance *increases* with k. Thus, though it is unbiased, this estimator has a very bad behaviour with respect to the variance.

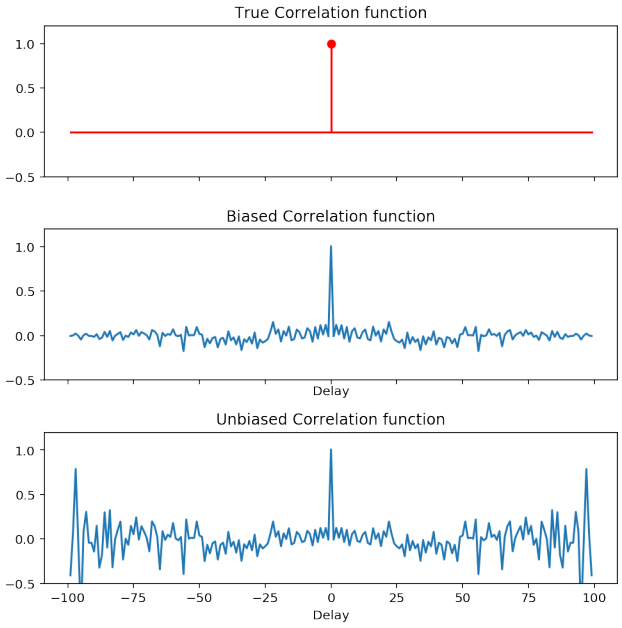
This is checked below. First we generate a gaussian white noise, compute the two estimates of the correlation function and compare them.

```
from correlation import xcorr
from scipy import stats as stats
N = 100
XGauss = stats.norm(loc=0, scale=1)
S = XGauss.rvs(size=N)
#
Rbiased, lags = xcorr(S, norm='biased')
Runbiased, lags = xcorr(S, norm='unbiased')
Rtheo = zeros(size(Rbiased))
Rtheo[lags == 0] = 1
```

```
Rt = ones(1)
fig, ax = subplots(3, 1, figsize=(7, 7), sharex=True, sharey=True)
# biased correlation
ax[1].plot(lags, Rbiased)
#ax[0].axvline(0,ymin=0,ymax=1,color='r',lw=3)
ax[1].set_title("Biased Correlation function")
ax[1].set_xlabel("Delay")
ax[1].axis('tight')  #Tight layout of the axis
# unbiased correlation
ax[2].plot(lags, Runbiased)
```

```
ax[2].set_title("Unbiased Correlation function")
ax[2].set_xlabel("Delay")
# theoretical correlation
ax[0].stem([0], [1], linefmt='r-', markerfmt='ro', basefmt='r-')
ax[0].plot([lags[0], lags[-1]], [0, 0], 'r')
ax[0].set_title("True Correlation function")
fig.tight_layout()
ax[1].axis('tight')
ax[0].set_ylim([-0.5, 1.2])
```

```
(-0.5, 1.2)
```



Detecting hidden periodicities

We consider here a time series composed of a periodic signal corrupted by a white noise. The signal is completely hidden by the noise. We show here that it is possible to find some information in the autocorrelation function.

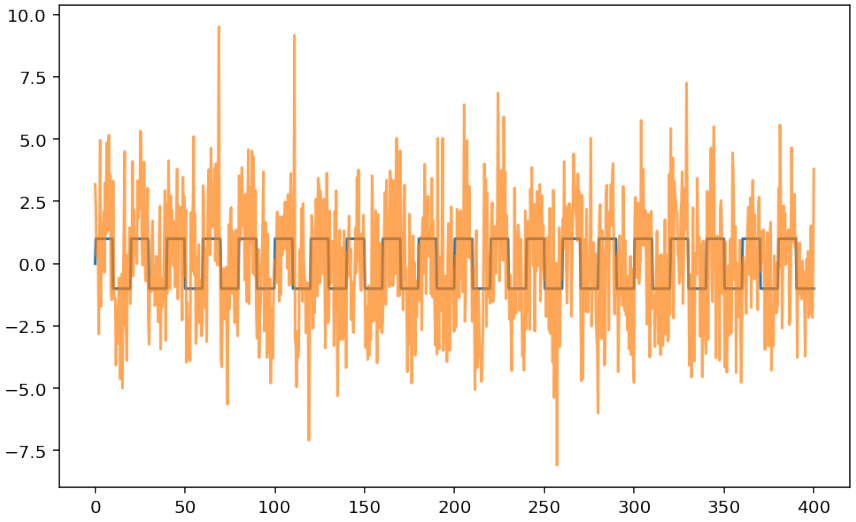
Exercises - (a) Check that the correlation of a periodic signal is periodic - (b) Give the correlation of $y = s + w$ if s and w are independent.

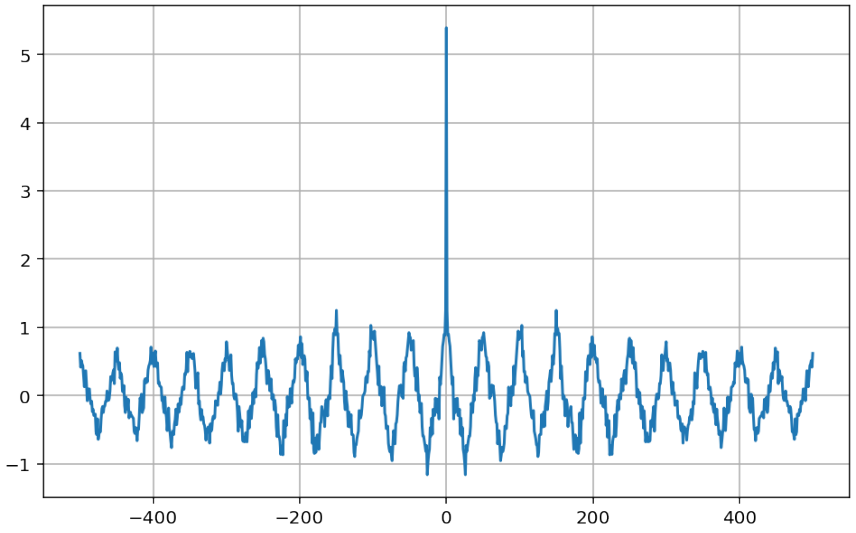
```
def square(x):
    """square(x): \n
    Returns a pulse train with period :math:`2\pi`
    """
    return sign(sin(x))


N = 1000
f0 = 0.05
t = np.linspace(0, 400, N)
```

```
x = 1 * square(2 * pi * f0 * t)
noise = stats.norm(loc=0, scale=2).rvs(N)
observation = x + noise
#
```

Plot the correlation of the noisy signal. Are you able to retrieve the unknown periodicities? Experiment with the parameters. Conclusion.

```
plt.plot(t, x, '-')
plt.plot(t, observation, alpha=0.7)
#
Rbiased, lags = xcorr(observation, norm='biased', maxlags=500)
plt.figure()
plt.plot(lags, Rbiased)
plt.grid(b=True)
```





The last figure shows the correlation of the noisy periodic signal. This correlation is simply the superposition of the correlation of the noise and of the correlation of the signal (Check it!)

$$R_{\text{obs,obs}} = R_{\text{sig,sig}} + R_{\text{noise,noise}}$$

Since the correlation of the noise (a Dirac impulse) is concentrated at zero, we can read - the period of the signal: 50 (that is a relative frequency of 50/1000=0.05) - the power of the signal: 0.5 - the power of the noise: 4.5 - 0.5 = 4 (was generated with a standard deviation of 2). The correlation function then enable us to grasp many informations that were not apparent in the time series!

```
from scipy.signal import lfilter
from scipy.fftpack import fft, ifft
%matplotlib inline
```

12.4 Filtering

12.4.1 General relations for cross-correlations

We consider a situation where we want to study the correlations between the different inputs and outputs of a pair of two filters:

$$\left\{ \begin{array}{rcl} Y_1(n,\omega) & = & (X_1 * h_1)(n,\omega), \\ Y_2(n,\omega) & = & (X_2 * h_2)(n,\omega), \end{array} \right.$$

Let us compute the intercorrelation between $Y_1(n)$ and $Y_2(n)$:

$$R_{Y_1Y_2}(m) = \mathbb{E}\left[Y_1(n,\omega)Y_2^*(n-m,\omega))\right] = \mathbb{E}\left[(X_1 * h_1)(n,\omega))(X_2^* * h_2^*)(n-m,\omega))\right].$$

The two convolution products are

$$\begin{array}{rcl} (X_1 * h_1)(n,\omega)) & = & \sum_u X_1(u,\omega))h_1(n-u), \\ (X_2 * h_2)(n-m,\omega)) & = & \sum_v X_2(v,\omega))h_2(n-m-v), \end{array}$$

and

$$\begin{array}{rcl} R_{Y_1Y_2}(m) & = & \mathbb{E}\left[\sum_u X_1(n-u,\omega))h_1(u)\sum_v X_2^*(n-m-v,\omega))h_2^*(v)\right] \\ & = & \mathbb{E}\left[\sum_u\sum_v X_1(n-u)h_1(u)X_2^*(n-m-v)h_2^*(v)\right] \\ & = & \sum_u\sum_v h_1(u)R_{X_1X_2}(m+v-u)h_2^*(v). \end{array}$$

Looking at the sum over u, we recognize a convolution product between h_1 and $R_{X_1X_2}$, expressed at time $(m+v)$:

$$\begin{aligned} R_{Y_1Y_2}(m) &= \sum_v (h_1 * R_{X_1X_2})(m+v) h_2^*(v) \\ &= \sum_v (h_1 * R_{X_1X_2})(m+v) h_2^{(-)*}(-v), \end{aligned}$$

where we have noted $h_2^{(-)}(v) = h_2(-v)$. In this last relation, we recognize anoter convolution product, this time between $(h_1 * R_{X_1X_2})$ and $h_2^{*(-)}$:

$$\begin{aligned} R_{Y_1Y_2}(m) &= \sum_v (h_1 * R_{X_1X_2})(m+v) h_2^{*(-)}(-v) \\ &= \sum_{v'} (h_1 * R_{X_1X_2})(m-v') h_2^{*(-)}(v') \\ &= \left(h_1 * R_{X_1X_2} * h_2^{*(-)}\right)(m). \end{aligned}$$

We finally obtain the important formula:

$$\boxed{R_{Y_1Y_2}(m) = \left(h_1 * R_{X_1X_2} * h_2^{*(-)}\right)(m)}.$$

12.4.2 By-products

- **[Autocorrelation of the output of a filter]** With a single filter we can apply the previous formula, with
$$\begin{cases} X_1 = X_2 = X, \\ h_1 = h_2 = h. \end{cases}$$
Of course $Y_1 = Y_2 = Y$, and
$$\boxed{R_{YY}(m) = \left(h * R_{XX} * h^{*(-)}\right)(m)}.$$

- **[Cross correlation between output and input]** We want to measure the correlation between the input and output of a filter. Toward this goal, we consider
$$\begin{cases} X_1 = X_2 = X, \\ Y_1 = Y, \\ Y_2 = X, h_1 = h, \\ h_2 = h. \end{cases}$$
In this case, we got
$$\boxed{R_{YX}(m) = (h * R_{XX})(m)}.$$
The cross correlation between the output and the input of filter has the very same apparence as the filtering which links the output to the input.

12.4.3 Examples

We study now the filtering of random signals. We begin with the classical impulse response $h(n) = a^n$, with $x(n)$ a uniform distributed white noise at the input, and we denote $y(n)$ the output.

1. Filter the signal $x(n)$, with the help of the function `lfilter`. Compare the input and output signals, and in particular their variations. Compare the histograms. Look at the Fourier transform of the output. Do this for several values of a, beginning with $a = 0.9$.

2. Using the function `xcorr` (import it via `from correlation import xcorr`), compute all the possible correlations between the input and the output. What would be the correlation matrix associated with the signal $x(n) = [x(n)\ y(n)]^t$? Compare the impulse response h to the cross-correlation $Ryx(k)$. Explain. Experiment by varying the number of samples N and a (including its sign).

3. Consider the identification of the impulse response by cross-correlation, as above, but in the noisy case. Add a Gaussian noise to the output and compute the cross-correlation. Observe, comment and experiment with the parameters.

The filtering is done thanks to the function `lfilter`. We have first to import it, *eg* as

```
from scipy.signal import lfilter
```

We will also need to play with ffts so it is a good time to import it from fftpack

```
from scipy.fft import fft, ifft
```

```
N = 1000  #Number of samples
x = stats.uniform(-0.5, 1).rvs(N)
a = 0.9
# Filtering and plots...
# FILL IN...
```

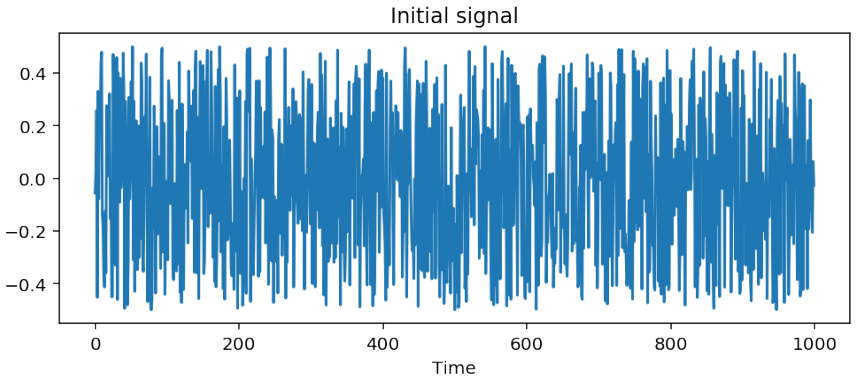
```
y = lfilter([1], [1, -a], x)
figure(figsize=(8, 3))
plot(x)
xlabel("Time")
title("Initial signal")
figure(figsize=(8, 3))
plot(y)
xlabel("Time")
title("Filtered signal")
```

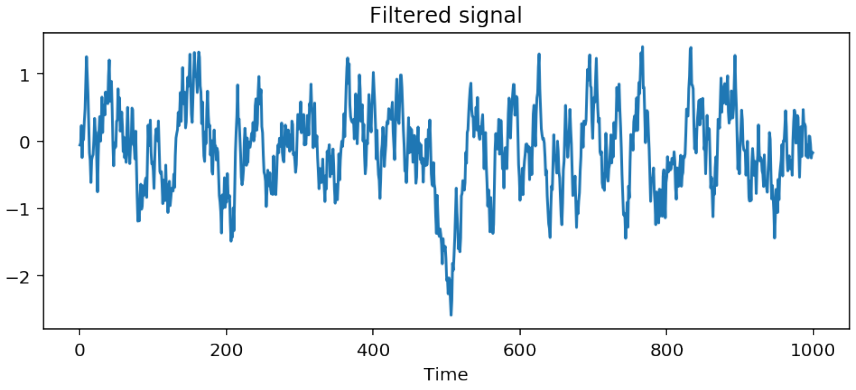
```
Text(0.5, 1.0, 'Filtered signal')
```

We see that the output has slower variations than the input. This is the result of the filtering operation. Let us now look at the histograms:

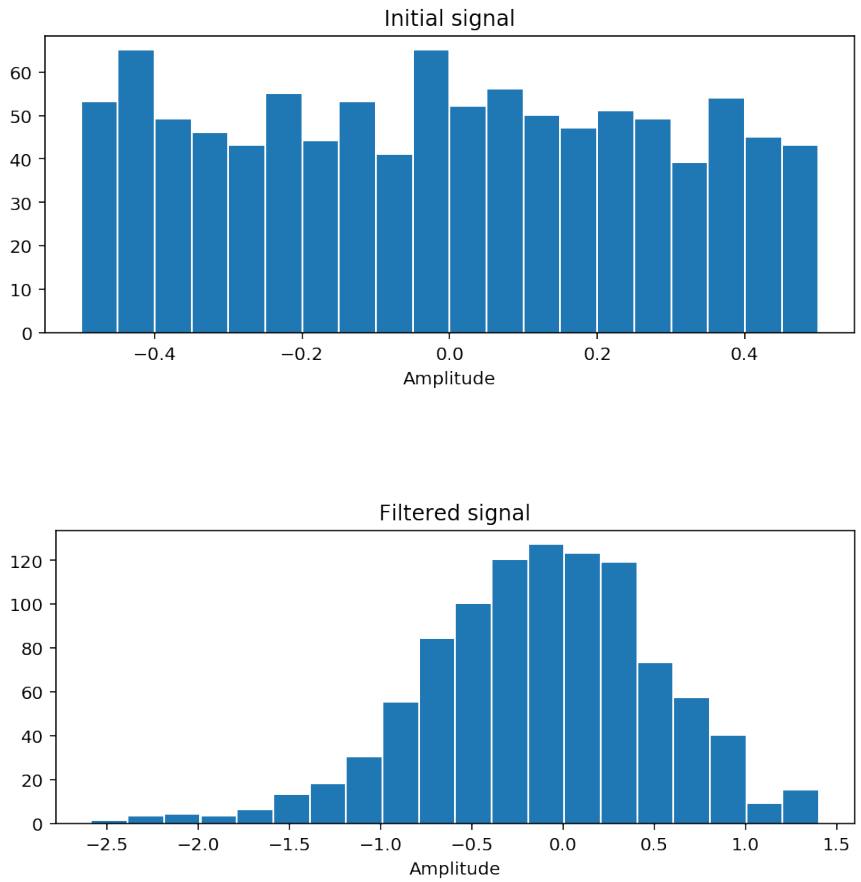
```
#Histograms
# FILL IN
```

```
# Histograms
figure(figsize=(8, 3))
plt.hist(x, bins=20, rwidth=0.95)
plt.xlabel("Amplitude")
plt.title("Initial signal")
figure(figsize=(8, 3))
plt.hist(y, bins=20, rwidth=0.95)
plt.xlabel("Amplitude")
plt.title("Filtered signal")
```


Initial signal
0.4
0.2
0.0
−0.2
−0.4
0
200
400
600
800
1000
Time


Filtered signal
1
0
−1
−2
0
200
400
600
800
1000
Time

```
Text(0.5, 1.0, 'Filtered signal')
```



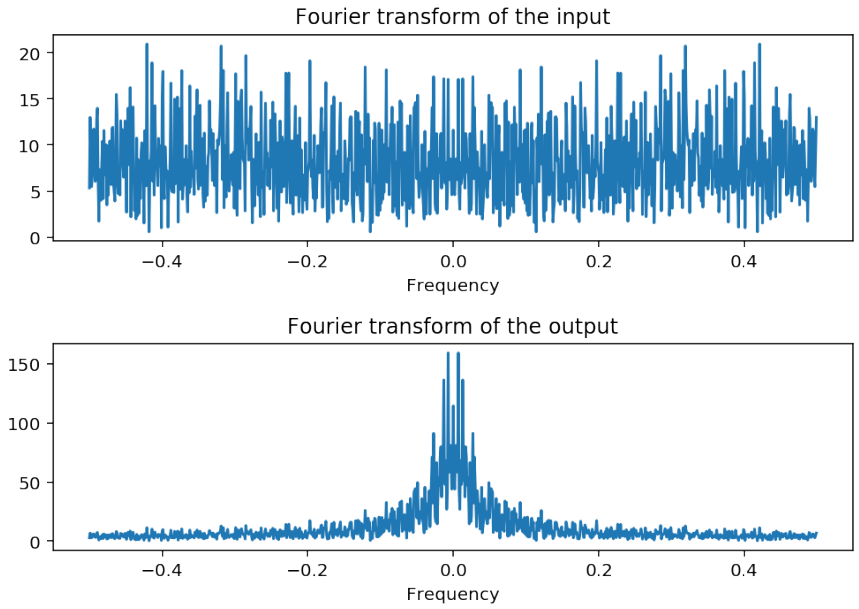
While the initial signal is uniformly distributed, the histogram of the output looks like the histogram of a gaussian. Actually, this is related to the central limit theorem: the mixture of iid variables tends to a gaussian. This also explains th emodification of the amplitudes observed on the time signal.

Let us finally look at the Fourier transform:

```
#FILL IN
```

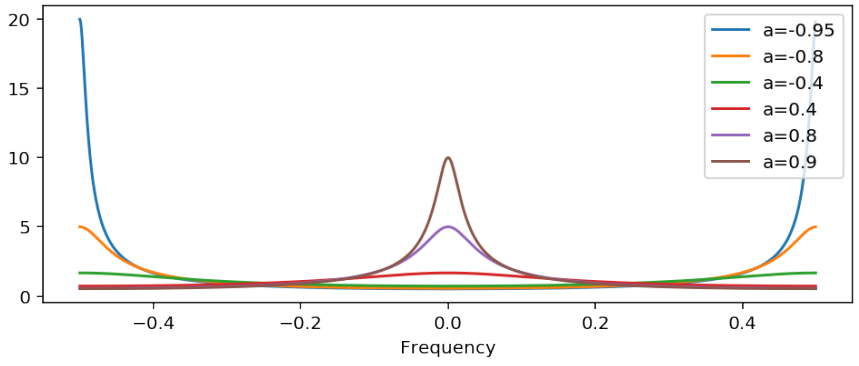
```
f = arange(N) / N - 0.5
fig, ax = subplots(2, 1, figsize=(7, 5))
ax[0].plot(f, abs(fftshift(fft(x))))
ax[0].set_title("Fourier transform of the input")
ax[0].set_xlabel("Frequency")
ax[0].axis('tight')  #Tight layout of the axis
ax[1].plot(f, abs(fftshift(fft(y))))
ax[1].set_title("Fourier transform of the output")
ax[1].set_xlabel("Frequency")
fig.tight_layout()
ax[1].axis('tight')
```

```
(-0.54995, 0.54895, -7.625895183114649, 167.46236056577)
```



Recall how the transfer function changes with a:

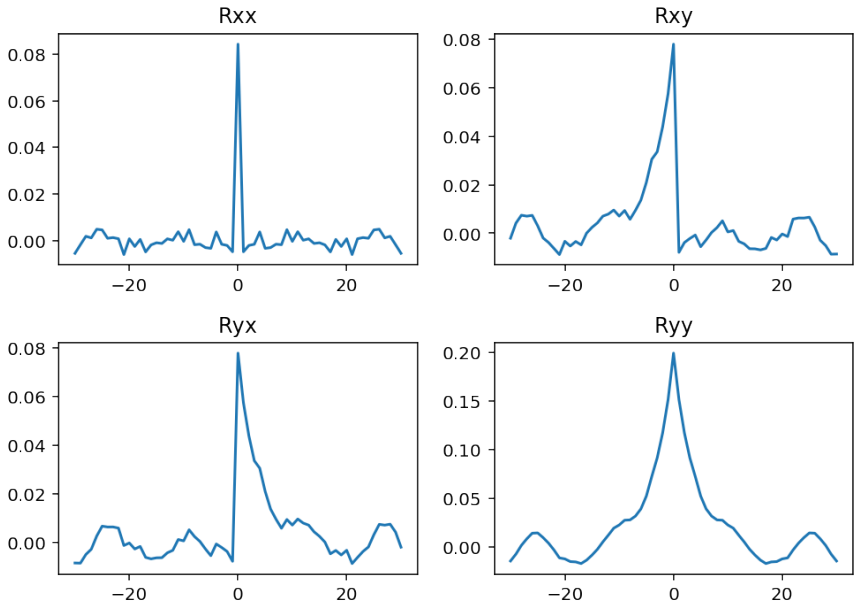
```
figure(figsize=(8, 3))
d = zeros(N)
d[0] = 1
for a in (−0.95, −0.8, −0.4, 0.4, 0.8, 0.9):
    h = lfilter([1], [1, −a], d)
    H = fft(h)
    plot(f, abs(fftshift(H)), label='a={}'.format(a))

legend()
axis('tight')
_ = xlabel("Frequency")
```



12.4.4 Correlation matrix

```
from correlation import xcorr
N = 1000  #Number of samples
```

```
x = stats.uniform(-0.5, 1).rvs(N)
a = 0.8
y = lfilter([1], [1, -a], x)

L = 30
Rxx, lags = xcorr(x, x, maxlags=L)
Rxy, lags = xcorr(x, y, maxlags=L)
Ryx, lags = xcorr(y, x, maxlags=L)
Ryy, lags = xcorr(y, y, maxlags=L)
fig, ax = subplots(2, 2, figsize=(7, 5))
axf = ax.flatten()
Rtitles = ('Rxx', 'Rxy', 'Ryx', 'Ryy')
for k, z in enumerate((Rxx, Rxy, Ryx, Ryy)):
    axf[k].plot(lags, z)
    axf[k].set_title(Rtitles[k])
fig.tight_layout()
```



We have represented above all the possible correlations between the input and the ouput. This representation corresponds to the correlation matrix of the vector $z(n) = [x(n)\ \ y(n)]^H$ that would give

$$\mathbb{E}\left\{ \begin{bmatrix} x(n) \\ y(n) \end{bmatrix} \begin{bmatrix} x(n-k)^* & y(n-k)^* \end{bmatrix} \right\} = \begin{bmatrix} R_{xx}(k) & R_{xy}(k) \\ R_{yx}(k) & R_{yy}(k) \end{bmatrix}$$

12.4.5 Identification of a filter by cross-correlation

We know that the cross-correlation of the output of a system with IR h with its input is given by

$$R_{yx}(k) = (R_{xx} * h)(k).$$

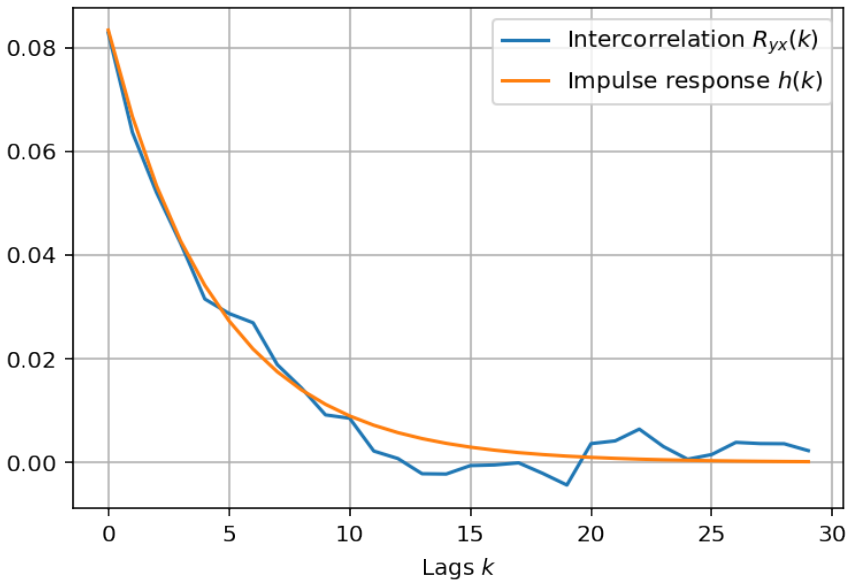
When the input is a white noise, then its autocorrelation $R_{xx}(k)$ is a Dirac impulse with weight σ_x^2, $R_{xx}(k) = \sigma_x^2 \delta(k)$, and R_{yx} is proportional to the impulse response:

$$R_{yx}(k) = \sigma_x^2 h(k).$$

This is what we observe here:

```
# An object "uniform random variable" with fixed bounds [0,1]
from correlation import xcorr
x_uni = stats.uniform(loc=0, scale=1)
(m, v) = x_uni.stats(moments='mv')
print("Uniform distribution: ",
      "Value of the mean : {0:2.3f} and of the variance {1:2.3f}".format(
          float(m), float(v)))
N = 1000  #Number of samples
x = stats.uniform(-0.5, 1).rvs(N)  #generates N values for x
a = 0.8
y = lfilter([1], [1, -a], x)  #Computes the output of the system
L = 30
Ryx, lags = xcorr(y, x, maxlags=L)  #then the cross-correlation
d = zeros(N)
d[0] = 1
h = lfilter([1], [1, -a], d)  #and the impulse response
plot(arange(L), Ryx[arange(L, L + L)], label="Intercorrelation $R_{yx}(k)$")
plot(arange(L), v * h[arange(L)], label="Impulse response $h(k)$")
xlabel("Lags $k$")
grid(True)
legend()
```

```
Uniform distribution:  Value of the mean : 0.500 and of the variance 0.083
```



In the noisy case, the same kind of observations hold. Indeed, if z is a corrupted version of y, with $z(n) = y(n) + w(n)$, then

$$R_{zx}(k) = R_{yx}(k) + R_{wx}(k) = R_{yx}(k)$$

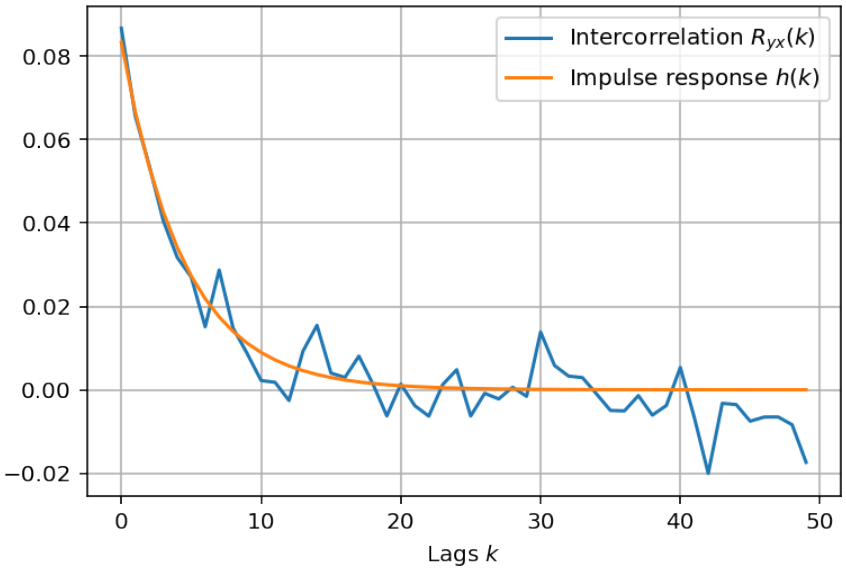
provided that x and w are uncorrelated, which is reasonable assumption.

```
N = 1000
#Remember that the variance of $x$ is given by
x_uni = stats.uniform(-0.5, 1)
```

```
(m, v) = x_uni.stats(moments='mv')
print("Uniform distribution: ",
      "Value of the mean : {0:2.3f} and of the variance {1:2.3f}".format(
          float(m), float(v)))
```

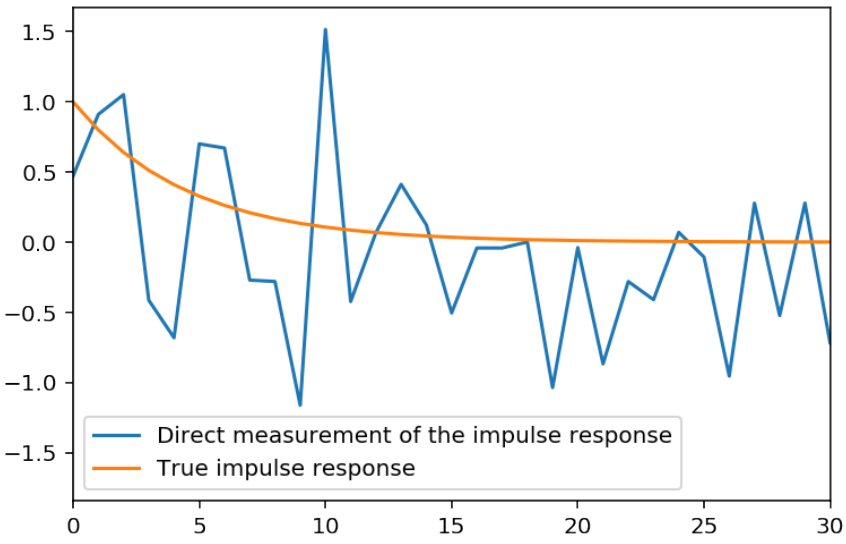
```
Uniform distribution:  Value of the mean : 0.000 and of the variance 0.083
```

```
N = 1000  #Number of samples
x = stats.uniform(-0.5, 1).rvs(N)  #generates N values for x
a = 0.8
y = lfilter([1], [1, -a], x)  #Computes the output of the system
w = stats.norm(0, 1).rvs(N)  #Gaussian noise
y = y + 0.5 * w
L = 50
Ryx, lags = xcorr(y, x, maxlags=L)  #then the cross-correlation
d = zeros(N)
d[0] = 1
h = lfilter([1], [1, -a], d)  #and the impulse response
plot(arange(L), Ryx[arange(L, L + L)], label="Intercorrelation $R_{yx}(k)$")
plot(arange(L), v * h[arange(L)], label="Impulse response $h(k)$")
xlabel("Lags $k$")
grid(True)
legend()
```



while the direct measure of the impulse response would give

```
plot(
    arange(N), h + 0.5 * w, label="Direct measurement of the impulse
        response")
plot(arange(N), h, label="True impulse response")
xlim([0, 30])
_ = legend()
```



Hence, we see that identification of a system is possible by cross-correlation, even when dealing with noisy outputs. Such identification would be impossible by direct measurement of the IR, because of the presence of noise.

12.5 Analyse dans le domaine fréquentiel

En repartant de la formule des interférences

$$R_{Y_1Y_2}(m) = \left(h_1 * R_{X_1X_2} * h_2^{*(-)}\right)(m),$$

on obtient simplement, après transformée de Fourier,

$$\boxed{S_{Y_1Y_2}(f) = H_1(f) S_{X_1X_2}(f) H_2^*(f)},$$

où $S_{Y_1Y_2}(f), S_{X_1X_2}(f), H_1(f)$ et $H_2(f)$ sont respectivement les transform'ees de Fourier de $R_{Y_1Y_2}(m)$, $R_{X_1X_2}(m)$, $h_1(m)$ et $h_2(m)$. Note{La transformée de Fourier de $h^*(-n)$ vaut $H^*(f)$.}

Conséquences :

1. En prenant $Y_1 = Y_2 = Y$, $X_1 = X_2 = X$ et $H_1 = H_2 = H$, c'est-à-dire que l'on considère un seul filtre, il vient

$$\boxed{S_{YY}(f) = S_{XX}(f)|H(f)|^2}.$$

2. Si $H_1(f)$ et $H_2(f)$ sont deux filtres *disjoints* en fréquence, alors

$$S_{Y_1Y_2}(f) = 0.$$

On en déduit que

$$R_{Y_1Y_2}(\tau) = \mathrm{TF}^{-1}[S_{Y_1Y_2}(f)] = \mathrm{TF}^{-1}[0] = 0.$$

si les filtres sont disjoints en fréquence, l'intercorrélation des sorties est nulle.

Application Considérons deux filtres parfaits autour de deux fréquences pures f_1 et f_2, de même entrée $X(n,\omega)$. On a $Y_1(n,\omega) = X(f_1,\omega)\exp(-j2\pi f_1 n)$, et $Y_2(n,\omega) = X(f_2,\omega)\exp(-j2\pi f_2 n)$, avec toutes les précautions d'usage sur la << non existence >> de la transformée de Fourier considérée pour des signaux aléatoires stationnaires. Dans ces conditions,

$$R_{Y_1Y_2}(0) = \mathbb{E}[X(f_1,\omega)X^*(f_2,\omega)]\exp(-j2\pi(f_1-f_2)n) = 0,$$

soit

$$\mathbb{E}[X(f_1,\omega)X^*(f_2,\omega)] = 0.$$

On dit que les composantes spectrales sont décorrélées.

12.5.1 Notion de densité spectrale de Puissance

La densité spectrale de puissance représente la répartition de la puissance du signal dans le domaine fréquentiel. Il s'agit exactement de la même notion que celle de densité de probabilité : lorsque l'on veut calculer probabilité qu'une variable aléatoire X appartienne à un certain intervalle $[x_1,x_2]$, il suffit d'intégrer la densité de probabilité de la variable entre ces deux bornes :

$$\Pr(X \in [x_1,x_2]) = \int_{x_1}^{x_2} p(X)\mathrm{d}X.$$

Si on appelle $D_{XX}(f)$ la densité spectrale de puissance d'un signal aléatoire $X(n,\omega)$, alors la puissance du signal portée par les composantes fréquentielles comprises entre f_1 et f_2 s'écrit

$$P_{XX}(f \in [f_1,f_2]) = \int_{f_1}^{f_2} D_{XX}(f)\mathrm{d}f.$$

Dès lors, la puissance totale du signal s'écrit

$$P_{XX} = \int_{-\frac{1}{2}}^{+\frac{1}{2}} D_{XX}(f)\mathrm{d}f.$$

Or on sait que, pour un signal stationnaire et ergodique,

$$P_{XX} = \mathbb{E}\left[|X(n,\omega)|^2\right] = R_{XX}(0) = \lim_{N\to+\infty}\frac{1}{2N}\sum_{-N}^{+N}|X(n,\omega)|^2.$$

Par ailleurs,

$$R_{XX}(\tau) = \int_{-\frac{1}{2}}^{+\frac{1}{2}} S_{XX}(f)\exp(j2\pi f\tau)\,\mathrm{d}f,$$

soit, pour $\tau = 0$,

$$R_{XX}(0) = P_{XX} = \int_{-\frac{1}{2}}^{+\frac{1}{2}} S_{XX}(f)\ \mathrm{d}f.$$

La transformée de Fourier $S_{XX}(f)$ de la fonction d'autocorrélation est ainsi une bonne candidate pour être la densité spectrale de puissance. Notons cependant, cette dernière relation ne prouve pas qu'elle le soit.

Considérons un filtre parfait, dont le module de la fonction de transfert est d'amplitude un dans une bande Δf centrée sur une fréquence f_0, et nul ailleurs :

$$\begin{cases} |H(f)| = 1 \text{ pour } f \in [f_0 - \frac{\Delta f}{2}, f_0 + \frac{\Delta f}{2}] \\ |H(f)| = 0 \text{ ailleurs.} \end{cases}$$

Notons $Y(n,\omega) = (h * X)(n,\omega)$ la réponse de ce filtre à une entrée $X(n,\omega)$. La puissance de la sortie est donnée par

$$P_{YY} = R_{YY}(0) = \int_{-\frac{1}{2}}^{+\frac{1}{2}} S_{YY}(f)\, \mathrm{d}f,$$

or la formule des interférences fournit

$$S_{YY}(f) = S_{XX}(f)|H(f)|^2,$$

avec les conditions sur le module de $H(f)$ données précédemment. On obtient donc

$$P_{YY}(f \in [f_0 - \frac{\Delta f}{2}, f_0 + \frac{\Delta f}{2}]) = \int_{f_0 - \frac{\Delta f}{2}}^{f_0 + \frac{\Delta f}{2}} S_{XX}(f)\, \mathrm{d}f,$$

ce qui correspond bien à la définition de la densité spectrale de puissance : la puissance pour les composantes spectrales comprises dans un intervalle est bien égale à l'intégrale de la densité spectrale de puissance sur cet intervalle. Si Δf est suffisamment faible, on pourra considérer la densité spectrale de puissance $S_{XX}(f)$ comme approximativement constante sur l'intervalle, et

$$P_{YY}(f \in [f_0 - \frac{\Delta f}{2}, f_0 + \frac{\Delta f}{2}]) \simeq S_{XX}(f_0)\Delta f.$$

Cette dernière relation indique que la densité spectrale de puissance doit s'exprimer en Watts par Hertz. Par ailleurs, lorsque Δf tend vers 0, la puissance recueillie est de plus en plus faible. Pour $\Delta f = 0$, la puissance obtenue est ainsi normalement nulle, sauf si la densité spectrale elle-même est constituée par une << masse >> de Dirac (de largeur nulle mais d'amplitude infinie) à la fréquence considérée.

Notons que le filtre que nous avons défini ci-dessus n'est défini, par commodité de présentation, que pour les fréquences positives. Sa fonction de transfert ne vérifie donc pas la propriété de symétrie hermitienne des signaux réels : la réponse impulsionnelle associée est donc complexe et la sortie $Y(t,\omega)$ également complexe. En restaurant cette symétrie, c'est-à-dire en imposant $H(f) = H^*(-f)$, ce qui entraîne (notez le module de f)

$$\begin{cases} |H(f)| = 1 \text{ pour } |f| \in [f_0 - \frac{\Delta f}{2}, f_0 + \frac{\Delta f}{2}] \\ |H(f)| = 0 \text{ ailleurs,} \end{cases}$$

la puissance en sortie est

$$P_{YY} = \int_{-f_0 - \frac{\Delta f}{2}}^{-f_0 + \frac{\Delta f}{2}} S_{XX}(f)\, \mathrm{d}f + \int_{f_0 - \frac{\Delta f}{2}}^{f_0 + \frac{\Delta f}{2}} S_{XX}(f)\, \mathrm{d}f.$$

La densité spectrale de puissance d'un signal aléatoire réel est une fonction paire, ce qui conduit enfin à

$$P_{YY} = 2 \int_{f_0 - \frac{\Delta f}{2}}^{f_0 + \frac{\Delta f}{2}} S_{XX}(f)\, \mathrm{d}f,$$

relation qui indique que la puissance se partage équitablement dans les fréquences positives et négatives.

Exemple :

Considérons le cas d'une sinusoïde à amplitude et phase aléatoire

$$X(n,\omega) = A(\omega)\sin(2\pi f_0 n + \phi(\omega)),$$

où $A(\omega)$ est une variable aléatoire centrée de variance σ^2 et $\phi(\omega)$ uniformément répartie sur $[0,2\pi]$. La fonction d'autocorrélation de ce signal vaut

$$R_{XX}(\tau) = \frac{\sigma^2}{2}\cos(2\pi f_0 \tau).$$

Par transformée de Fourier, on obtient la densité spectrale :

$$S_{XX}(f) = \frac{\sigma^2}{4}[\delta(f+f_0) + \delta(f-f_0)].$$

Enfin, en intégrant la densité spectrale

$$\int \frac{\sigma^2}{4}[\delta(f+f_0) + \delta(f-f_0)]\mathrm{d}f = \frac{\sigma^2}{2},$$

on retrouve la puissance de la sinusoïde, $\sigma^2/2$, comme il se doit.

Les fonctions de corrélation et les densités spectrales de puissance forment des paires de transformées de Fourier :

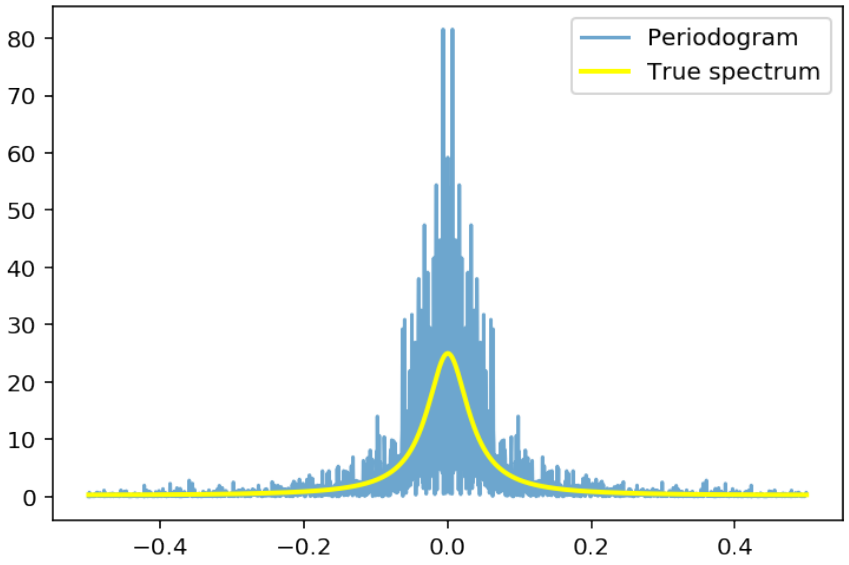
$$\boxed{\begin{array}{rcl} S_{XX}(f) & \rightleftharpoons & R_{XX}(\tau), \\ S_{XY}(f) & \rightleftharpoons & R_{XY}(\tau), \end{array}}$$

où $S_{XX}(f)$, $S_{XY}(f)$ sont les densités spectrale de puissance et de puissance d'interaction, respectivement. Ces relations constituent le **théorème de Wiener-Kintchine-Einstein**.

12.5.2 Power spectrum estimation

```
from scipy.signal import lfilter
from numpy.fft import fft, ifft, fftshift, fftfreq
```

```
N = 2000
a = -0.8
x = stats.norm(0, 1).rvs((N))
y = lfilter([1], [1, a], x)
Yf = fft(y)
Py = 1 / N * abs(Yf)**2
f = fftfreq(N)
f = np.linspace(-0.5, 0.5, N)
Sy = abs(1 / abs(fft([1, a], N))**2)
plt.plot(f, fftshift(Py), alpha=0.65, label="Periodogram")
plt.plot(f, fftshift(Sy), color="yellow", lw=2, label="True spectrum")
plt.legend()
#
# Smoothing
#
Ry = ifft(Py)
hh = sig.hamming(200, sym=True)
```




```
z = np.zeros(N)
L = 100
h = fftshift(sig.windows.hann(L, sym=True))
z[0:round(L / 2)] = h[0:round(L / 2)]
z[-1:-round(L / 2) - 1:-1] = h[-1:-round(L / 2) - 1:-1]
Py_smoothed = abs(fft(z * Ry))
plt.plot(f, fftshift(Py), alpha=0.6, label="Periodogram")
plt.plot(f, fftshift(Sy), lw=2, color="yellow", label="True spectrum")
plt.plot(
    f,
    fftshift(Py_smoothed),
    alpha=0.7,
    color="lightgreen",
    lw=2,
    label="Smoothed \nPeriodogram")
_ = plt.legend()
figcaption("Smoothed Periodogram")
```

```
#Averaging
def averaged_perio(y, M):
    N = np.size(y)
    L = int(np.round(N / M))
    Py_averaged = np.zeros(N)
    for m in range(M):
        Py_averaged += 1 / L * (abs(fft(y[m * L:(m + 1) * L], N))**2)
    return Py_averaged / M


Py_averaged = averaged_perio(y, 20)

plt.plot(f, fftshift(Py), alpha=0.6, label="Periodogram")
plt.plot(f, fftshift(Sy), lw=2, color="yellow", label="True spectrum")
plt.plot(
    f,
    fftshift(Py_averaged),
    alpha=0.7,
```

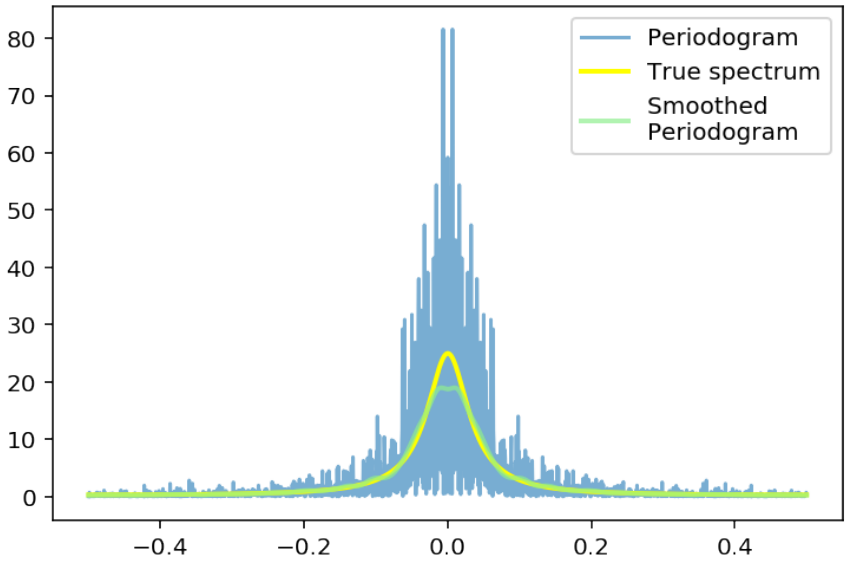


Figure 12.1: Smoothed Periodogram

```
    color="lightgreen",
    lw=2,
    label="Averaged \nPeriodogram")
_ = plt.legend()
figcaption("Averaged Periodogram")
```

12.6 Applications

12.6.1 Matched filter

We consider a problem frequently encountered in practice, in applications as echography, seismic reflexion, sonar or radar. The problem at hand is as follows: we look for a *known waveform* $s(n)$, up to a delay n_0 in a mixture

$$y(n) = As(n-n_0) + v(n),$$

where A and n_0 are unknowns and $v(n)$ is an additive noise. The problem is to find the delay n_0, which typically corresponds to a time-to-target. In order to do that, supppose that we filter the mixture by a filter with impulse response h. The ouput has the form

$$z(n) = x(n) + w(n),$$

with $x(n) = A[h * s](n-n_0)$ and $w(n) = [h * v](n)$, respectively the outputs of the signal and noise part. Clearly, if $v(n)$ is stationnary, so is $w(n)$. Therefore, the idea is to design h so that the signal output is as greater as possible than the noise output, at time n_0. In statistical terms, we put this as choosing the filter such that ratio of the signal output'*power* to the noise output's power is maximum. Hence, our goal is to design a filter which maximizes the signal-to-noise ratio at time n_0. We suppose that the desired signal is deterministic and thus consider its instantaneous power $|x(n_0)|^2$.

The signal-to-noise ratio at time n_0 is

$$SNR(n_0) = \frac{|x(n_0)|^2}{\mathbb{E}\left[|w(n)|^2\right]}.$$

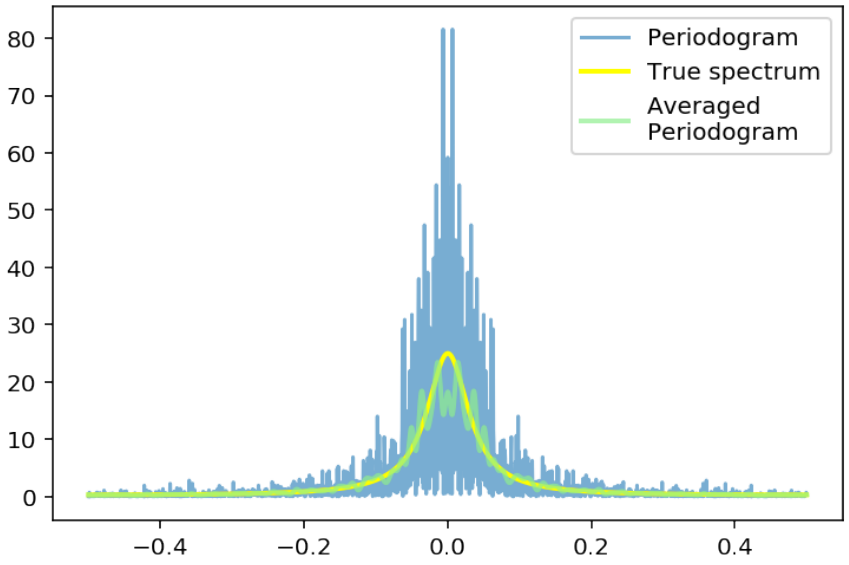


Figure 12.2: Averaged Periodogram

Of course, both the numerator and the denominator depends on the filter. Lets us first consider the numerator. We have

$$x(n_0) = \mathrm{FT}^{-1}[X(f)]_{n=n_0} \tag{12.3}$$

$$= \int H(f)\mathrm{FT}[s(n-n_0)]e^{j2\pi fn}\mathrm{d}f\Big|_{n=n_0} \tag{12.4}$$

$$= \int H(f)S(f)e^{-j2\pi fn_0}e^{j2\pi fn}\mathrm{d}f\Big|_{n=n_0} \tag{12.5}$$

$$= \int H(f)S(f)\mathrm{d}f. \tag{12.6}$$

As far as the denominator is concerned, we have by the Wiener-Kintchine theorem, that

$$\mathbb{E}\left[|w(n)|^2\right] = \int S_{WW}(f)\mathrm{d}f = \int |H(f)|^2 S_{VV}(f)\mathrm{d}f.$$

Finally, the signal-to-noise ratio becomes

$$\boxed{SNR(n_0) = \frac{|\int H(f)S(f)\mathrm{d}f|^2}{\int |H(f)|^2 S_{VV}(f)\mathrm{d}f}.}$$

In order to maximize the signal-to-noise ratio we invoke the Cauchy-Schwarz inequality. Recall that that this inequality states that given to integrable functions f and g and a positive measure w, then

$$\left|\int f(x)g(x)^* w(x)\mathrm{d}x\right|^2 \leq \int |f(x)|^2 w(x)\mathrm{d}x \int |g(x)|^2 w(x)\mathrm{d}x$$

with equality if and only if $f(x) = kg(x)$ for any arbitrary real constant k.

The idea is to apply this inequality in order to simplify the $SNR(n_0)$. For that, let us express the numerator as

$$\int H(f)S(f)\mathrm{d}f = \int H(f)\sqrt{S_{VV}(f)}\frac{S(f)}{\sqrt{S_{VV}(f)}}\mathrm{d}f.$$

By the Cauchy-Schwarz inequality, we then get that

$$\left|\int H(f)S(f)\mathrm{d}f\right|^2 \leq \int |H(f)|^2 S_{VV}(f)\mathrm{d}f \int \left|\frac{S(f)}{\sqrt{S_{VV}(f)}}\right|^2 \mathrm{d}f$$

Injecting this inequality in the $SNR(n_0)$ we obtain that

$$\boxed{SNR(n_0) \leq \int \left|\frac{S(f)}{\sqrt{S_{VV}(f)}}\right|^2 \mathrm{d}f}.$$

This shows that the SNR at n_0 **is upper bounded** by a quantity which is independent of $H(f)$. Furthermore, by the conditions for equality in the Cauchy-Schwartz inequality, we have that the bound is attained if and only if

$$H(f) = k\frac{S(f)^*}{S_{VV}(f)}.$$

In the special case where $v(n)$ is a white, then $S_{VV}(f)$ is a constant, say $S_{VV}(f) = \sigma^2$, and

$$H(f) = k'S(f)^*.$$

By inverse Fourier transform, the corresponding impulse response is nothing but

$$h(n) = k's(-n)^*,$$

that is, the **complex conjugate and reversed** original waveform. This will be important to link the output of the filter to an estimate of the cross-correlation function. For now, let us also observe that the general transfer function $H(f)$ can be interpreted as a a **whitening operation** followed by the matched filter for an additive white noise:

$$H(f) = k\frac{S(f)^*}{S_{VV}(f)} = k\underbrace{\frac{1}{\sqrt{S_{VV}(f)}}}_{\text{whitening}} \times \underbrace{\frac{S(f)^*}{\sqrt{S_{VV}(f)}}}_{\text{matched filter}}$$

Finally, the output of the matched filter can be viewed as the computation of an estimated of the cross-correlation function. Indeed, the output of the $h(n)$ with input x is

$$\begin{aligned} y(n) &= \sum_l h(l)x(n-l) && (12.7)\\ &= \sum_l s(-l)^* x(n-l) && (12.8)\\ &= \sum_m s(m)^* x(n+m) && (12.9)\\ &= \hat{R}_{xs}(n), && (12.10)\end{aligned}$$

where $\hat{R}_{xs}(n)$ is, up to a factor, an estimate of the cross-correlation between x and s. Applying this remark to our initial mixture

$$y(n) = As(n-n_0) + v(n)$$

we get that

$$z(n) = A\hat{R}_{ss}(n-n_0) + \hat{R}_{vs}(n).$$

Finally, since v and s are uncorrelated, $\hat{R}_{vs}(n) \simeq 0$ and since $\hat{R}_{ss}(n)$ is maximum at zero, we see that the output will present a peak at $n = n_0$, thus enabling to locate the value of the delay n_0.

Matched filter - Experiment

We simulate now a problem in seismic reflection (or in sonar, or radar), where the goal is to detect the positions of interfaces reflecting the incident waves. The time it takes for a reflection from a particular boundary to arrive at the recorder (a geophone) is called the travel time. For a simple vertically traveling wave, the travel time τ from the surface to the reflector and back is called the Two-Way Time (TWT) and is given by the formula $\tau = 2d/c$, with d the distance from the origin to the refector. To a whole set of interfaces then corresponds the observation

$$r(t) = \sum A_i s(t - t_i) + b(t)$$

where the t_i are the delays associated with each interface and A_i the reflection coefficients.

In order to localize the interfaces, we use a matched filter, which maximizes the signal to noise ratio.

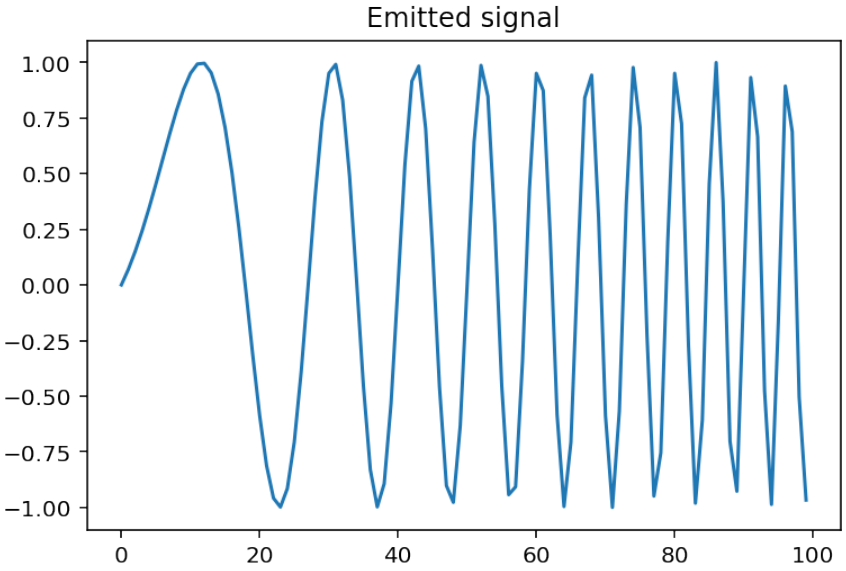
1. Implement the matched filter. Examine the different signals. Is it possible to detect the positions of the interfaces on the time series? using the correlation functions? What is the interest to choose a stimulation signal with a very peaky autocorrelation?

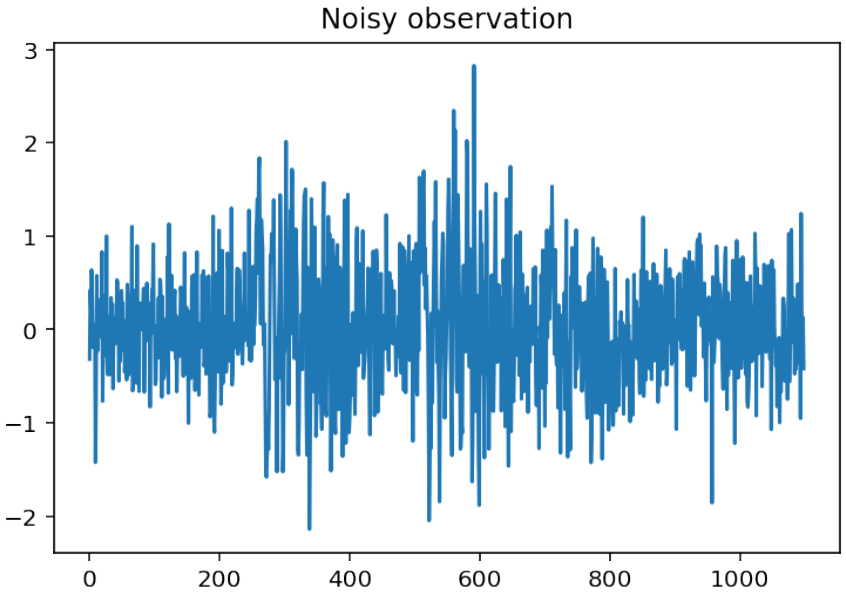
2. Consider a noisy version of the observation (add a Gaussian noise with standard deviation A). Compute the output of the matched filter, with impulse response $h(n) = s(-n)$ and introduce a threshold at 3.3 times the noise standard deviation. Interpret this threshold. Conclusions. Experiment with the level of noise, the number of samples, etc

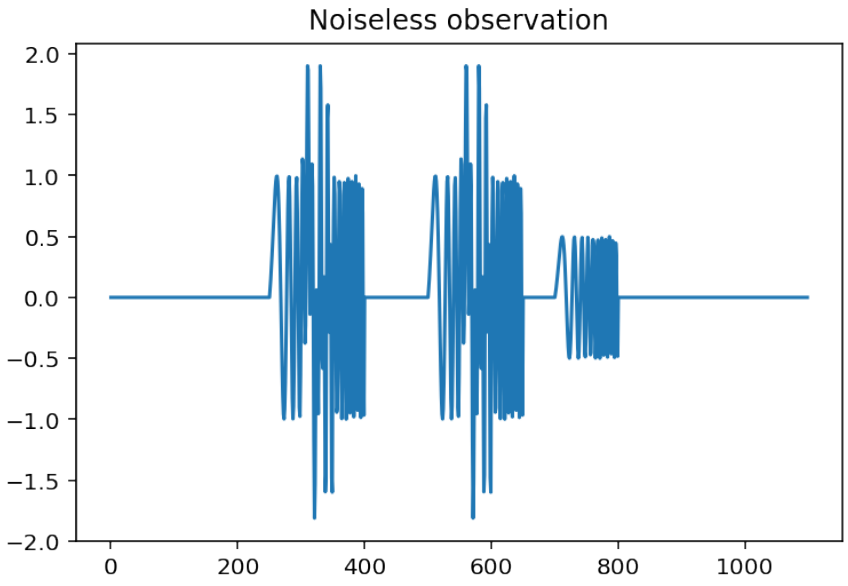
```
def zeropad(v, N):
    a = zeros(N)
    a[arange(len(v))] = v
    return a
```

```
N = 1000
#Interface detection by cross-correlation
t = np.arange(100)
A = 0.5
s = 1 * sin(2 * pi * 0.01 * (1 + 0.1 * t) * t)  #emitted signal
figure()
plot(t, s)
title('Emitted signal')
# List of interfaces
pos = array([250, 300, 500, 550, 700])
amp = array([1, 1, 1, 1, 0.5])
g = zeros(N)
g[pos] = amp
y = np.convolve(s, g)
z = y + A * randn(size(y))
figure(2)
plot(z)
title('Noisy observation')
figure(3)
plot(y)
title('Noiseless observation')
```

```
Text(0.5, 1.0, 'Noiseless observation')
```


Emitted signal
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
20
40
60
80
100


Noisy observation
3
2
1
0
−1
−2
0
200
400
600
800
1000




```
from correlation import xcorr
sp = zeropad(s, len(z))
figure(4)
Rzs, lags = xcorr(z, sp)
plot(lags, Rzs)
title('Cross-correlation (noisy case)')
figure(5)
Rys, lags = xcorr(y, sp)
plot(lags, Rys)
title('Cross-correlation (noiseless case)')
```

```
Text(0.5, 1.0, 'Cross-correlation (noiseless case)')
```

Finally, we introduce a threshold in order to eliminate the peaks due to the noise. For that, we compute the threshold so as to have less than some fixed probability to exceed this level.

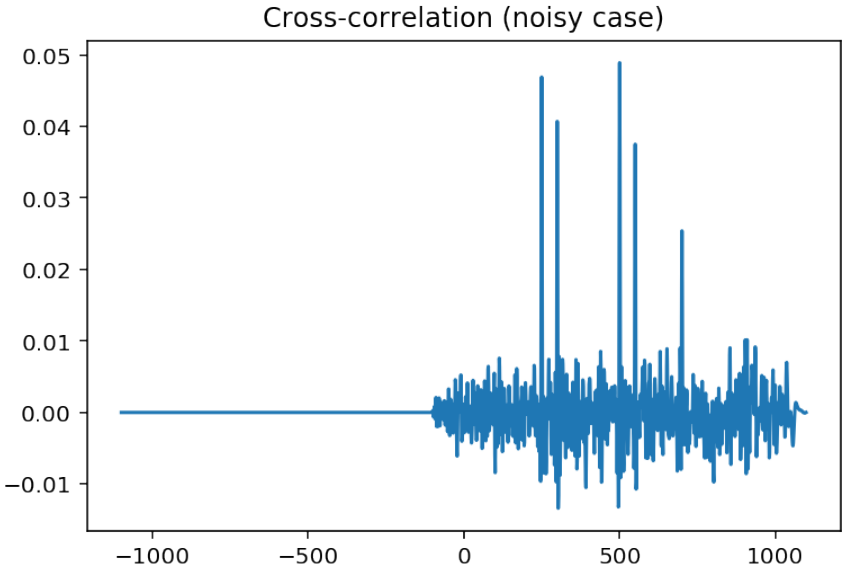
The method `interval` of an object `stats.norm` returns the endpoints of the range that contains alpha percents of the distribution.

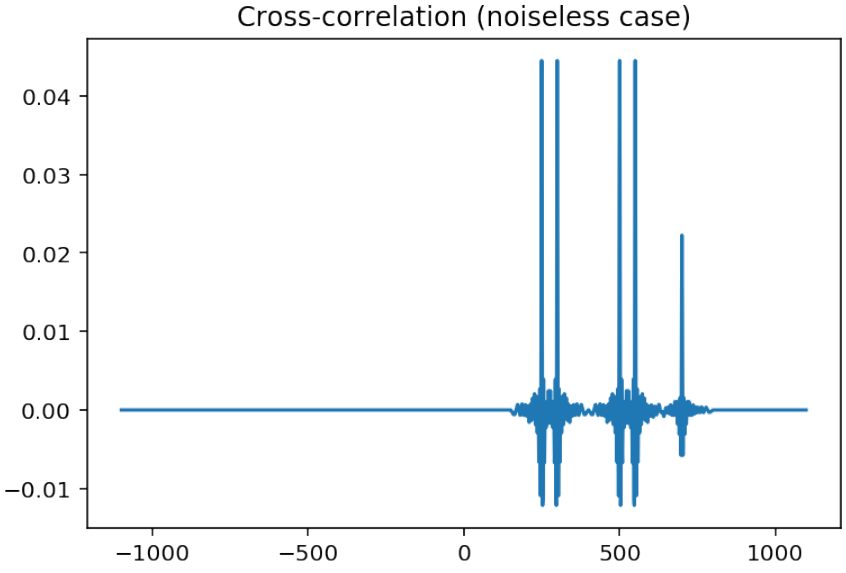
```
interv = stats.norm.interval(alpha=0.999, loc=0, scale=1)
print(interv)
```

```
(-3.2905267314918945, 3.2905267314919255)
```

And the actual thresholding:

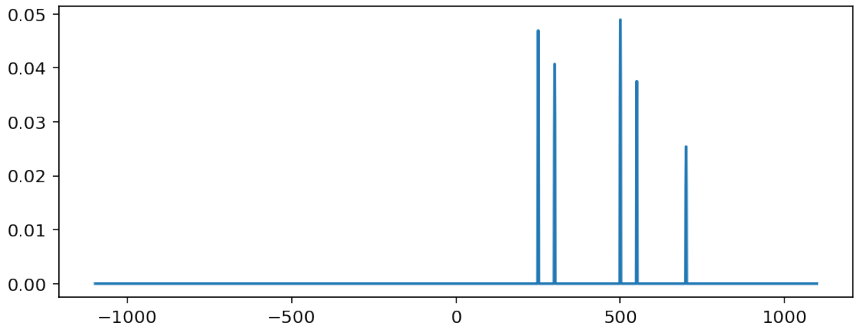
```
LR = len(Rzs)
Rzs_th = zeros(LR)
intervs = array(interv) * std(Rzs[500:])

Rzs_th = array([
    Rzs[u] if (Rzs[u] < intervs[0] or Rzs[u] > intervs[1]) else 0
```


Cross-correlation (noisy case)
0.05
0.04
0.03
0.02
0.01
0.00
−0.01
−1000
−500
0
500
1000


Cross-correlation (noiseless case)
0.04
0.03
0.02
0.01
0.00
−0.01
−1000
−500
0
500
1000

```
    for u in range(LR)
])
fig, ax = subplots(1, 1, figsize=(8, 3))
ax.plot(lags, Rzs_th)
print("The position of interfaces are at", where(Rzs_th != 0)[0] + lags[0])
```

```
The position of interfaces are at [249 250 251 299 300 301 499 500 501 549 550 551
```



Quick and Dirty thing to find the “center” of consecutive value ranges

```
def find_center(v):
    Beg = v[0]
    Endy = v[0]
    u = 0
    C = []
    for k in range(1, len(v)):
        if (v[k] - v[k - 1]) in (1, 2):
            Endy = Endy + 1
        else:
            C.append((Endy + Beg) / 2)
            u = u + 1
            Beg = v[k]
            Endy = v[k]
    if Endy == v[len(v) - 1]:
        C.append((Endy + Beg) / 2)
    return C
```

```
posit = find_center(where(Rzs_th != 0)[0] + lags[0])
print("Positions where the signal exceeds threshold:\n".ljust(35),
      where(Rzs_th != 0)[0] + lags[0])
print("Detected interfaces positions: ".ljust(35), posit)
print("True positions; ".ljust(35), pos)
```

```
Positions where the signal exceeds threshold:
 [249 250 251 299 300 301 499 500 501 549 550 551 699 700 701]
Detected interfaces positions:      [250.0, 300.0, 500.0, 550.0, 700.0]
True positions;                     [250 300 500 550 700]
```

12.6.2 Wiener filtering

Introduction

We consider now the problem of recovering a signal $s(n)$ from an indirect and noisy measurement

$$x(n) = [h * s](n) + v(n).$$

This problem involves actually two sub-problems that are very interesting on their own: - *smoothing* of the additive noise, - inversion.

Let us first examine a simple experiment which points-out the necessity of developing a rational approach instead of a adopting a naive one. We generate a random pulse train, filter it, and then reconstruct the input signal by direct division by the transfer function:

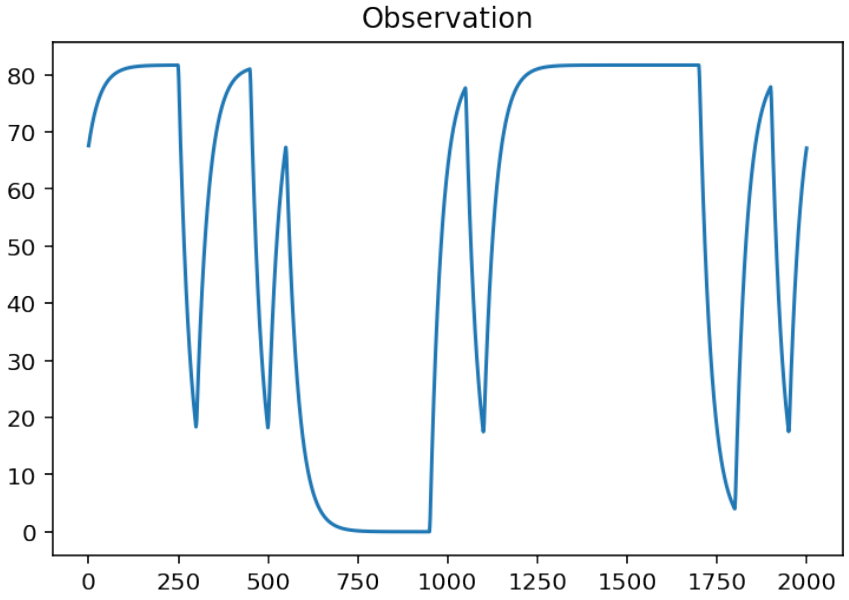
$$S(f) \simeq \frac{X(f)}{H(f)} = S(f) + \frac{V(f)}{H(f)}$$

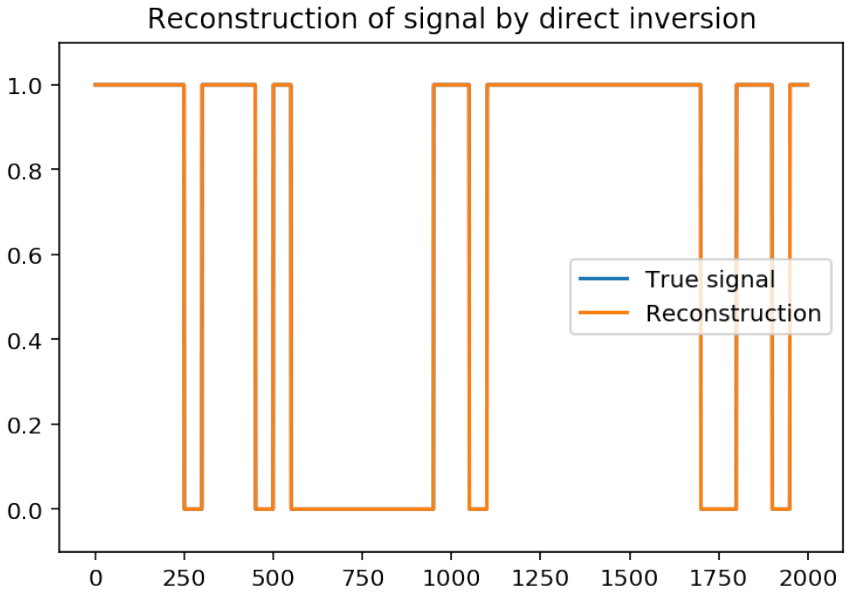
We consider both a noiseless case and a noisy case.

Illustrative experiment

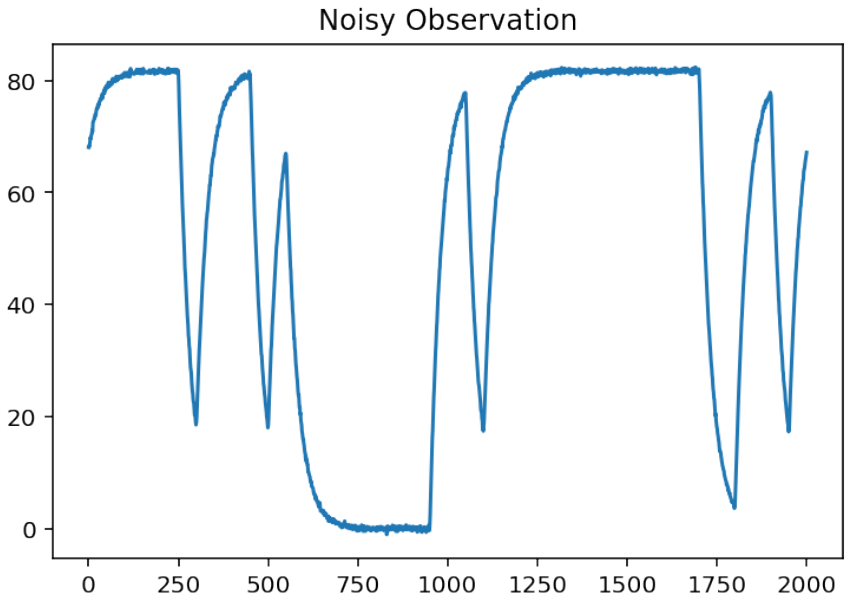
```
N = 2000
a = -0.97
L = 50
spos = stats.bernoulli.rvs(loc=0, p=0.6, size=int(N / L))
s = np.kron(spos, np.ones(L))
#x=stats.norm(0,1).rvs((N))
d = np.zeros(N)
d[0] = 1  #Dirac impulse
h = sig.lfilter([1, 0.5, 0.95], [1, a], d)
#h=sig.lfilter([1, 0.6, 0.95, 1.08, 0.96],[1, a],d)
H = fft(h, N)
X = fft(s) * H
x = real(ifft(X))

plt.figure()
plt.plot(x)
plt.title("Observation")

#
plt.figure()
x_rec = real(ifft(X / H))
plt.plot(s, label="True signal")
plt.plot(x_rec, label="Reconstruction")
plt.title("Reconstruction of signal by direct inversion")
plt.ylim([-0.1, 1.1])
_ = plt.legend()
```

```
# Noisy observation
z = x + 0.25 * stats.norm(0, 1).rvs((N))
Z = fft(z)
plt.figure()
plt.plot(z)
plt.title("Noisy Observation")
plt.figure()
x_rec = real(ifft(Z / H))
plt.plot(s, label="True signal")
```


Observation
80
70
60
50
40
30
20
10
0
0
250
500
750
1000
1250
1500
1750
2000


Reconstruction of signal by direct inversion
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
2000
True signal
Reconstruction

```
plt.plot(x_rec, alpha=0.6, label="Reconstruction")
plt.title("Reconstruction of (noisy) signal by direct inversion")
_ = plt.legend()
```




```
plt.plot(1 / abs(H))
```

```
[<matplotlib.lines.Line2D at 0x7f3fa40f5b38>]
```

Derivation of the Wiener filter

Instead of a direct inversion, we put the problem as the design of a filter w which enables to recover an estimate of $s(n)$, from the noisy observation $x(n)$.

$$y(n) = [w * x](n)$$

The objective is to minimize the error $e(n) = y(n) - s(n)$, and more precisely of the mean square error

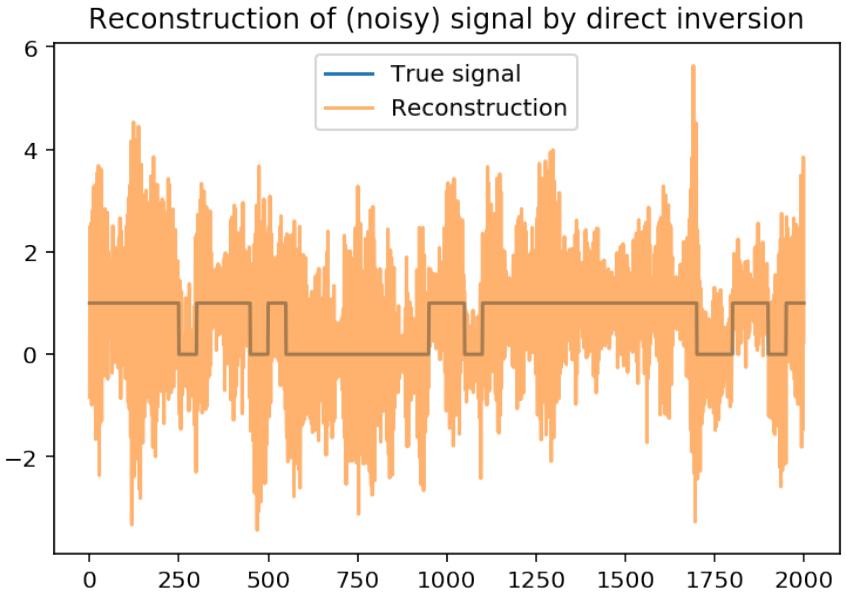
$$\mathbb{E}\left[e(n)^2\right].$$

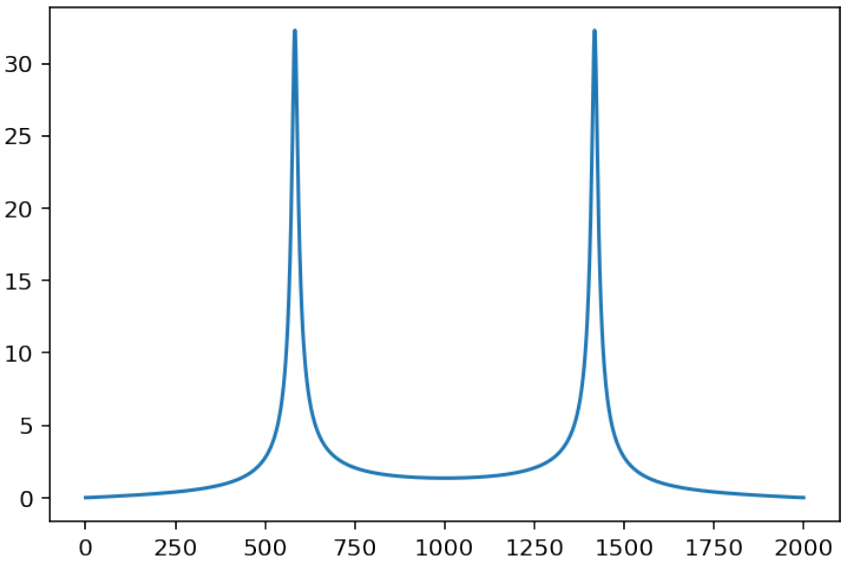
Recall that

$$\mathbb{E}\left[e(n)^2\right] = R_{EE}[0] = \int S_{EE}(f)\mathrm{d}f.$$

Since $e(n) = y(n) - s(n)$, we have that

$$R_{Y-S,Y-S}(k) = R_{YY}(k) - R_{YS}(k) - R_{SY}(k) + R_{SS}(k) \tag{12.11}$$
$$S_{Y-S,Y-S}(f) = S_{YY}(f) - S_{YS}(f) - S_{SY}(f) + S_{SS}(f) \tag{12.12}$$


Reconstruction of (noisy) signal by direct inversion
True signal
Reconstruction
6
4
2
0
−2
0
250
500
750
1000
1250
1500
1750
2000


30
25
20
15
10
5
0
0
250
500
750
1000
1250
1500
1750
2000

From the transformation of the power spectrum by filtering and the symmetries of the cross-spectra, we have

$$S_{YY}(f) = |H(f)W(f)|^2 S_{SS}(f) + |W(f)|^2 S_{VV}(f), \tag{12.13}$$
$$S_{YS}(f) = H(f)W(f)S_{SS}(f), \tag{12.14}$$
$$S_{SY}(f) = S_{YS}(f)^*. \tag{12.15}$$

Taking this into account, we arrive at

$$S_{Y-S,Y-S}(f) = |H(f)|^2|W(f)|^2 S_{SS}(f) + |W(f)|^2 S_{VV}(f) + H(f)W(f)S_{SS}(f) + H(f)^*W(f)^*S_{SS}(f) + S_{SS}(f).$$

It is easy to check that this formula can be rewritten as

$$S_{Y-S,Y-S}(f) = (S_{SS}(f) + S_{VV}(f)) \left| W(f) - \frac{H(f)^* S_{SS}(f)}{|H(f)|^2 S_{SS}(f) + S_{VV}(f)} \right|^2 + S_{SS}(f).$$

Clearly, it is minimum if

$$\boxed{W(f) = \frac{H(f)^* S_{SS}(f)}{|H(f)|^2 S_{SS}(f) + S_{VV}(f)}}.$$

From this relation, we learn the following: - In the noiseless case, that is $S_{VV}(f) = 0$, then $W(f) = 1/H(f)$. THis is the diert inversion, which is only valid if no noise corrupts the output. - for frequencies where the transfer function $H(f)$ is very small, that is where we have a very small signal part,then $W(f) \sim 0$ (no inversion). - elsewhere, the filter makes a conservative inversion which depends on the local signal-to-noise ratio.

- In the case $H(f) = 1$, the problem reduces to a smoothing problem, that is to suppress the noise without too much corrupting of the signal part. The Wiener filter reduces to

$$\boxed{W(f) = \frac{S_{SS}(f)}{S_{SS}(f) + S_{VV}(f)}}. \tag{12.16}$$

In such case, we see that the transfer function tends to 1 if $S_{SS}(f) \gg S_{VV}(f)$ (frequency bands where the signal is significantly higher than the noise), to zero if $S_{SS}(f) \ll S_{VV}(f)$ (much more noise than signal), and otherwise realises a tradeoff guided by the signal-to-noise ratio in the frequency domain.

Experiment

We consider an example of optimum filtering, the Wiener smoother. Beginning with a noisy mixture $x(n) = s(n) + v(n)$, the goal is to find the best filter which minimizes the noise while preserving the signal: $y(n) = (h * x)(n) \simeq s(n)$.

Simulate a signal

$$s(n) = \exp(-at)\sin(2\pi f_0 t + \phi(\omega)).$$

The corresponding implementation lines are

```
A=0.2; N=5000
t=arange(N)
s=exp(-0.001*t)*sin(2*pi*0.001*t+2*pi*rand(1))
w=A*randn(N)
x=s+w
```

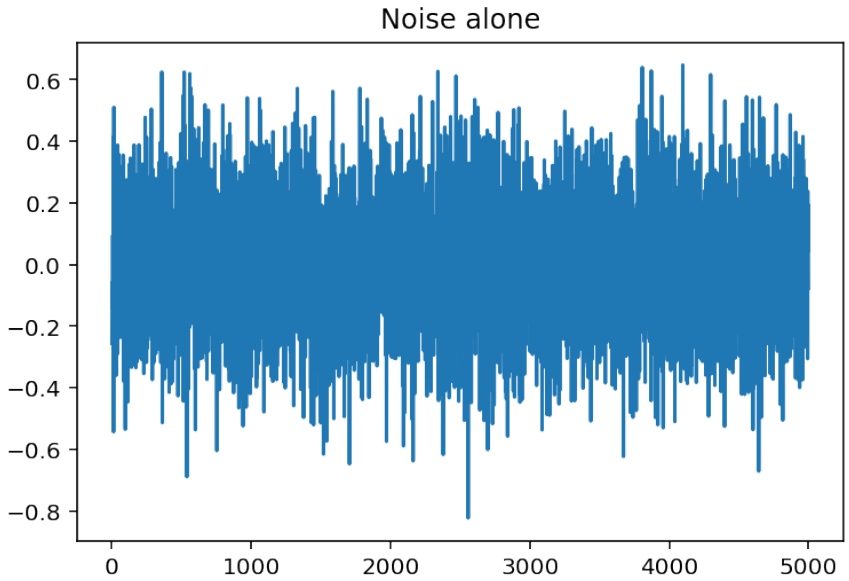
It can be shown that the optimum Wiener filter is such that

$$H(f) = \frac{S_{ss}(f)}{S_{SS}(f) + S_{VV}(f)},$$

where $S_{SS}(f)$ and $S_{VV}(f)$ are respectively the power spectra of the signal and of the noise. Implement this filter and compute its output. In practice, what must be known in order to implement this filter? Is this reasonable? Look at the impulse response and comment. What are the other difficulties for implementation?

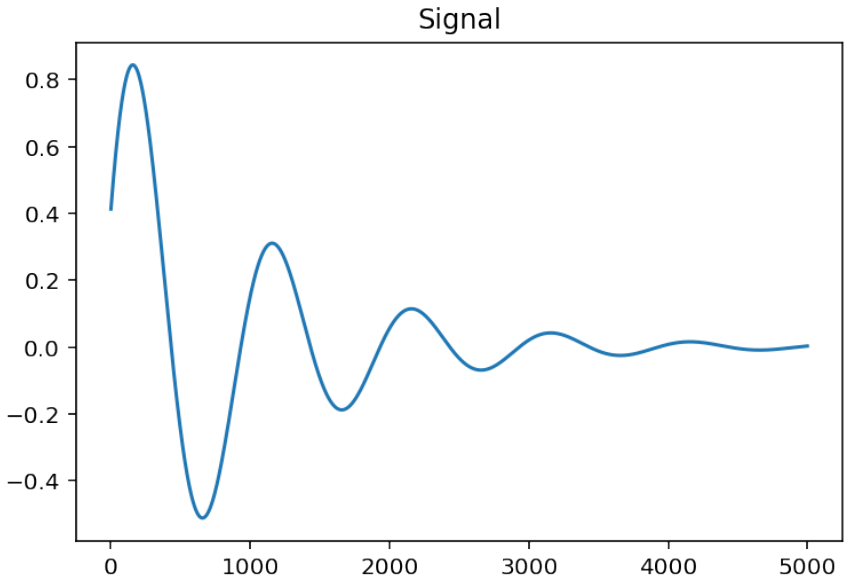
```
A = 0.2
N = 5000
t = arange(N)
s = exp(-0.001 * t) * sin(2 * pi * 0.001 * t + 2 * pi * rand(1))
w = A * randn(N)
figure(1)
plot(w)
title('Noise alone')
x = s + w
figure(2)
plot(s)
title('Signal')
figure(3)
plot(x)
title('Observed signal')
```

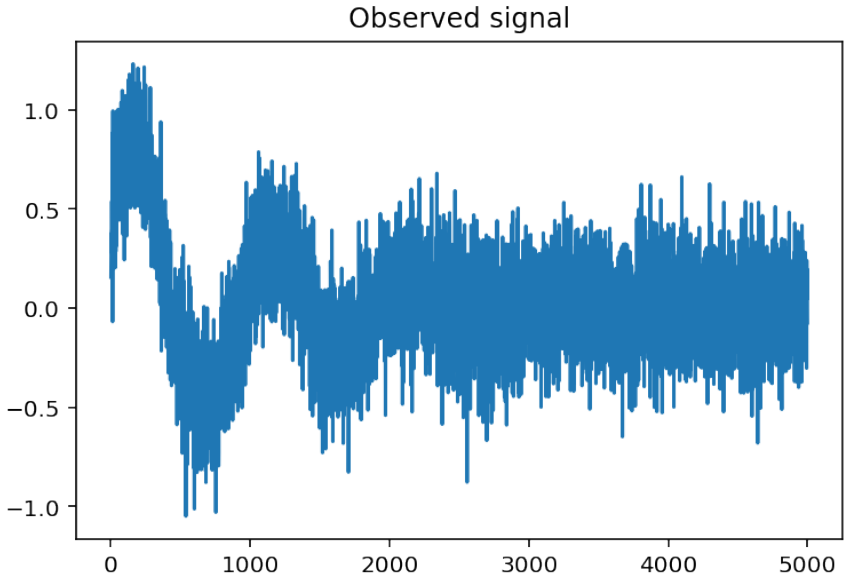
```
Text(0.5, 1.0, 'Observed signal')
```



Implementation

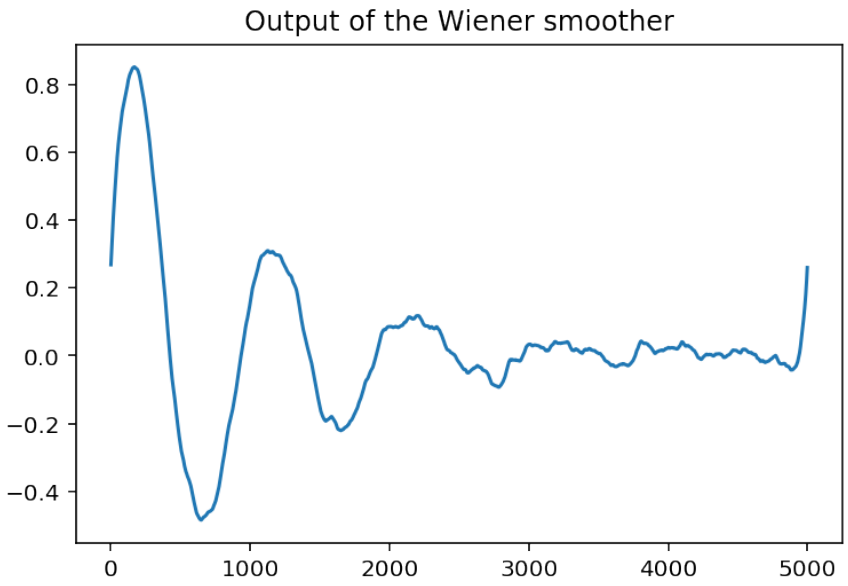
```
Sss = 1 / N * abs(fft(s))**2
Svv = A * A * ones(N)
H = Sss / (Sss + Svv)
xx = real(ifft(H * fft(x)))
```


Signal
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
1000
2000
3000
4000
5000


Observed signal
1.0
0.5
0.0
−0.5
−1.0
0
1000
2000
3000
4000
5000

```
plot(xx)
title('Output of the Wiener smoother')
```

```
Text(0.5, 1.0, 'Output of the Wiener smoother')
```



The drawbacks are that

- One must know the spectra of the signal and of the noise. Here we have suppposed that the noise is white and that we knew its variance. Furthermore, we assumed that the spectrum of the signal is known.
- The impulse response may have an infinite support and is not causal. For implementation in real time, on should select a causal solution. This requires to perform a spectral factorization and this is another story, see here or here, page 208 for details.

Wiener Smoother in the time domain

We now look for the expression of an optimal smoother in the time domain. Of course, we could simply take the impulse response associated with the frequency response (12.16). However, as we saw above, this impulse response is non-causal and has infinite duration. Instead, we shall reformulate the problem to include the fact that we look for a causal finite impulse response. We begin with the observation equation

$$x(n) = s(n) + v(n).$$

and we look for the filter with impulse response $w(n)$ such that $y(n) = [w * x](n)$ is as near as possible of $s(n)$: this can be formulated as the search for w which minimizes the mean square error

$$\mathbb{E}\left[([w * x](n) - s(n))^2\right].$$

For a FIR filter, the convolution cans be written as the scalar product $

$$

$$y(n) = [w * x](n) = \sum_{m=0}^{p-1} w(m)x(n-m) = \mathbf{w}^t \mathbf{x}(n)$$

where $\mathbf{w}^t = [w(0, w(1) \dots w(p-1)]$ and $\mathbf{x}(n)^t = [x(n), x(n-1), \dots x(n-p+1)]$. The mean square error can then be written as the function of $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}\left[\left(\mathbf{w}^t \mathbf{x}(n) - s(n)\right)^2\right].$$

By the chain rule for differentiation and the fact that

$$\frac{\mathrm{d}\mathbf{w}^t \mathbf{x}(n)}{\mathrm{d}\mathbf{w}} = \mathbf{x}(n),$$

we get that

$$\begin{aligned}
\frac{\mathrm{d}J(\mathbf{w})}{\mathrm{d}\mathbf{w}} &= 2\mathbb{E}\left[\mathbf{x}(n)\left(\mathbf{w}^t \mathbf{x}(n) - s(n)\right)\right] && (12.17)\\
&= 2\mathbb{E}\left[\mathbf{x}(n)\left(\mathbf{x}^t \mathbf{w}(n) - s(n)\right)\right], && (12.18)\\
&= 2\mathbb{E}\left[\mathbf{x}(n)\mathbf{x}(n)^t\right]\mathbf{w} - \mathbb{E}\left[\mathbf{x}(n)s(n)\right]. && (12.19)\\
& && (12.20)
\end{aligned}$$

The first term involves a correlation matrix of $\mathbf{x}(n)$ and the second the vector of cross correlations between $\mathbf{x}(n)$ and $s(n)$. Denoting

$$\begin{cases} \mathbf{R}_{XX} = \mathbb{E}\left[\mathbf{x}(n)\mathbf{x}(n)^t\right], \\ \mathbf{r}_{SX} = \mathbb{E}\left[\mathbf{x}(n)s(n)\right] \end{cases}$$

we obtain

$$\mathbf{R}_{XX}\mathbf{w} = \mathbf{r}_{SX}$$

or

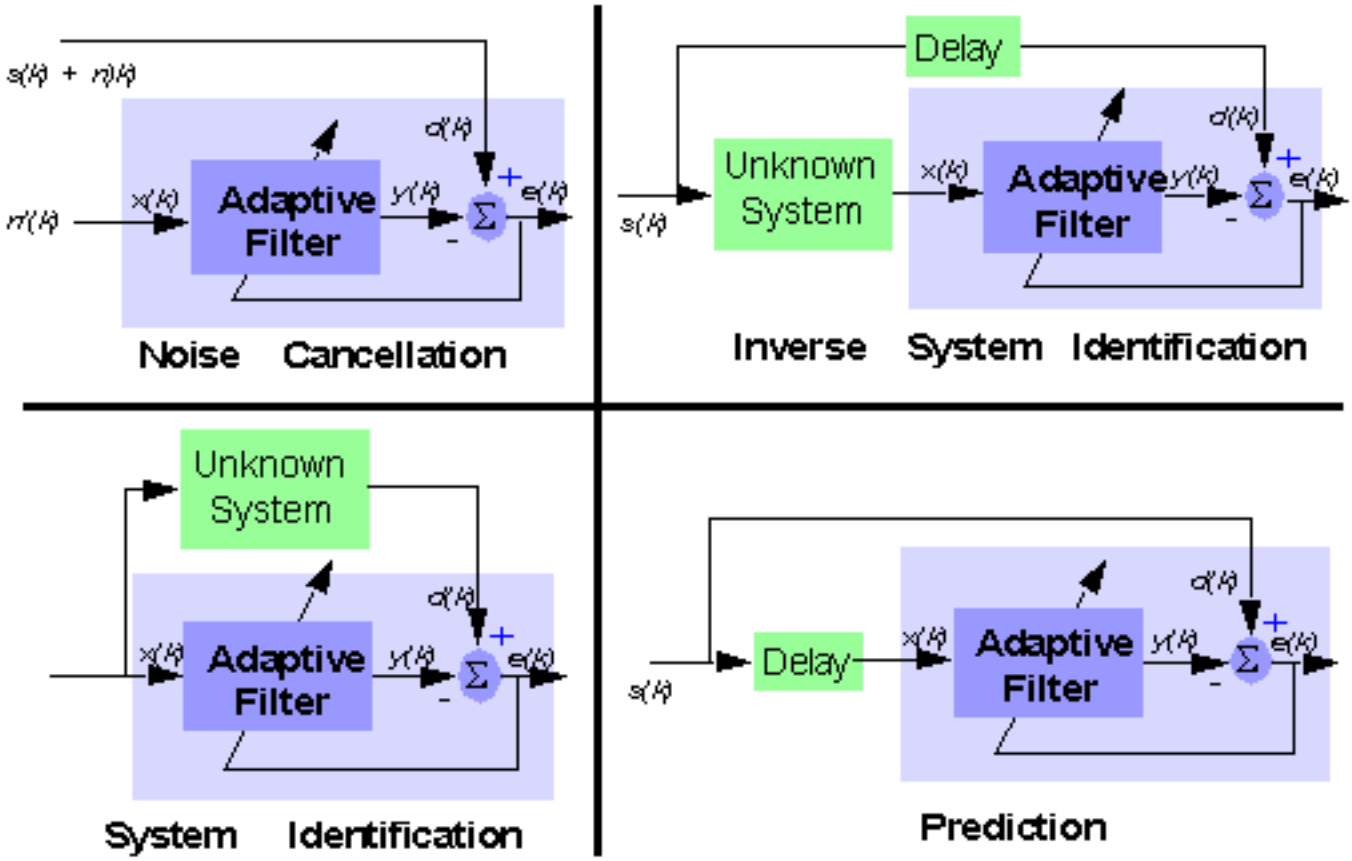
$$\boxed{\mathbf{w} = \mathbf{R}_{XX}^{-1}\mathbf{r}_{SX}}$$

if $\mathbf{R}_{XX}$ is invertible.

13

Adaptive Filters

Adaptive filters are systems that are able to adapt their coefficients with respect to the properties of their environment, in order to satisfy a given objective. Furthermore, they may also be able to adapt themselves to modifications of the environment and track them. Many real-world applications employ adaptive filters, as Hearing aids, Localization and tracking Active noise control (anti-noise), Noise suppression, Audio upmix of stereo signals,Adaptive beamforming, MPEG audio coding, Non-linear echo cancellation, Adaptation of neural networks, etc. The following figure, taken from [Ref][1], presents some possible applications:



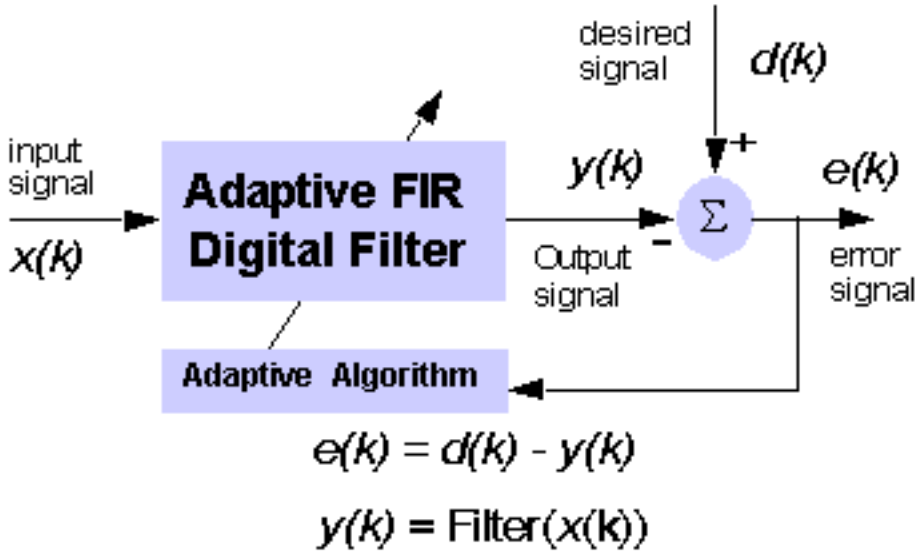
We will first begin by describing the general filtering problem and derive the optimal solution, known

as the Wiener filter. We will then explain how the solution can be obtained through iterative algorithms. Finally, we will describe how these algorithms can be turned into adaptive filters.

[1]: M. Harteneck and R.W. Stewart, Adaptive Digital Signal Processing JAVA Teaching Tool, IEEE TRANSACTIONS ON EDUCATION, MAY 2001, VOLUME 44, NUMBER 2, IEEDAB (ISSN 0018-9359) online here

13.1 A general filtering problem

In what follows, we will consider a general problem, which is sometimes called the `Wiener problem`. Many actual problems, including filter identification, noise cancellation, linear prediction, etc can be formulated as special cases of this Wiener problem.



The classical formulation is as follows: Given a random signal $u(n)$, we would like to find a transform $\mathcal{T}\{u\}$ such that the result is as close as possible to some `desired response` $d(n)$. We will restrict this general problem on two aspects.

- First, we will only consider `linear` transforms of the sequence $\{u(n)\}_{n=0..N-1}$; that is filterings of $u(n)$. Furthermore, we will even restrict ourselves to causal, finite impulse response filters with p taps. We denote by w (with w for Wiener) the impulse response. For now, we assume that the system is stationary, which implies that the impulse response does not depend on time n. Hence, the output can be computed as the convolution product

$$y(n) = [w * u](n) = \sum_{m=0}^{p-1} w(m)u(n-m) \tag{13.1}$$

- Second, the notion of "as close as" will be quantified by a cost function on the error

$$e(n) = y(n) - d(n). \tag{13.2}$$

  Any cost function could be used, such as $|\bullet|$, $|\bullet|^2$, $|\bullet|^3$ or even $\sinh e(n)\ldots$ Among these possibilities, the square of the error yields interesting, closed-form solutions and simple computations.

We can choose to work only with the sequences at hand and look at an integrated error such as

$$J_{\text{ls}}(w,n_0,n_1) = \sum_{n=n_0}^{n_1} e(n)^2 \tag{13.3}$$

Such a criterion is called the Least Square criterion. We may also choose to work with the stochastic processes on average, and consider a `mean square error`

$$J_{\text{mse}}(w,n) = \mathbb{E}\left[e(n)^2\right]. \tag{13.4}$$

The corresponding criterion is the Minimum Mean Square Error criterion.

13.1.1 Introduction

Definitions

13.1.2 The Linear Minimum Mean Square Error Estimator

Let us define by $\mathbf{u}(n)$ the $p \times 1$ column vector collecting p consecutive samples of the input

$$\mathbf{u}(n)^T = [u(n), u(n-1), \dots u(n-p+1)],$$

and by $\mathbf{w}$ the vector collecting the samples of the impulse response:

$$\mathbf{w}^T = [w(0), w(1), \dots w(p-1)].$$

Clearly, the output (13.1) of filter $\mathbf{w}$ can be written as the dot product $y(n) = \mathbf{w}^T\mathbf{u}(n)$, and the error is simply

$$e(n) = \mathbf{w}^T\mathbf{u}(n) - d(n).$$

Observe that $J_mse(w,n) $ is a quadratic form in $\mathbf{w}$. Therefore, the criterion admits a single global minimum. To see this, let us develop the MSE:

$$\begin{aligned} J_{\text{mse}}(\mathbf{w},n) &= \mathbb{E}\left[\left(\mathbf{w}^T\mathbf{u}(n) - d(n)\right)\left(\mathbf{u}(n)^T\mathbf{w}(n) - d(n)\right)\right] && (13.5)\\ &= \mathbf{w}^T\mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^T\right]\mathbf{w} - 2\mathbf{w}^T\mathbb{E}\left[\mathbf{u}(n)d(n)\right] + \mathbb{E}\left[d(n)^2\right] && (13.6)\\ &= \mathbf{w}^T\mathbf{R}_{uu}\mathbf{w} - 2\mathbf{w}^T\mathbf{R}_{du} + \sigma_d^2 && (13.7) \end{aligned}$$

where we denoted

$$\begin{cases} \mathbf{R}_{uu} = \mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^T\right] & \text{the correlation matrix of } \mathbf{u}(n) \\ \mathbf{R}_{du} = \mathbb{E}\left[d(n)\mathbf{u}(n)\right] & \text{the correlation vector of } d(n) \text{ and } \mathbf{u}(n) \end{cases}$$

We also used the fact that the dot product between two vectors is scalar and therefore equal to its transpose: e.g. $\mathbf{w}^T\mathbf{u}(n) = \mathbf{u}(n)^T\mathbf{w}$.

From formula (13.7), it can be checked that the MSE can also be put into the form of a perfect square, as

$$J_{\text{mse}}(\mathbf{w},n) = (\mathbf{w} - \overset{\triangle}{\mathbf{w}})^T\mathbf{R}_{uu}(\mathbf{w} - \overset{\triangle}{\mathbf{w}}) - \overset{\triangle}{\mathbf{w}}{}^T\mathbf{R}_{uu}\overset{\triangle}{\mathbf{w}} + \sigma_d^2 \tag{13.8}$$

if

$$\boxed{\overset{\triangle}{\mathbf{w}} : \mathbf{R}_{uu}\overset{\triangle}{\mathbf{w}} = \mathbf{R}_{du}} \tag{13.9}$$

Since the quadratic form in (13.8) is always nonnegative, we see that the MSE is minimum if and only if

$$\mathbf{w} = \overset{\triangle}{\mathbf{w}} = \mathbf{R}_{uu}^{-1}\mathbf{R}_{du}, \tag{13.10}$$

assuming that $\mathbf{R}_{uu}$ is invertible. The minimum error is then given by

$$J_{\text{mse}}(\overset{\triangle}{\mathbf{w}}, n) = \sigma_d^2 - \overset{\triangle}{\mathbf{w}}^T \mathbf{R}_{du} \tag{13.11}$$

Alternatively, the minimum can also be found by equating the derivative of the criterion to zero. Indeed, this derivative is

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{w}} J_{\text{mse}}(\mathbf{w}, n) = \frac{\mathrm{d}\mathbb{E}\left[e(n)^2\right]}{\mathrm{d}\mathbf{w}} = 2\mathbb{E}\left[\frac{\mathrm{d}e(n)}{\mathrm{d}\mathbf{w}} e(n)\right].$$

Since $e(n) = \mathbf{w}^T\mathbf{u}(n) - d(n)$, its derivative with respect to $\mathbf{w}$ is $\mathbf{u}(n)$, and it remains

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\mathbf{w}} J_{\text{mse}}(\mathbf{w}, n) &= 2\mathbb{E}\left[\mathbf{u}(n)e(n)\right] && (13.12)\\
&= 2\mathbb{E}\left[\mathbf{u}(n)\left(\mathbf{u}(n)^T\mathbf{w} - d(n)\right)\right] && (13.13)\\
&= 2\left(\mathbf{R}_{uu}\mathbf{w} - \mathbf{R}_{du}\right). && (13.14)
\end{aligned}$$

Hence, the derivative is zero if and only if $R{uu} w= R{du} $ which is the solution (13.10).

Interestingly, we see that the optimum estimator depends only on the second order properties of the desired response and the input sequence. This is a consequence of our choice of restricting ourselves to a quadratic criterion and a linear transform.

13.1.3 The Least Square Error Estimator

The derivation of the least-squares estimator closely follows the steps we used for the MMSE estimator. This follows easily once the problem is formulated in matrix form. Define the error vector as $\mathbf{e}(n_0, n_1)^T = [e(n_0), e(n_0+1), \ldots e(n_1)]$. Each component of the error, say $e(k)$ is equal to

$$e(k) = \mathbf{u}(k)^T\mathbf{w} - d(k).$$

Therefore, we have

$$\begin{bmatrix} u(n_0) & u(n_0-1) & \ldots & u(n_0-p+1) \\ u(n_0+1) & u(n_0) & \ldots & u(n_0-p+2) \\ \vdots & \ddots & & \vdots \\ u(n_1) & u(n_1-1) & \ldots & u(n_1-p+1) \end{bmatrix} \mathbf{w} - \begin{bmatrix} d(n_0) \\ d(n_0+1) \\ \vdots \\ d(n_1) \end{bmatrix} = \begin{bmatrix} e(n_0) \\ e(n_0+1) \\ \vdots \\ e(n_1) \end{bmatrix}. \tag{13.15}$$

This can also be written in compact form as

$$\mathbf{U}(n_0, n_1)\mathbf{w} - \mathbf{d}(n_0, n_1) = \mathbf{e}(n_0, n_1).$$

Then, the LS criterion (13.3) can be reformulated as

$$J_{\text{ls}}(w, n_0, n_1) = \sum_{n=n_0}^{n_1} e(n)^2 = \mathbf{e}(n_0, n_1)^T\mathbf{e}(n_0, n_1)$$

that is

$$J_{\text{ls}}(w, n_0, n_1) = \left(\mathbf{U}(n_0, n_1)\mathbf{w} - \mathbf{d}(n_0, n_1)\right)^T \left(\mathbf{U}(n_0, n_1)\mathbf{w} - \mathbf{d}(n_0, n_1)\right).$$

Now, it is a simple task to compute the derivative of this LS criterion with respect to $\mathbf{w}$. One readily obtain

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{w}} J_{\mathrm{ls}}(w, n_0, n_1) = 2\mathbf{U}(n_0, n_1)^T \left(\mathbf{U}(n_0, n_1)\mathbf{w} - \mathbf{d}(n_0, n_1)\right),$$

which is equal to zero if and only if

$$\mathbf{U}(n_0, n_1)^T \mathbf{U}(n_0, n_1)\mathbf{w} = \mathbf{U}(n_0, n_1)^T \mathbf{d}(n_0, n_1). \tag{13.16}$$

The different matrices and vectors above depend on two indexes n_0 and n_1. It is now time to discuss the meaning of these indexes and the possible choices for their values. Suppose that the data are available on N samples, from $n = 0$ to $n = N-1$. When we want to compute the error $e(k)$, with $k < p$, we see that the result depend on unobserved values. The same kind of problem occurs if we want to compute the error for $k > N-1$. Therefore we face the problem of affecting a value to unobserved values. A possibility is to take a value of zero for unobserved vales. Another possibility consists in affecting the values by periodization, modulo N, of the available data. A last possibility is to avoid the situations which request the use of unknown values.

The two main choices are the following:

- If we want to use only known values, it suffices to restrict the summation interval to the interval with $n_0 = p-1$ and $n_1 = N-1$. The matrix $\mathbf{U}$ has dimensions $(N-p) \times p$.This choice is sometimes known as the `covariance form`.
- If we choose $n_0 = 0$ and $n_1 = N-p-2$, with unknown values taken as zero, the corresponding choice is called `correlation form`. The data matrix has now dimensions $N+p-1 \times p$.

It is now easy to see that the generic term of $\left[\mathbf{U}(n_0, n_1)^T \mathbf{U}(n_0, n_1)\right]_{ij}$ has the form $\sum_n u(n-i)u(n-j)$, that is, is (up to a factor) an estimate of the correlation $R_{uu}(i-j)$. Consequently, we have an estimate of the correlation matrix $\mathbf{R}_{uu}$ given by

$$\hat{\mathbf{R}}_{uu} = \left[\mathbf{U}(n_0, n_1)^T \mathbf{U}(n_0, n_1)\right].$$

In the case of the choice of the correlation form for the data matrix, the resulting estimate of the correlation matrix has Toeplitz symmetry. It is interesting to note that by construction, the estimated correlation matrix is automatically non-negative definite. Similarly, $\mathbf{R}_{du}$ can be estimated as

$$\hat{\mathbf{R}}_{du} = \mathbf{U}(n_0, n_1)^T \mathbf{d}(n_0, n_1).$$

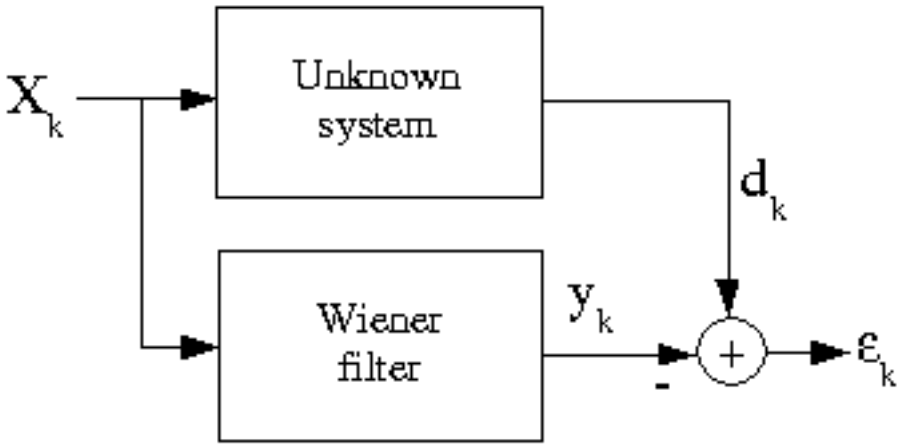
Finally, the LS estimate is

$$\overset{\triangle}{\mathbf{w}}_{ls} = \left[\mathbf{U}(n_0, n_1)^T \mathbf{U}(n_0, n_1)\right]^{-1} \mathbf{U}(n_0, n_1)^T \mathbf{d}(n_0, n_1) = \hat{\mathbf{R}}_{uu}^{-1} \hat{\mathbf{R}}_{du}. \tag{13.17}$$

13.1.4 Application to filter identification

We will apply these results to the problem of filter identification. Let us briefly state the problem: we observe the noisy output y of an unknown system with impulse response h_{test} and a known input x. The goal is to identify h_{test} given y and x.

This figure is taken from cnx.org



We begin by simulating the problem. You may use the function `lfilter` to compute the output of the system. Take for x a gaussian noise, `np.random.normal` or `np.random.randn`, with unit variance on N points, and add a gaussian noise with scale factor 0.1 on the output.

```
# DO IT YOURSELF!
#
from scipy.signal import lfilter
N=0  # update this
x=0  # update this
htest=10*np.array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
y0=0 #ăFILL IN SOMETHING CORRECT HERE
y=0 #ăFILL IN SOMETHING CORRECT HERE
#y0 = #noiseless output
#y= #noisy output

plt.plot(y)
plt.xlabel("Time")
plt.title("Observation")
figcaption("System output in an identification problem")
```

```
from scipy.signal import lfilter
# test
N=200
x=np.random.randn(N)
htest=10*np.array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
#L=size(htest)
#yo=zeros(N)
#for t in range(L,200):
#    yo[t]=htest.dot(x[t:t-L:-1])
#y=yo+ 0.1*randn(N)
y=lfilter(htest,[1],x)+0.1*randn(N)
plt.plot(y)
plt.xlabel("Time")
plt.title("Observation")
figcaption("System output in an identification problem")
```

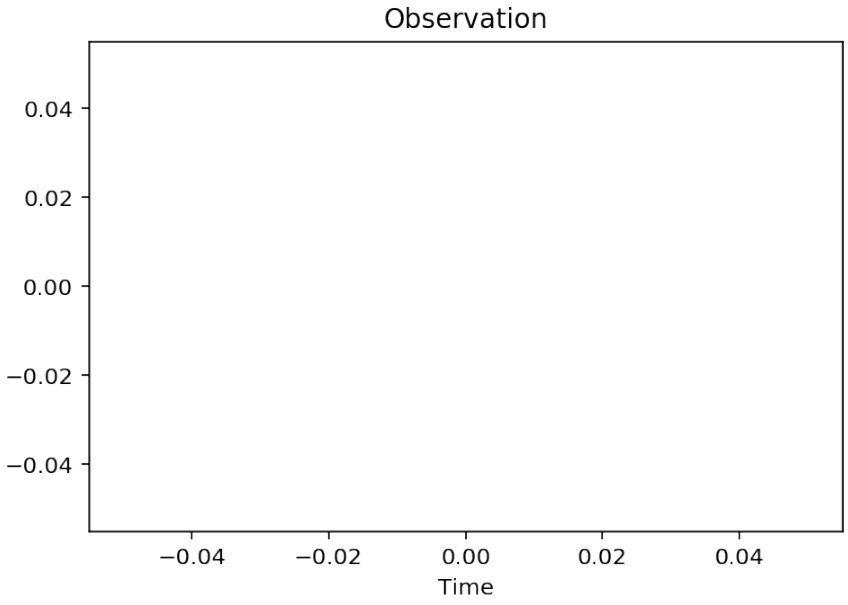


Figure 13.1: System output in an identification problem

```
/usr/local/lib/python3.5/site-packages/scipy/signal/signaltools.py:1344: FutureWarn
  out = out_full[ind]
```

Once this is done, we shall solve the normal equation (13.9). Of course, we firts need to estimate the correlation matrix $\mathbf{R}_{uu}$ and the correlation vector $\mathbf{R}_{du}$. This can be done with the functions `xcorr` and `toeplitz`. Beware on the fact that `xcorr` returns two vectors and that the returned correlation vector is the symmetric sequence with positive and negative indexes.

> Now, in order to implement the identification procedure, one has to put the problem as a Wiener problem and identify the input sequence u and the desired one d. Actually, here one should simply observe that we look for a filter, which excited by the same $x(n)$ should yield an output $z(n)$ as similar as $y_0(n)$ as possible. So, what would you take for u and d?

One thus take `u=x`, and `d=y`(the wanted sequence is $y_0(n)$, which shall be substituted by $y(n)$ – since y_0 is unknown).

> We now have to implement the estimation of correlations and then compute the solution to the normal equation. We note $q+1$ the size of the filter (then of the correlation vector and matrix). The inverse of a matrix can be obtained using the function `inv` in the module `np.linalg`. The matrix mutiplication can be done using the `.dot()` method. Finally, you may evaluate the performance by displaying the identified coefficients and by computing the MMSE according to (13.11).

```
# DO IT YOURSELF!

from correlation import xcorr
```

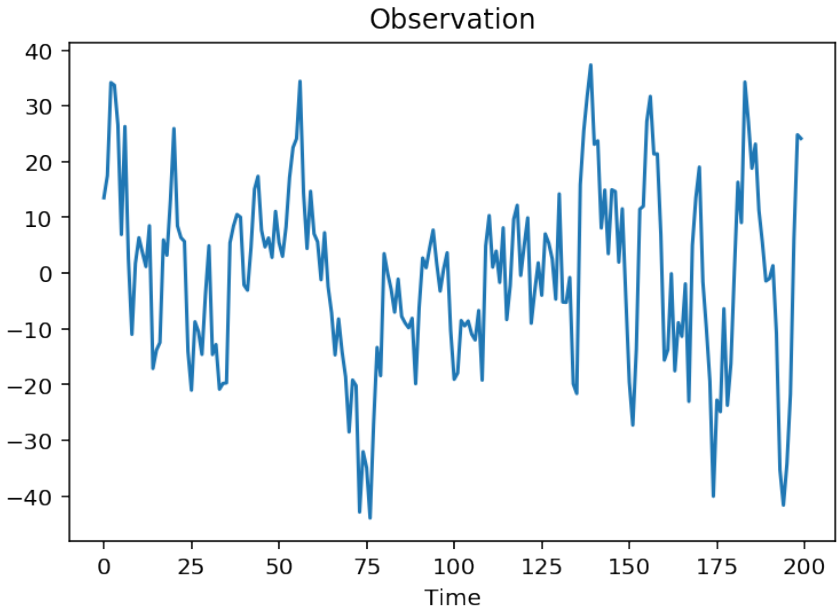


Figure 13.2: System output in an identification problem

```
from scipy.linalg import toeplitz
from numpy.linalg import inv
q=5
z=np.zeros(q+1)

u=z #update this
d=z #update this
c=z #update this #correlation vector
Ruu=np.outer(z,z) #update this
Rdu=z #update this

w=z #update this
print("Estimated filter", w)
print("True filter", htest)
# Minimum error
sigma2d=mean(d**2)
mmse=sigma2d-w.dot(Rdu)
print("MMSE: ",mmse)
```

```
Estimated filter [0. 0. 0. 0. 0. 0.]
True filter [10.  7.  7.  7.  3.  0.]
MMSE:  0.0
```

```
from correlation import xcorr
from scipy.linalg import toeplitz
from numpy.linalg import inv
q=5

u=x
```

```
d=y
c=xcorr(u,u,maxlags=q)[0][q::] #correlation vector
Ruu=toeplitz(c)
Rdu=xcorr(d,u,maxlags=q)[0][q::]
w=inv(Ruu).dot(Rdu)
print("Estimated filter", w)
print("True filter", htest)
# Minimum error
sigma2d=mean(d**2)
mmse=sigma2d-w.dot(Rdu)
print("MMSE: ",mmse)
```

```
Estimated filter [ 9.98339589  7.00502002  6.92998889  6.84380767  2.8490178  -0.12
True filter [10.  7.  7.  7.  3.  0.]
MMSE:  2.089640763511909
```

Finally, it is interesting to transform the lines above in order to plot the MMSE error as a function of q.

```
from correlation import xcorr
from scipy.linalg import toeplitz
from numpy.linalg import inv

u=x
d=y
qmax=18 # max value for q
mmse=np.zeros(qmax) # initialize the vector of errors
for q in range(0,qmax):
    c=xcorr(u,u,maxlags=q)[0][q::] #correlation vector
    Ruu=toeplitz(c)
    Rdu=xcorr(d,u,maxlags=q)[0][q::]
    w=inv(Ruu).dot(Rdu)
    # Minimum error
    sigma2d=mean(d**2)
    mmse[q]=sigma2d-w.dot(Rdu)
print("MMSE: ", mmse)
plt.plot(range(0,qmax),mmse)
plt.xlabel("Order of the filter")
plt.ylabel("MMSE")
plt.title("MMSE as a function of the length of the identification filter")
figcaption("MMSE as a function of the length of the identification filter")
```

```
MMSE:  [162.1082212  113.44267243  60.84982367  10.40603336   2.10644923
   2.08964076   2.07972667   1.95845393   1.89574241   1.89198481
   1.89091256   1.89084977   1.89082315   1.88830182   1.8820792
   1.87776324   1.87627425   1.84640675]
```

The evolution of the MMSE with respect to q shows that the MMSE is important while the length of the identification filter is underestimated. The MMSE falls to a "floor" when the length is equal to or higher than the true value. This offers an easy way to detect an "optimal" order for the identification.

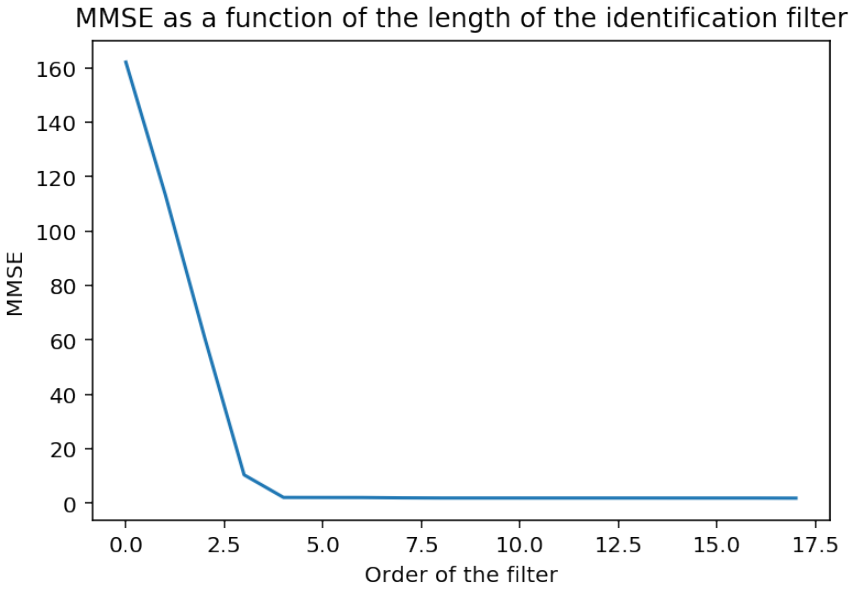


Figure 13.3: MMSE as a function of the length of the identification filter

Remark 1. *Actually, the identification error always decreases when one increases the length of the filter, that is add degrees of freedom to perform the identification. Usually, increasing the number of parameters decreases the statistical stability of the estimate, and one has to made a trade-off between a sufficient number of parameters to avoid a bias ans a low number of parameter to lower the variance of the estimate. This is the notion of bias-variance trade-off that appears in many areas of statistical signal processing. Thus, for choosing an "optimal" order, one usually use a composite criterion where the first term is the MMSE, decreasing with the order, and a second term which increases with the order, thus penalizing high orders.*

13.2 The steepest descent algorithm

Although direct inversion provide the solution in a finite number of steps, it is sometimes preferable to use alternative iterative methods because they may require less numerical precision, are usually computationally and can even be applied in the case of non-quadratic criteria. The adaptive filtering algorithms we will see later will be obtained by simple modifications of iterative methods. Before indicating how such methods can be useful for solving our normal equations, we begin by describing the Steepest Descent Algorithm (SDA).

Let $f(\mathbf{x})$ be a differentiable function of $\mathbf{x}$ with continuous derivatives. It is then possible to approximate the function at a point $\mathbf{x}+\Delta\mathbf{x}$ using the Taylor expansion

$$f(\mathbf{x}+\Delta\mathbf{x}) = f(\mathbf{x}) + \Delta\mathbf{x}^T\nabla f(\mathbf{x}) + \frac{1}{2}\Delta\mathbf{x}^T\nabla^2 f(\mathbf{x})\Delta\mathbf{x} + \ldots,$$

where $\nabla f(\mathbf{x})$ denotes the gradient of f at $\mathbf{x}$ and $\nabla^2 f(\mathbf{x})$ the Hessian. Restricting ourselves to the first order approximation, we see that if we choose $\Delta\mathbf{x}^T\nabla f(\mathbf{x}) < 0$, then $f(\mathbf{x}+\Delta\mathbf{x}) < f(\mathbf{x})$, i.e. f decreases. The higher $|\Delta\mathbf{x}^T\nabla f(\mathbf{x})|$, the most important the decrease. The scalar product is maximum when the two vectors are colinear, and they must have opposite direction so as to obtain a negative scalar product. This yields

$$\Delta\mathbf{x} = -\nabla f(\mathbf{x}).$$

The negative of the gradient is known as the direction of steepest descent. Usually, to keep $\Delta\mathbf{x}$ small enough for the validity of the Taylor approximation, one uses a small positive factor μ in front of the gradient. This leads to the following iterative algorithm

$$\boxed{\mathbf{x}_{k+1} = \mathbf{x}_k - \mu\nabla f(\mathbf{x})}, \tag{13.18}$$

which is known as the *steepest descent algorithm*. We begin with an initial guess $\mathbf{x}_0$ of the solution and take the gradient of the function at that point. Then we update the solution in the negative direction of the gradient and we repeat the process until the algorithm eventually converges where the gradient is zero. Of course, this works if the function at hands possesses a true minimum, and even in that case, the solution may correspond to a local minimum. In addition, the value of the step-size μ can be crucial for the actual convergence and the speed of convergence to a minimum.

We give below a simple implementation of a steepest descent algorithm. Beyond formula (13.18), we have refined by

- specifying a stopping rule: error less than a given precision `err` or number of iteration greater than a maximum number of iterations `itermax`
- a line-search procedure `line_search` (True by default) which adapts the step-size in order to ensure that the objective function actually decreases
- a verbose mode `verbose` (True by default) which prints some intermediary results.

Some references available online:

- Gradient descent
- Gradient desxcent (2)
- Conjugate gradients

```
def grad_algo(f, g, mu, x0=0, eps=0.001, grad_prec=0.0001, itermax=200,
    line_search=True, verbose=True):

    def update_grad(xk,mu):
        return xk-mu*g(xk)

    xk=np.zeros((np.size(x0),itermax))
    xk[:,0]=x0
    err=1
    k=0
    while err>eps and k<itermax-1:
        err=norm(xk[:,k]-update_grad(xk[:,k],mu),1)
        xk[:,k+1]=update_grad(xk[:,k],mu)
        if (np.any(np.isnan(xk[:,k+1])) or np.any(np.isinf(xk[:,k+1])) ):
            break
```

```
        m=0
        #line search: look for a step that ensures that the objective
            function decreases
        if line_search:
            while f(xk[:,k+1])>f(xk[:,k]):
                #print("Updating..", f(xk[k+1]),f(xk[k]))
                m=m+1
                xk[:,k+1]=update_grad(xk[:,k],mu*(0.5)**m)
        # avoid to stay stalled
        if norm(g(xk[:,k])+g(xk[:,k-1]),1)<grad_prec:
            #print("gradients..", g(xk[k+1]),g(xk[k]))
            mu=mu*0.99
            xk[:,k+1]=update_grad(xk[:,k],mu)
        if verbose:
            if np.size(x0)==1:
               print("current solution {:2.2f}, error: {:2.2e}, gradient
                   {:2.2e}, objective {:2.2f}".format(xk[0,k+1],err,g(xk[0,k
                   +1]),f(xk[0,k+1])))
            else:
                print("error: {:2.2e}, gradient {:2.2e}, objective {:2.2f}".
                    format(err,norm(g(xk[:,k+1]),2),f(xk[:,k+1])))
               #pass
        k=k+1
    return xk[:,:k]
```

Let us illustrate the SDA in the case of an bivariate quadratic function. You may experiment by modifying the initial guess and the step-size μ.

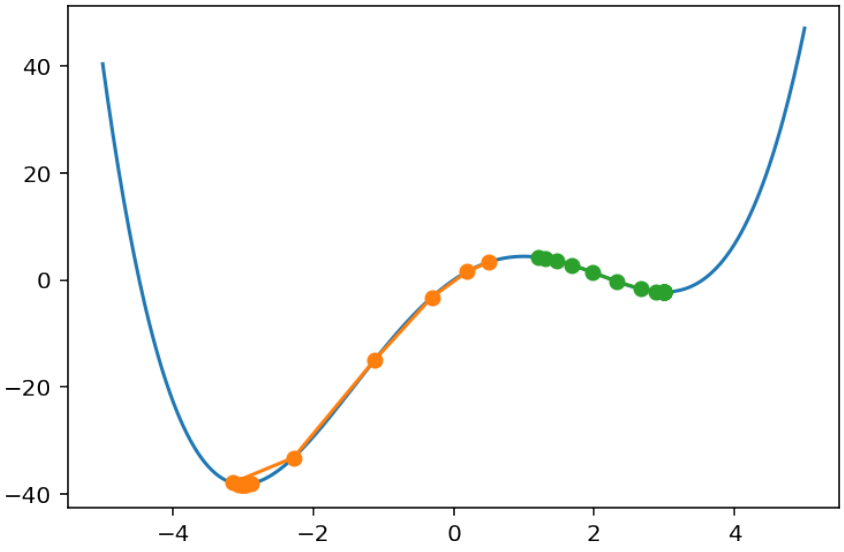
```
def f(x):  #objective function
    return np.sum(x**2) #
def ff(x):
    return np.array([f(xx) for xx in x])

def g(x): #gradient
    return 2*x #

#Test # ------------------------------------------------------------------

def tst(ini0, ini1, mu):
    eps=0.001
    xk=grad_algo(f, g, mu=mu, x0=[ini0, ini1], eps=0.001, grad_prec=0.0001,
        itermax=200, line_search=False, verbose=False)
    #xk=grad_algo(f, g, mu=0.05, x0=0.5, eps=0.001, grad_prec=eps/10,
        itermax=200, line_search=False, verbose=True)

    clear_output(wait=True)
    x=np.linspace(-5,5,400)
    plt.plot(x,ff(x))
    x=xk[0,:]
    plt.plot(x,ff(x),'o-')
    x=xk[1,:]
    plt.plot(x,ff(x),'o-')

def tsto(val):
    tst(x0.value, x1.value, mu.value)

x0=widgets.FloatText(value=3.5)
x1=widgets.FloatText(value=-4.2)
mu=widgets.FloatSlider(min=0, max=1.4, step=0.01, value=0.85)
```

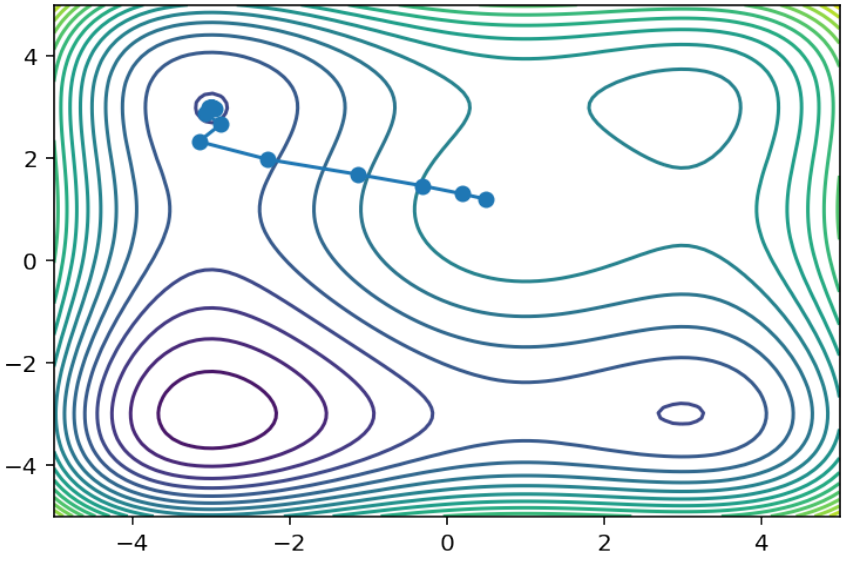
```
#c=widgets.ContainerWidget(children=(ini0, ini1))
x0.observe(tsto,names=["value"])
x1.observe(tsto,names=["value"])
mu.observe(tsto,names=["value"])
display(widgets.VBox([x0, x1, mu]))
#_=interact(tst, ini0=x0, ini1=x1, mu=mu    )
```

```
Widget Javascript not detected.  It may not be installed or enabled properly.
```

It is also instructive to look at what happens in the case of a non-quadratic function.

```
def f(x):
    return np.sum(x**4/4 - x**3/3 - 9*x**2/2 + 9*x) #np.sum(x**2) #
def ff(x):
    return np.array([f(xx) for xx in x])

def g(x):
    return((x-1)*(x+3)*(x-3)) # 2*x #

#Test # ------------------------------------------------------------------

def tst(ini0, ini1, mu):
    eps=0.001
    xk=grad_algo(f, g, mu=mu, x0=[ini0, ini1], eps=0.001, grad_prec=0.0001,
        itermax=200, line_search=False, verbose=False)
    #xk=grad_algo(f, g, mu=0.05, x0=0.5, eps=0.001, grad_prec=eps/10,
        itermax=200, line_search=False, verbose=True)


    x=np.linspace(-5,5,400)
    plt.plot(x,ff(x))
    x=xk[0,:]
    plt.plot(x,ff(x),'o-')
    x=xk[1,:]
    plt.plot(x,ff(x),'o-')
    plt.figure()


    x=np.linspace(-5,5,100)
    xx,yy=meshgrid(x,x)
    z=np.zeros((len(xx),len(yy)))
    for m,a in enumerate(x):
        for n,b in enumerate(x):
            z[n,m]=f(array([a,b]))
            #print(m,n,a,b,z[m,n])
    #z=[[f(array([a,b])) for a in xx] for b in yy]
    h = plt.contour(x,x,z,20)
    plt.plot(xk[0,:],xk[1,:], 'o-')

x0=widgets.FloatText(value=0.5)    # or -1.5
x1=widgets.FloatText(value=1.2)    # 0.8
mu=widgets.FloatSlider(min=0, max=1.4, step=0.01, value=0.07)
#c=widgets.ContainerWidget(children=(ini0, ini1))
_=interact(tst, ini0=x0, ini1=x1, mu=mu    )
```

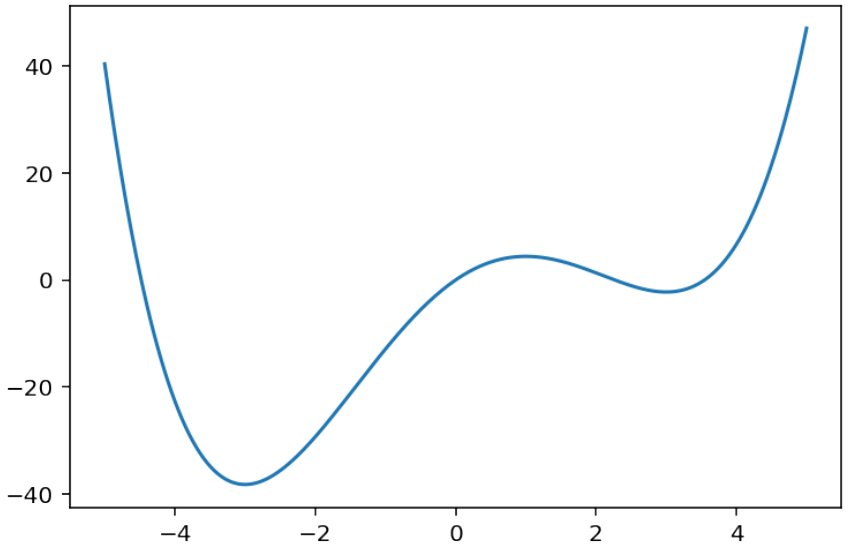

40
20
0
−20
−40
−4
−2
0
2
4


4
2
0
−2
−4
−4
−2
0
2
4

13.3 Application to the iterative resolution of the normal equations

```
x=np.linspace(-5,5,400)
y=(x-1)*(x+3)*(x-3) #
y=x**4/4 - x**3/3 - 9*x**2/2 + 9*x
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fd4ec5a77b8>]
```




```
import sympy
x=sympy.symbols('x')
e=sympy.expand((x-1)*(x+3)*(x-3))
print(e)
sympy.integrate(e)
sympy.plot(e)
```

```
x**3 - x**2 - 9*x + 9
```

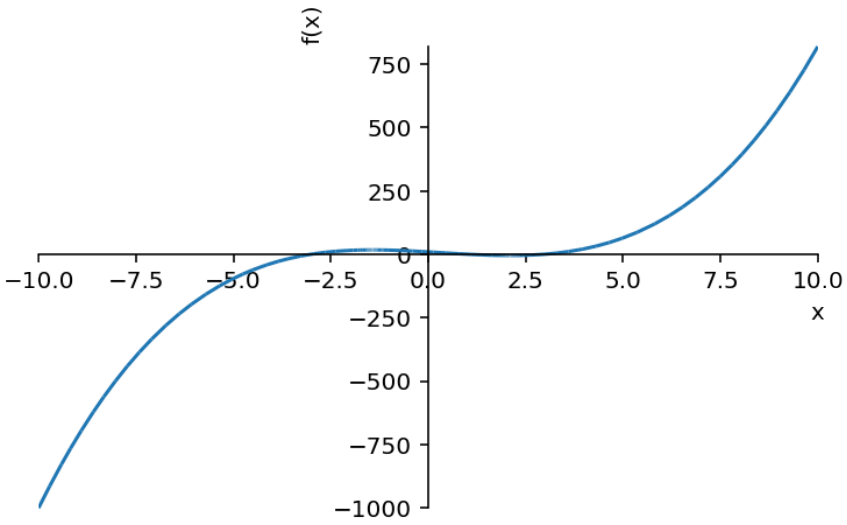
```
<sympy.plotting.plot.Plot at 0x7fd4e9efb3c8>
```

Definitions

Recall that the MMSE criterion

$$J_{\text{mse}}(w) = \mathbb{E}\left[e(n)^2\right] \tag{13.19}$$

$$= \mathbf{w}^T \mathbf{R}_{uu} \mathbf{w} - 2\mathbf{w}^T \mathbf{R}_{du} + \sigma_d^2 \tag{13.20}$$



has for gradient vector

$$\nabla_{\mathbf{w}} J_{\text{mse}}(\mathbf{w}) = 2\mathbb{E}\left[\mathbf{u}(n)e(n)\right] \tag{13.21}$$
$$= 2\left(\mathbf{R}_{uu}\mathbf{w} - \mathbf{R}_{du}\right). \tag{13.22}$$

The derivative is zero if and only if $R{uu} w= R{du} $ which is the normal equation.

Instead of directly solving the normal equation by taking the inverse of $\mathbf{R}_{uu}$, we can also minimize the original criterion using a SDA algorithm. Since the MMSE criterion is a quadratic form in $\mathbf{w}$, it has an only minimum $\overset{\triangle}{\mathbf{w}}$ which will be reached regardless of the initial condition.

Beginning with the general formulation (13.18) of the SDA, and using the expression of the gradient of the MMSE, we readily obtain

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) - \mu\mathbb{E}\left[\mathbf{u}(n)e(n)\right] \\ &= \mathbf{w}(n) - \mu\left(\mathbf{R}_{uu}\mathbf{w}(n) - \mathbf{R}_{du}\right)\end{aligned}$$

(we absorbed the factor 2 in the gradient into the constant μ). It is important to stress that here, the index n represents the iterations of the algorithm, *and has nothing to do with time*. [1]

Even before studying convergence properties of the resulting algorithm, let us examine its behavior in the very same example of filter identification we used in section 13.1.4

```
np.random.seed(749)
from scipy.signal import lfilter
# test
N=800
x=lfilter([1, 1], [1], np.random.randn(N))
htest=10*np.array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
y=lfilter(htest,[1],x)+0.1*randn(N)
plt.plot(y)
plt.xlabel("Time")
plt.title("Observation")
figcaption("System output in an identification problem")
```

[1] Actually the normal equation we are solving is independent of time – it is only in the non-stationary case that normal equation depends on time; in such a case, the SDA would depend on both iterations and time.

```
/usr/local/lib/python3.5/site-packages/scipy/signal/signaltools.py:1344: FutureWarn
  out = out_full[ind]
```

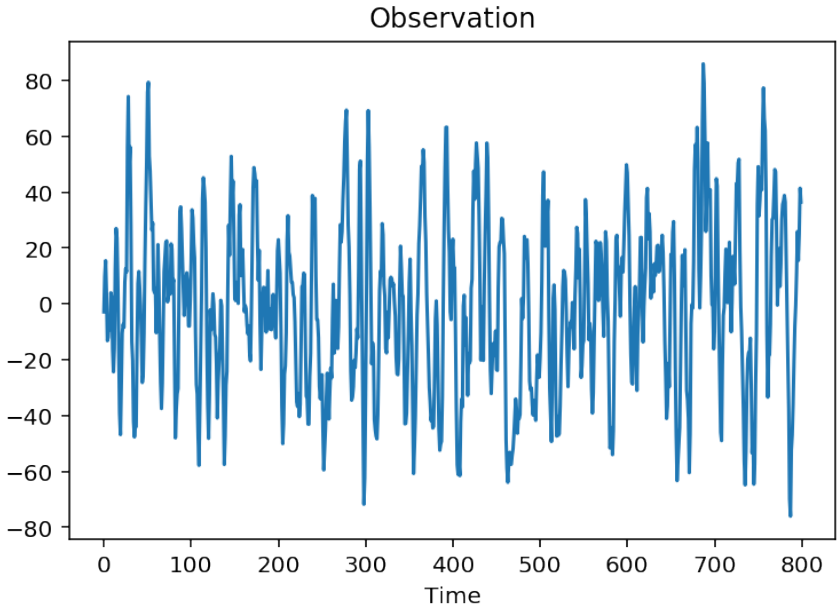


Figure 13.4: System output in an identification problem

Implement a function that iterates the SDA, beginning with an initial condition `winit` until the (norm of the) increment between two successive updates is less than a given precision `eps` (use a `while` loop). The syntax of the function should be

```
sda(Ruu,Rdu, winit, mu, eps)
```

It will return the optimum vector w and the number of iterations.

```
# DO IT YOURSELF!
#def sda(Ruu,Rdu, winit, mu=0.05, eps=0.001):
#    itermax=2000
#    err=100
#    k=0
#    w=winit
#    while ...
#
#    return w,niter
```

```
def sda(Ruu,Rdu, winit, mu=0.05, eps=0.001, verbose=False):
    itermax=2000
    err=(100, 100)
    k=0
    w=winit
    while np.linalg.norm(err,2)>eps and k<itermax-1:
        err=(Ruu.dot(w)-Rdu)
        w=w-mu*err
```

```
        k+=1
        if verbose: print("Iteration {0:d}, error: {1:2.2e}".format(k,np.
            linalg.norm(err,2)))
    return w, k
```

As an example, we test the implementation with $\mu = 0.05$:

```
from correlation import xcorr
from scipy.linalg import toeplitz
from numpy.linalg import inv
mu=0.05
(u, d, q)=(x, y, 6)
c=xcorr(u,u,maxlags=q)[0][q::] #correlation vector
Ruu=toeplitz(c)
Rdu=xcorr(d,u,maxlags=q)[0][q::]
```

```
w, nbiter=sda(Ruu,Rdu, winit=np.zeros(q+1), mu=0.05, eps=0.001, verbose=
    False)
print("for mu={0:1.3f}, number of iterations: {1:}".format(mu,nbiter))
print("Identified filter", w)
print("True filter", htest)
```

```
for mu=0.050, number of iterations: 567
Identified filter [10.00337544  6.99712839  6.96204905  6.93485836  3.00723524 -0.0
 -0.02476235]
True filter [10.  7.  7.  7.  3.  0.]
```

We can also study the behavior and performance of the SDA as a function of the step-size μ.

```
from correlation import xcorr
from scipy.linalg import toeplitz
from numpy.linalg import inv

u=x
d=y
q=6
c=xcorr(u,u,maxlags=q)[0][q::] #correlation vector
Ruu=toeplitz(c)
Rdu=xcorr(d,u,maxlags=q)[0][q::]
wopt=inv(Ruu).dot(Rdu)
k=0
mu_iter=np.arange(0,0.51,0.01)
niter=np.empty(np.shape(mu_iter))

for mu in mu_iter:
    w, nbiter=sda(Ruu,Rdu, winit=np.zeros(q+1), mu=mu, eps=0.001, verbose=
        False)
    niter[k]=nbiter
    k+=1
    #print("for mu={0:1.3f}, number of iterations: {1:}".format(mu,nbiter))
print("Last identified filter", w)
print("true filter", htest)
plt.plot(mu_iter,niter)
plt.xlabel("$\mu$")
plt.ylabel("Number of iterations")
figcaption("Number of iterations of the gradient algorithm as a function of
    $\mu$",
           label="fig:itergrad")
```

```
Last identified filter [10.00463266  6.9950947   6.96491799  6.93197326  3.01010416
 -0.02350515]
true filter [10.  7.  7.  7.  3.  0.]
```

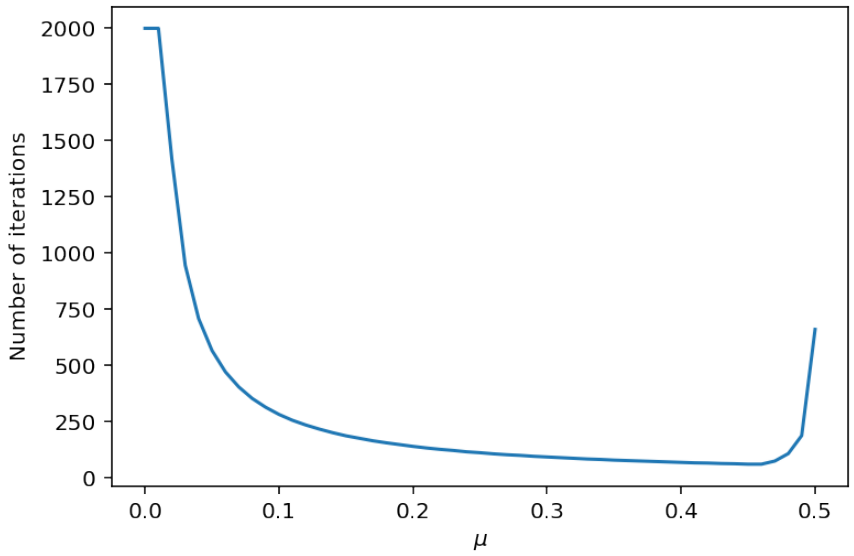


Figure 13.5: Number of iterations of the gradient algorithm as a function of μ

We observe that the number of iterations needed to obtain the convergence (up to a given precision) essentially decreases with μ, up to a minimum. After this minimum, the number of iterations shortly increases, up to a value of μ where the algorithm begins to diverge.

13.3.1 Convergence analysis

The choice of the step-size is crucial. If the steps are too large, then the algorithm may diverge. If they are too small, then convergence may take a long time

Conditions on the step-size

Let $\mathbf{v}(n) = \mathbf{w}(n) - \overset{\triangle}{\mathbf{w}}$ denote the error between the filter at step n and the optimum filter $\overset{\triangle}{\mathbf{w}}$. Subtracting $\overset{\triangle}{\mathbf{w}}$ from both sides of the SDA

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu\left(\mathbf{R}_{uu}\mathbf{w}(n) - \mathbf{R}_{du}\right)$$

we get

$$\mathbf{v}(n+1) = \mathbf{v}(n) - \mu\left(\mathbf{R}_{uu}\mathbf{w}(n) - \mathbf{R}_{du}\right).$$

Using the fact that $\mathbf{R}_{du} = \mathbf{R}_{uu}\overset{\triangle}{\mathbf{w}}(n)$, it comes

$$\mathbf{v}(n+1) = \mathbf{v}(n) - \mu\left(\mathbf{R}_{uu}\mathbf{w}(n) - \mathbf{R}_{uu}\overset{\triangle}{\mathbf{w}}(n)\right) \tag{13.23}$$

$$= \mathbf{v}(n) - \mu\mathbf{R}_{uu}\mathbf{v}(n) \tag{13.24}$$

$$= \left(\mathbf{I} - \mu\mathbf{R}_{uu}\right)\mathbf{v}(n). \tag{13.25}$$

It is then immediate to express the error at iteration $n+1$ in terms of the initial error $\mathbf{v}(0)$:

$$\mathbf{v}(n+1) = (\mathbf{I} - \mu \mathbf{R}_{uu})^{n+1} \mathbf{v}(0).$$

Clearly, if the algorithm converges, the error shall tends to zero and so doing forget the initial conditions. Here, the error decreases to zero if $(\mathbf{I} - \mathbf{R}_{uu})^{n+1}$ tends to the null matrix. This happens if all the eigenvalues of $(\mathbf{I} - \mathbf{R}_{uu})$ have a modulus inferior to 1. To see this, let us introduce the eigen-decomposition of $\mathbf{R}_{uu}$:

$$\mathbf{R}_{uu} = \mathbf{V} \blacksquare \mathbf{V}^H$$

where $\mathbf{V}$ is the matrix of right eigenvectors of $\mathbf{R}_{uu}$, and $\blacksquare$ the corresponding diagonal matrix of eigenvalues. The superscript ^{H} indicates the conjugate transposition (that is transposition plus conjugation). In the case of a correlation matrix, the eigenvalues are all non-negative, and the eigenvectors can be chosen normed and orthogonal to each other. In other terms, $\mathbf{V}$ is unitary:

$$\mathbf{V}\mathbf{V}^H = \mathbf{I} \text{ or } \mathbf{V}^{-1} = \mathbf{V}^H.$$

Therefore, $(\mathbf{I} - \mu \mathbf{R}_{uu})$ can be put under the form $\mathbf{V}(\mathbf{I} - \mu \blacksquare)\mathbf{V}^H$. This shows that the eigenvalues of the matrix have the form $1 - \mu\lambda_i$, where the λ_i are the eigenvalues of the correlation matrix. For the power $(n+1)$ we then obtain

$$(\mathbf{I} - \mu \mathbf{R}_{uu})^{n+1} = \mathbf{V}(\mathbf{I} - \mu \blacksquare)^{n+1} \mathbf{V}^H.$$

Hence we see that this matrix will converge to zero if and only if

$$|1 - \mu\lambda_i| < 1 \quad \forall i.$$

- If $1 - \mu\lambda_i > 0$, this yields $1 - \mu\lambda_i < 1$ and therefore since $\lambda_i \geq 0$ implies $\mu \geq 0$; - If $1 - \mu\lambda_i < 0$, we obtain $\mu\lambda_i - 1 < 1$, so that $\mu < 2/\lambda_i$. Since this must be true for all λ_i, we can only keep the most restrictive inequality: $\mu < 2/\lambda_{\max}$, where $\lambda_{\max}$ denotes the maximum eigenvalue.

Finally, we obtain the following condition

$$\boxed{0 \leq \mu < \frac{2}{\lambda_{\max}}}$$

on the step-size, for ensuring the convergence of the algorithm.
[2]

Optimum step-size

From (13.3.1), we see that the speed of convergence will be limited by slowest eigenvalue, that is by the eigenvalue whose modulus is the nearest to one. Hence, in order to minimize the convergence time, we have to select the maximum of the $1 - \mu\lambda_k$, with respect to k, and minimize that value with respect to μ. Hence we face a minimax problem:

$$\min_{\mu} \max_{k} |1 - \mu\lambda_k|$$

Suppose that there exists a μ_{opt} that realizes the minimum with respect to μ. For $\mu > \mu_{\text{opt}}$, we then have

$$\mu\lambda_k - 1 > \mu_{\text{opt}}\lambda_k - 1 \geq \mu_{\text{opt}}\lambda_{\min} - 1$$

[2] The correlation matrix $\mathbf{R}_{uu}$ is non-negative definite. This means that $\forall \mathbf{v}$, $\mathbf{v}^H \mathbf{R}_{uu} \mathbf{v} \geq 0$. This is easy to check. Indeed, since $\mathbf{R}_{uu} = \mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^H\right]$, we have $\mathbf{v}^H \mathbf{R}_{uu} \mathbf{v} = \mathbb{E}\left[\mathbf{v}^H \mathbf{u}(n)\mathbf{u}(n)^H \mathbf{v}\right] = \mathbb{E}\left[||\mathbf{v}^H \mathbf{u}(n)||^2\right] \geq 0$. Let now $\mathbf{v}$ be any eigenvector of $\mathbf{R}_{uu}$ with eigenvalue λ. In such a case, we have $\mathbf{v}^H \mathbf{R}_{uu} \mathbf{v} = \mathbf{v}^H \lambda \mathbf{v} = \lambda ||\mathbf{v}||^2$. Since we just seen that $\mathbf{v}^H \mathbf{R}_{uu} \mathbf{v} \geq 0$, we deduce that all the eigenvalues λ are non-negative.

On the other hand, for $\mu < \mu_{\text{opt}}$, we have

$$\mu\lambda_k - 1 < \mu_{\text{opt}}\lambda_k - 1$$

or

$$1 - \mu\lambda_k > 1 - \mu_{\text{opt}}\lambda_k \geq 1 - \mu_{\text{opt}}\lambda_{\max}$$

Hence, we obtain that the solution of (13.3.1) is $1 - \mu_{\text{opt}}\lambda_{\max}$ from above and $\mu_{\text{opt}}\lambda_{\min} - 1$ from below. Of course we have, by continuity,

$$1 - \mu_{\text{opt}}\lambda_{\max} = \mu_{\text{opt}}\lambda_{\min} - 1$$

which yields

$$\boxed{\mu_{\text{opt}} = \frac{2}{\lambda_{\max} + \lambda_{\min}}}$$

These two results: convergence condition and optimum step-size completely corresponds to what we observed numerically in figure 13.5. If we compute the eigenvalues of the correlation matrix $\mathbf{R}_{uu}$, we obtain

```
L,V=np.linalg.eig(Ruu)
print("Maximum step-size: ", 2/(np.max(L)))
print("Optimum step-size: ", 2/(np.max(L)+np.min(L)))
```

```
Maximum step-size:  0.504097034235286
Optimum step-size:  0.4840425320713463
```

It is worth recalling that we introduced the iterative algorithm in order to avoid the direct inversion of the correlation matrix, possibly for computational load reasons. However, computing the eigenvalues of the correlation matrix is at least as complicated as computing the inverse. Thus we do not gain anything if we compute the optimum step-size (13.3.1). Fortunately, we can use the following value:

$$\mu = \frac{2}{\text{Tr}\left[\mathbf{R}_{uu}\right]},$$

where Tr denotes the trace operator, that is the sum of the eigenvalues. Sine we know that the trace is also the sum of therms in the main diagonal, and since the matrix is Toeplitz, we also have

$$\mu = \frac{2}{pR_{uu}(0)},$$

where $R_{uu}(0) = \mathbb{E}\left[|u(n)|^2\right]$ and p is the dimension of the correlation matrix.

13.3.2 An alternative view of the Steepest Descent Algorithm

The sum of n terms of a geometric series of matrices

Proposition 1. *The sum of the first n terms of the geometric series B^k, where B is any square matrix*

$$S_n = B^0 + B + B^2 + \ldots + B^k + \ldots + B^n$$

is given by

$$\boxed{S_n = (1-B)^{-1}(1-B^{n+1})}. \tag{13.26}$$

If the spectral radius of B is less than 1, then $\lim_{n\to\infty} B^n = 0$, and

$$\boxed{S_\infty = (1-B)^{-1}}.$$

Proof. Consider the geometric series

$$B^0 + B + B^2 + \ldots + B^k + \ldots$$

where B is any matrix. The sum of the first n terms of this geometric series is given by (1). Of course, we also have

$$BS_n = B + B^2 + \ldots + B^n + B^{n+1} \tag{13.27}$$
$$= S - 1 + B^{n+1}. \tag{13.28}$$

Therefore we have

$$(B-1)S_n = -1 + B^{n+1},$$

and finally the result (13.26) follows after applying the left inverse of $(B-1)$ to both sides. □

Application – This can be applied for instance to the matrix $B = 1 - \mu A$. Here it gives

$$\boxed{\mu S_\infty = \mu \sum_{k=0}^{+\infty} (1-\mu A)^k = A^{-1}}$$

provided that the spectral radius of $(1-\mu A)$ is less than one.

Remark 2. *If B has a spectral radius less than one, then $(1-B)$ is invertible. Consider the eigendecomposition of B as:*

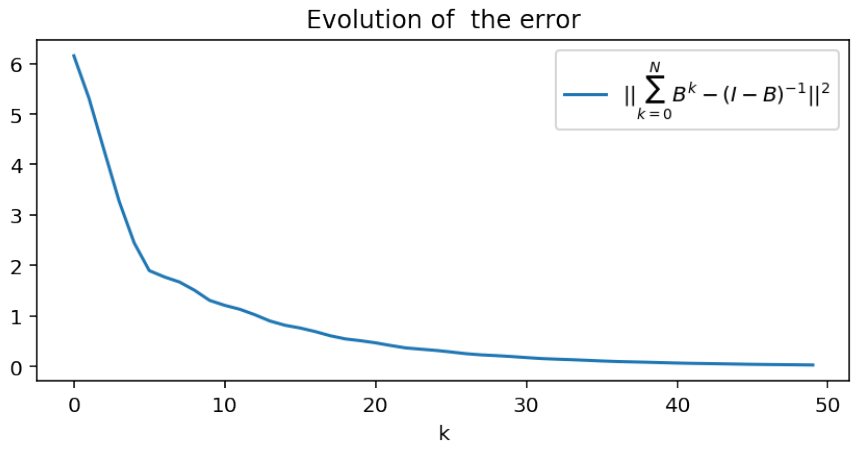
$$B = V \Lambda V^{-1},$$

where V is the matrix of right eigenvectors of B, and Λ the corresponding diagonal matrix of eigenvalues. Then $(1-A) = (VV^{-1} - V\Lambda V^{-1}) = V(1-\Lambda)V^{-1}$. The last relation is noting but a possible eigendecomposition of $(1-B)$. This shows that the corresponding eigenvalues have the form $1-\lambda_i$. If all the eigenvalues have a modulus inferior to 1, then $1-\lambda_i$ is never equal to zero and the matrix $(1-B)$ is invertible.

Let us illustrate numerically that the sum of the geometric series generated by B is indeed $(I-B)^{-1}$

```
# We generate a random matrix B, compute its eigendecomposition and
    normalize by the maximum
# eignevalue. Therefore, the spectal radius is inferior to 1, and the
    property applies
p=50
B=np.random.randn(p,p)
L,V= np.linalg.eig(B)
ll=np.max(np.abs(L))
B=B/(1.1*ll)

# Now we compute the true inverse of (I-B):
I=np.eye(p)
IBi=np.linalg.inv(I-B)

N=50  # number of terms in the sum
err=np.zeros(N) # Evolution of error
S=np.zeros(p)
C=I   # initial C

for k in np.arange(N):
    S=S+C
```

```
    C=C.dot(B)
    err[k]=np.linalg.norm(IBi-S,2)

plt.figure(figsize=(7,3))
plt.plot((abs(err)), label="$||\sum_{k=0}^N B^k - (I-B)^{-1}||^2$")
plt.title("Evolution of  the error")
plt.xlabel("k")
_=plt.legend()
```



An iterative formula for computing the solution of the normal equation

Let us now return to the normal equation

$$\mathbf{w} = \mathbf{R}_{uu}^{-1}\mathbf{R}_{du}.$$

By the property (13.3.2), the inverse of the correlation matrix can be computed as

$$\mathbf{R}_{uu}^{-1} = \mu \sum_{k=0}^{+\infty} (1-\mu\mathbf{R}_{uu})^k.$$

Therefore, if we use a sum of order n, we have

$$\mathbf{w}(n) = \mu \sum_{k=0}^{n} (1-\mu\mathbf{R}_{uu})^k \mathbf{R}_{du}.$$

The term of rank $(n+1)$ can then be expressed as

$$\mathbf{w}(n+1) = \mu \sum_{k=0}^{n+1} (1-\mu\mathbf{R}_{uu})^k \mathbf{R}_{du} \tag{13.29}$$

$$= \mu(1-\mu\mathbf{R}_{uu}) \sum_{k=0}^{n} (1-\mu\mathbf{R}_{uu})^k \mathbf{R}_{du} + \mu(1-\mu\mathbf{R}_{uu})^0 \mathbf{R}_{du} \tag{13.30}$$

$$= (1-\mu\mathbf{R}_{uu})\mathbf{w}(n) + \mu\mathbf{R}_{du} \tag{13.31}$$

$$\tag{13.32}$$

that is also

$$\boxed{\mathbf{w}(n+1) = \mathbf{w}(n) - \mu(\mathbf{R}_{uu}\mathbf{w}(n) - \mathbf{R}_{du})}$$

Hence, we obtain an iterative formula for computing the solution of the normal equation (13.9), without explicitly computing the inverse of the correlation matrix. It is an exact algorithm, which converges to the true solution:

$$\lim_{n\to\infty} \mathbf{w}(n) = \overset{\triangle}{\mathbf{w}} = \mathbf{R}_{uu}^{-1}\mathbf{R}_{du}.$$

As we saw above, this algorithm also appears as a steepest descent algorithm applied to the minimization of the Mean Square Error.

A few references –

[http://nowak.ece.wisc.edu/ece830/ece830_spring13_adaptive_filtering.pdf](htt

[http://www.roma1.infn.it/exp/cuore/pdfnew/ch07.pdf](http://www.roma1.infn.it/exp/c

[http://www.ece.utah.edu/~mathews/ece6550/chapter4.pdf](http://www.ece.utah.edu/~ma

[http://en.wikipedia.org/wiki/Least_mean_squares_filter#Normalised_least_

```
import mpld3
mpld3.enable_notebook()
import warnings
warnings.simplefilter('default')
```

13.4 Adaptive versions

The steepest descent algorithm employs the gradient of the mean square error to search for the Wiener filter coefficients. The drawbacks are that - this relies on the knowledge of the true second-order statistics (correlations), while they are evidently non available; - the resulting filter is not adaptive to a non-stationary environment, since the normal equations have been derived in a stationary context.

In order to take into account those two drawbacks, we need to define `estimates of the correlation functions` able track non-stationarities of signals. With these estimates at hand, we will just have to plug them in the normal equations.

Let us consider the simple example where we have to estimate the power of a non-stationary signal:

$$\sigma(n)^2 = \mathbb{E}\left[X(n)^2\right].$$

A simple solution is to approximate the ensemble average as a time average is some neighborhood of point n:

$$\sigma_L(n)^2 = \frac{1}{2L+1}\sum_{l=-L}^{L} x(n-l)^2.$$

which corresponds to filtering with a sliding (rectangular) window of length $2L+1$. Note that it is possible to compute this recursively as

$$\sigma_L(n)^2 = \sigma_L(n-1)^2 + x(n+L) - x(n-L-1).$$

Another solution is to introduce a forgetting factor λ which enables to give more weight to the more recent samples and forget the older ones. The corresponding formula is

$$\sigma_\lambda(n)^2 = K_n \sum_{l=0}^{n} \lambda^{n-l} x(l)^2,$$

where K_n is a factor which ensures unbiaseness of the estimate, i.e. $\mathbb{E}\left[\sigma_\lambda(n)^2\right] = \sigma(n)^2$. As an exercise, you should check that $K_n = (1-\lambda^{n+1})/(1-\lambda)$. For $\lambda < 1$, K_n converges rapidly and we may take it as a constant. In such case, denoting

$$s_\lambda(n)^2 = \sigma_\lambda(n)^2/K,$$

we have a simple recursive formula:

$$s_\lambda(n)^2 = \lambda s_\lambda(n-1)^2 + x(n)^2.$$

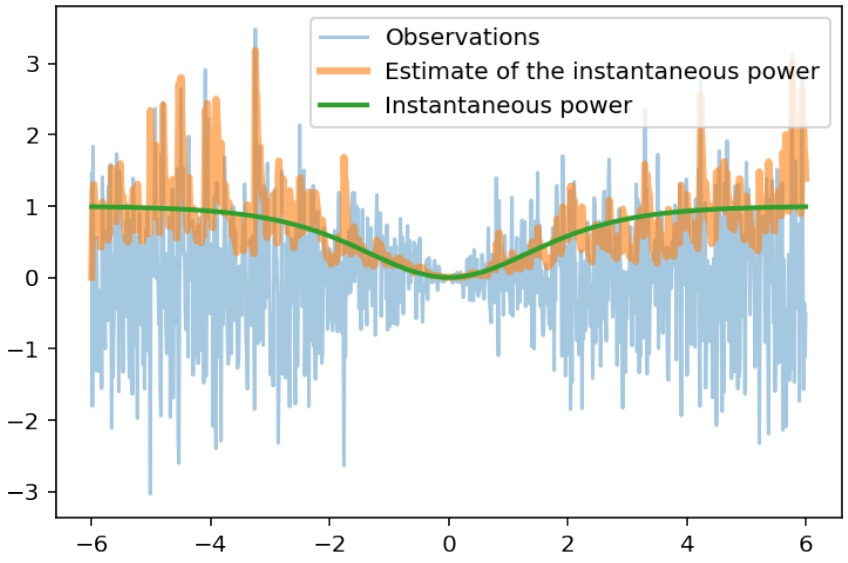
The following lines simulate a non-stationary signal with time-varying power. We implement the exponential average for estimating the power. You should experiment with the values of λ.

```
import matplotlib.pyplot as plt
from IPython.display import clear_output, display, HTML, Image, Javascript
```

```
%matplotlib inline
import numpy as np
import ipywidgets as widgets
from ipywidgets import interact, interactive

N=1000
#mpld3.disable_notebook()
from scipy.special import expit  # logistic function
from IPython.display import display, clear_output
x=np.random.normal(size=N)
t=np.linspace(-6,6,N)
z=x*(2*expit(t)-1)
```

```
def plt_vs_lambda(lamb):

    plt.plot(t,z,alpha=0.4,label='Observations')
    #We implement $s_\lambda(n)^2 = \lambda s_\lambda(n-1)^2 + x(n)^2.$
    slambda=np.zeros(N)
    for n in np.arange(1,N):
        slambda[n]=lamb*slambda[n-1]+z[n]**2
    plt.plot(t,slambda*(1-lamb),lw=3,alpha=0.6,label='Estimate of the
        instantaneous power')
    plt.plot(t,(2*expit(t)-1)**2,lw=2,label='Instantaneous power')
    plt.legend(loc='best')
    clear_output(wait=True)
```

```
lamb=widgets.FloatSlider(min=0,max=1
                         ,value=0.8, step=0.01)
_=interact(plt_vs_lambda, lamb=lamb)
```



Let us return to the normal equation (13.10):

$$\overset{\triangle}{\mathbf{w}} = \mathbf{R}_{uu}^{-1}\mathbf{R}_{du}$$

and to its SDA version (13.3):

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) - \mu\mathbb{E}\left[\mathbf{u}(n)e(n)\right] && (13.33)\\ &= \mathbf{w}(n) - \mu\left(\mathbf{R}_{uu}\mathbf{w}(n) - \mathbf{R}_{du}\right) && (13.34)\\ & && (13.35)\end{aligned}$$

We will substitute the true values with estimated ones. An important remark is that the result of the normal equation is insensitive to a scale factor on the estimates. It is thus possible to estimate the correlation matrix and vector using a sliding average

$$\begin{cases}\hat{\mathbf{R}}_{uu}(n) = \sum_{l=-L}^{L}\mathbf{u}(n-l)\mathbf{u}(n-l)^H\\ \hat{\mathbf{R}}_{du}(n) = \sum_{l=-L}^{L}\mathrm{d}(n-l)\mathbf{u}(n-l)\end{cases}$$

or by an exponential mean

$$\hat{\mathbf{R}}_{uu}(n) = \sum_{l=0}^{n}\lambda^{l-n}\mathbf{u}(l)\mathbf{u}(l)^H = \lambda\hat{\mathbf{R}}_{uu}(n-1) + \mathbf{u}(n)\mathbf{u}(n)^H$$

which yields

$$\begin{cases}\hat{\mathbf{R}}_{uu}(n) = \sum_{l=0}^{n}\lambda^{l-n}\mathbf{u}(l)\mathbf{u}(l)^H = \lambda\hat{\mathbf{R}}_{uu}(n-1) + \mathbf{u}(n)\mathbf{u}(n)^H\\ \hat{\mathbf{R}}_{du}(n) = \lambda\hat{\mathbf{R}}_{du}(n-1) + d(n)\mathbf{u}(n).\end{cases}$$

13.4.1 The Least Mean Square (LMS) Algorithm

The simplest estimator that can be defined is the limit case where we do not average at all… That is we take either $L = 0$ or $\lambda = 0$ in the previous formulas, to get the `instantaneous estimates`

$$\begin{cases}\hat{\mathbf{R}}_{uu}(n) = \mathbf{u}(n)\mathbf{u}(n)^H \\ \hat{\mathbf{R}}_{du}(n) = d(n)\mathbf{u}(n).\end{cases}$$

This merely consists in suppressing the expectations in the theoretical formulas. So doing, we obtain formulas which directly depend on the data, with no need to know something on the theoretical statistics, and which also depend on time, thus confering adaptivity to the algorithm. Plugging these estimates in the SDA, we obtain

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu\mathbf{u}(n)e(n) \tag{13.36}$$
$$= \mathbf{w}(n) - \mu\mathbf{u}(n)\left(\mathbf{u}(n)^H\mathbf{w}(n) - d(n)\right) \tag{13.37}$$
$$\tag{13.38}$$

Substituting $\mathbf{u}(n)\mathbf{u}(n)^H$ for $\mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^H\right]$, or $\mathbf{u}(n)e(n)$ for $\mathbb{E}\left[\mathbf{u}(n)e(n)\right]$ is really a crude approximation. Nevertheless, the averaging occurs by the iterating process so that this kind of method works. The LMS algorithm is by far the most commonly used adaptive filtering algorithm, because it is extremely simple to implement, has a very low computational load, works relatively well and has tracking capabilities.

In order to illustrate the behavior of the LMS algorithm, we continue the example of the identification of an unknown system. We first recreate the data:

13.4.2 Illustation of the LMS in an identification problem

```
from scipy.signal import lfilter
# test
figplot=False
N=800
x=lfilter([1, 1], [1], np.random.randn(N))
htest=10*np.array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
y0=lfilter(htest,[1],x)
y=y0+0.1*randn(N)
if figplot:
    plt.plot(y)
    plt.xlabel("Time")
    plt.title("Observation")
    figcaption("System output in an identification problem")
```

```
/usr/local/lib/python3.5/site-packages/scipy/signal/signaltools.py:1344: FutureWarn
  out = out_full[ind]
```

Now, since one should do it at least one time, try to implement a LMS algorithm. You will will define a function with the following syntax:

```
def lms(d,u,w,mu):
    """
    Implements a single iteration of the stochastic gradient (LMS)\n
    :math:`w(n+1)=w(n)+\\mu u(n)\\left(d(n)-w(n)^T u(n)\\right)`

    Input:
    ======
        d : desired sequence at time n
        u : input of length p
        w : wiener filter to update
        mu : adaptation step
```

```
    Returns:
    =======
        w : upated filter
        err : d-dest
        dest : prediction = :math:`u(n)^T w`
    """
    dest=0
    err=d-dest
#
# DO IT YOURSELF!
#
    return (w,err,dest)
```

You may test your function using the following validation:

```
np.random.seed(327)
wout, errout, destout = lms(np.random.normal(1),np.random.normal(6),np.zeros
    (6),0.05)
wtest = np.array([ 0.76063565,  0.76063565,  0.76063565,  0.76063565,
    0.76063565, 0.76063565])
#Test
if np.shape(wout)==np.shape(wtest):
    if np.sum(np.abs(wout-wtest))<1e-8:
        print("Test validated")
    else:
        print("There was an error in implementation")
else:
    print("Error in dimensions")
```

```
There was an error in implementation
```

A possible implementation is given now

```
def lms(d,u,w,mu):
    """
    Implements a single iteration of the stochastic gradient (LMS)\n
    :math:`w(n+1)=w(n)+\\mu u(n)\\left(d(n)-w(n)^T u(n)\\right)`

    Input:
    ======
        d : desired sequence at time n
        u : input of length p
        w : wiener filter to update
        mu : adaptation step

    Returns:
    =======
        w : upated filter
        err : d-dest
        dest : prediction = :math:`u(n)^T w`
    """
    dest=u.dot(w)
    err=d-dest
    w=w+mu*u*err
    return (w,err,dest)
```

Identification procedure

- Begin by some direct commands (initializations and a `for` loop on the time variable) for identifying the filter; once this works you will implement th commands as a function `ident`
- If necessary, the function `squeeze()` enable to remove single-dimensional entries from the shape of an n-D array (*e.g.* transforms an array (3,1,1) into a vector of dimension 3)

In order to evaluate the algorithm behavior, you will plot the estimation error, the evolution of the coefficients of the identified filter during the iterations of the algorithm; and finally the quadratic error between the true filter and the identified one. This should be done for several orders p (the exact order is unknown...) and for different values of the adaptation step μ.

- The quadratic error can be evaluated simply thanks to a *comprehension list* according to Errh=[sum(he-w[:,n])**2 for n in range(N+1)]

Study the code below, and implement the missing lines.
Étudiez le code ci-dessous et mettez en uvre les lignes manquantes.

```
mu=0.1 # an initial value for mu
L=6  # size of identified filter (true size is p)
NN=200 #number of iterations
err=np.zeros(NN)
w=zeros((L,NN+1))
yest=np.zeros(NN)

# The key lines are here: you have to iterate over time and compute
# the output of the LMS at each iteration.You may save all outputs in the
    matrix
# w initialized above -- column k contains the solution at time k. You must
#äalso save the succession of errors, and the estimated output
#
# You have two lines to implement here.
# DO IT YOURSELF!
#
# After these lines, (w[:,t+1],err[t],yest[t]) are defined


# This is used to define the "true" impulse response  vector with the same
    size as w:
# a shorter (truncated) one if L<p, and a larger one (zero-padded) if L>p.
newhtest=np.zeros(L)
if np.size(htest)<L:
    newhtest=htest[:L]
else:
    newhtest[:np.size(htest)]=htest

# Results:
plt.figure(1)
tt=np.arange(NN)
plt.plot(tt,y0[:NN],label='Initial Noiseless Output')
plt.plot(tt,yest[:NN], label="Estimated Output")
plt.xlabel('Time')
figcaption("Comparison of true output and estimated one after identification
    ",
           label="fig:ident_lms_compareoutputs")
```

```
plt.figure(2)
errh=[sum((newhtest-w[:,t])**2) for t in range(NN)]
plt.plot(tt,errh,label='Quadratic error on h')
plt.legend()
plt.xlabel('Time')
figcaption("Quadratic error between true and estimated filter",
           label="fig:ident_lms_eqonh")
```

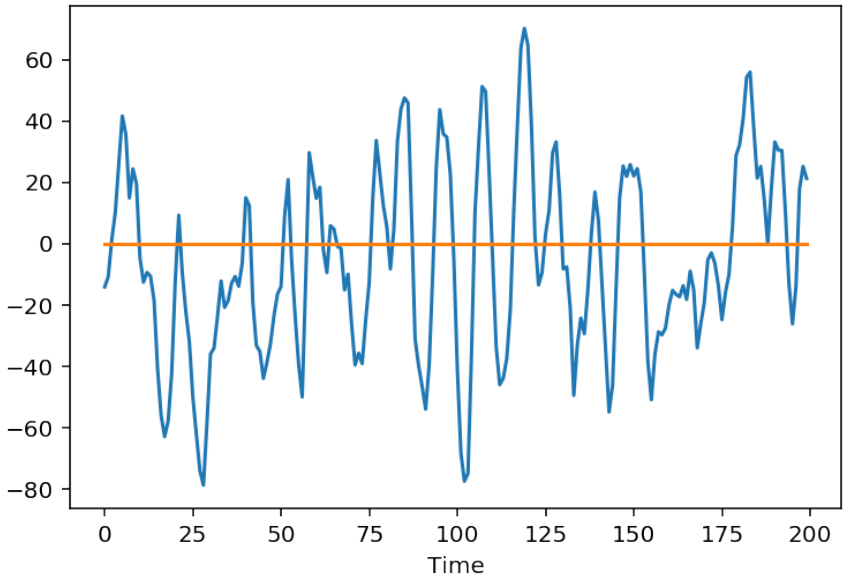


Figure 13.6: Quadratic error between true and estimated filter

The solution is given below:

```
mu=0.05 # an initial value for mu
L=6  # size of identified filter (true size is p)
NN=200 #number of iterations
err=np.zeros(NN)
w=zeros((L,NN+1))
yest=np.zeros(NN)


# The key lines are here: you have to iterate over time and compute
# the output of the LMS at each iteration.You may save all outputs in the
    matrix
# w initialized above -- column k contains the solution at time k. You must
#ăalso save the succession of errors, and the estimated output

for t in np.arange(L,NN):
    (w[:,t+1],err[t],yest[t])=lms(y[t],x[t:t-L:-1],w[:,t],mu)


# This is used to define the "true" impulse response  vector with the same
    size as w:
# a shorter (truncated) one if L<p, and a larger one (zero-padded) if L>p.
newhtest=np.zeros(L)
```

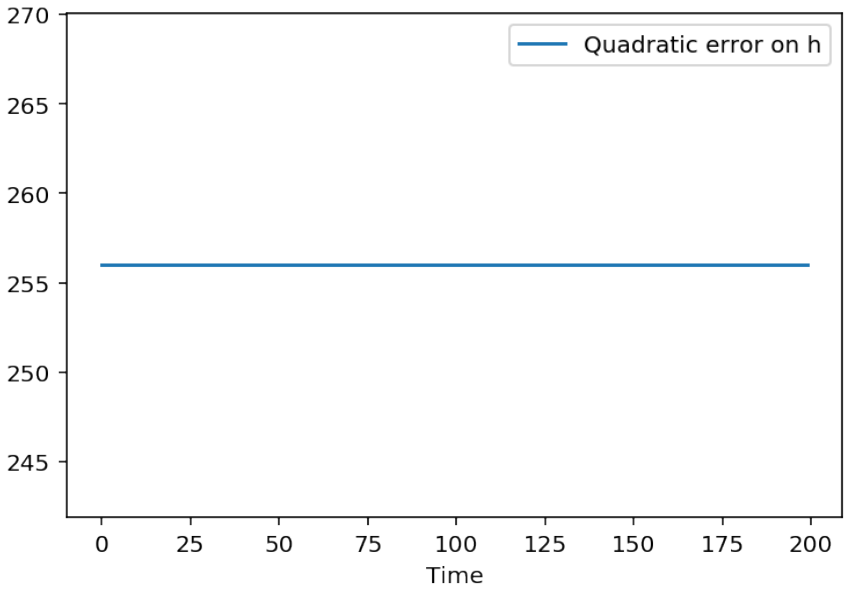


Figure 13.7: Comparison of true output and estimated one after identification

```
LL=np.min([np.size(htest),L])
newhtest[:LL]=htest[:LL]

# Results:
plt.figure(1)
tt=np.arange(NN)
plt.plot(tt,y0[:NN],label='Initial Noiseless Output')
plt.plot(tt,yest[:NN], label="Estimated Output")
plt.xlabel('Time')
figcaption("Comparison of true output and estimated one after identification
    ",
           label="fig:ident_lms_compareoutputs")

plt.figure(2)
errh=[sum((newhtest-w[:,t])**2) for t in range(NN)]
plt.plot(tt,errh,label='Quadratic error on h')
plt.legend()
plt.xlabel('Time')
figcaption("Quadratic error between true and estimated filter",
           label="fig:ident_lms_eqonh")
```

We can now implement the identification as a function on its own, which simply maked some initializations and use a loop on the LMS. Implement this function according to the following syntax.

```
def ident(observation,input_data,mu,p=20,h_initial=zeros(20)):
    """ Identification of an impulse response from an observation
    `observation` of its output, and from its input `input_data`
    `mu` is the adaptation step\n
    Inputs:
    =======
    observation: array
        output of the filter to identify
    input_data: array
```

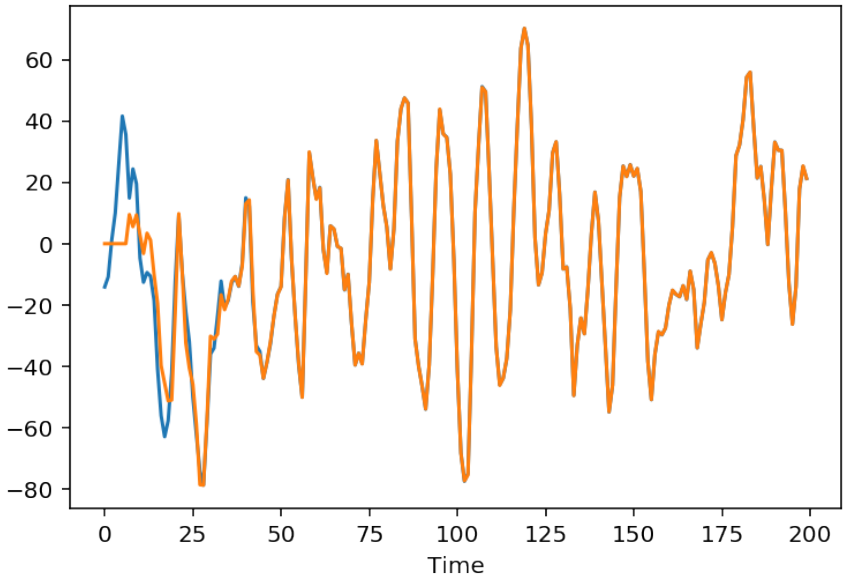


Figure 13.8: Quadratic error between true and estimated filter

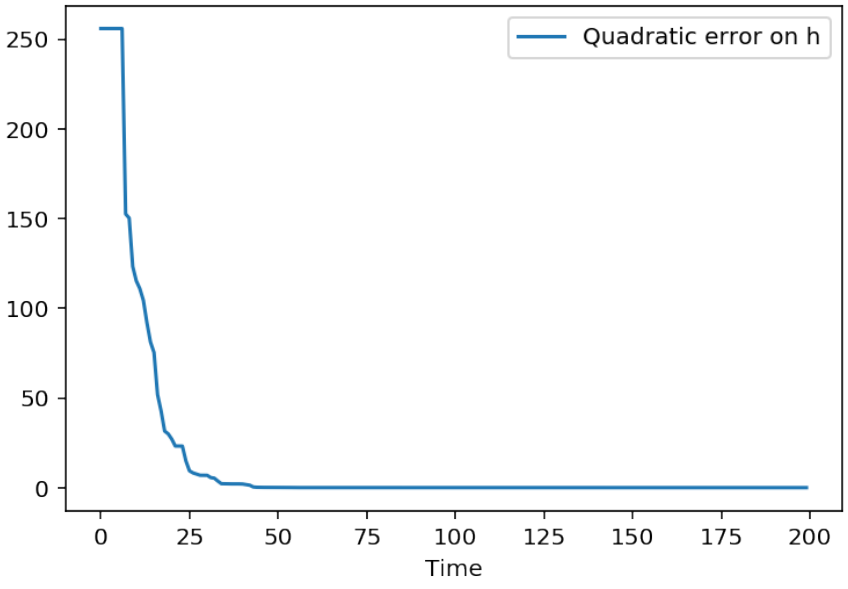


Figure 13.9: Comparison of true output and estimated one after identification

```
        input of the filter to identify
    mu: real
        adaptation step
    p: int (default =20)
        order of the filter
    h_initial: array (default h_initial=zeros(20))
        initial guess for the filter
    normalized: boolean (default False)
        compute the normalized LMS instead of the standard one

    Outputs:
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
    N=np.size(input_data)
    err=np.zeros(N)
    w=np.zeros((p,N+1))
    yest=np.zeros(N)

   #
   # DO IT YOURSELF!
   #

    return (w,err,yest)
```

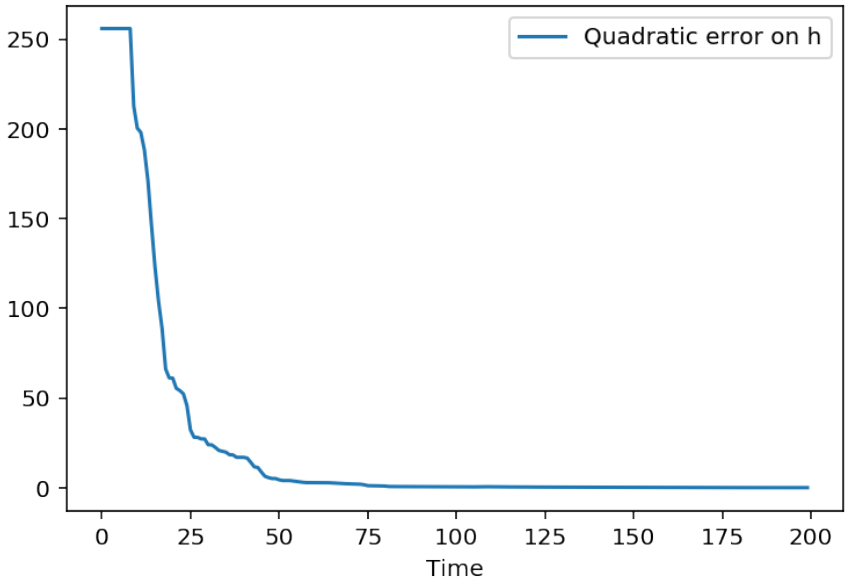
```
def ident(observation,input_data,mu,p=20,h_initial=zeros(20),normalized=
    False):
    """ Identification of an impulse response from an observation
    `observation` of its output, and from its input `input_data` \n
    `mu` is the adaptation step\n
    Inputs:
    =======
    observation: array
        output of the filter to identify
    input_data: array
        input of the filter to identify
    mu: real
        adaptation step
    p: int (default =20)
        order of the filter
    h_initial: array (default h_initial=zeros(20))
        initial guess for the filter
    Outputs:
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
    N=np.size(input_data)
    input_data=squeeze(input_data) #reshape(input_data,(N))
    observation=squeeze(observation)
```

```
    err=np.zeros(N)
    w=np.zeros((p,N+1))
    yest=np.zeros(N)

    w[:,p]=h_initial
    for t in range(p,N):
        if normalized:
            mun=mu/(dot(input_data[t:t-p:-1],input_data[t:t-p:-1])+1e-10)
        else:
            mun=mu
        (w[:,t+1],err[t],yest[t])=lms(observation[t],input_data[t:t-p:-1],w
            [:,t],mun)

    return (w,err,yest)
```

Your implementation can simply be tested with

```
L=8
(w,err,yest)=ident(y,x,mu=0.05,p=L,h_initial=zeros(L))

newhtest=np.zeros(L)
LL=np.min([np.size(htest),L])
newhtest[:LL]=htest[:LL]

NN=np.min([np.size(yest),200])
errh=[sum((newhtest-w[:,t])**2) for t in range(NN)]
plt.plot(tt,errh,label='Quadratic error on h')
plt.legend()
_=plt.xlabel('Time')
print("Identified filter: ",w[:,-1])
```

```
Identified filter:  [ 9.99955367e+00  6.98547337e+00  7.01066832e+00  7.02008973e+0
  2.97926346e+00 -4.36889483e-03 -3.42515122e-02  3.05724531e-02]
```



```
def ident(observation,input_data,mu,p=20,h_initial=zeros(20),normalized=
    False):
    """ Identification of an impulse response from an observation
    'observation' of its output, and from its input 'input_data'
    'mu' is the adaptation step\n
    Inputs:
    =======
    observation: array
        output of the filter to identify
    input_data: array
        input of the filter to identify
    mu: real
        adaptation step
    p: int (default =20)
        order of the filter
    h_initial: array (default h_initial=zeros(20))
        initial guess for the filter
    normalized: boolean (default False)
        compute the normalized LMS instead of the standard one

    Outputs:
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
    N=np.size(input_data)
    err=np.zeros(N)
    w=np.zeros((p,N+1))
    yest=np.zeros(N)

    w[:,p]=h_initial
    for t in np.arange(p,N):
        if normalized:
            assert mu<2, "In the normalized case, mu must be less than 2"
            mun=mu/(np.dot(input_data[t:t-p:-1],input_data[t:t-p:-1])+1e-10)
        else:
            mun=mu
        (w[:,t+1],err[t],yest[t])=lms(observation[t],input_data[t:t-p:-1],w
            [:,t],mun)

    return (w,err,yest)
```

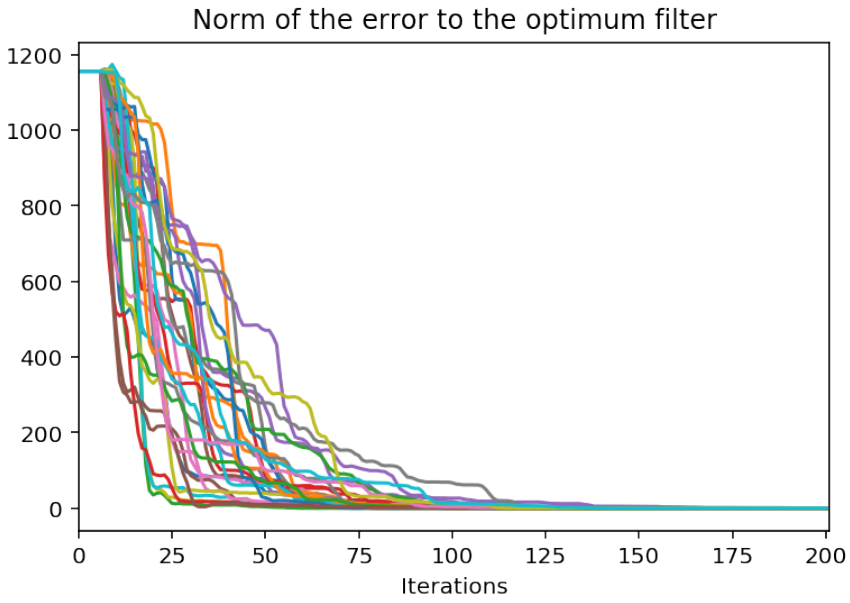
Stability of results

It is very instructive to look at the reproductibility of results when the data change. Let μ fixed and generate new data. Then apply the identification procedure and plot the learning curve.

```
p=6 #<-- actual length of the filter
for ndata in range(30):
    ## Generate new datas
    N=200
    x=lfilter([1, 1], [1], np.random.randn(N))
    htest=10*np.array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
    y0=lfilter(htest,[1],x)
```

```
    y=y0+0.1*randn(N)
    iterations=np.arange(NN+1)
    # ----------------------

    for mu in [0.01]:
        (w,erreur,yest)=ident(y,x,mu,p=p,h_initial=zeros(p))
        Errh=[sum(htest-w[:,n])**2 for n in range(NN+1)]
        plt.plot(iterations,Errh, label="$\mu={}$".format(mu))
        plt.xlim([0, NN+1])


plt.title("Norm of the error to the optimum filter")
_=plt.xlabel("Iterations")
```

```
/usr/local/lib/python3.5/site-packages/scipy/signal/signaltools.py:1344: FutureWarn
  out = out_full[ind]
```



The data are random; the algorithm is stochastic and so is the learning curve! Fortunately, we still check that the algorithms converge... since the error goes to zero. So, it works.

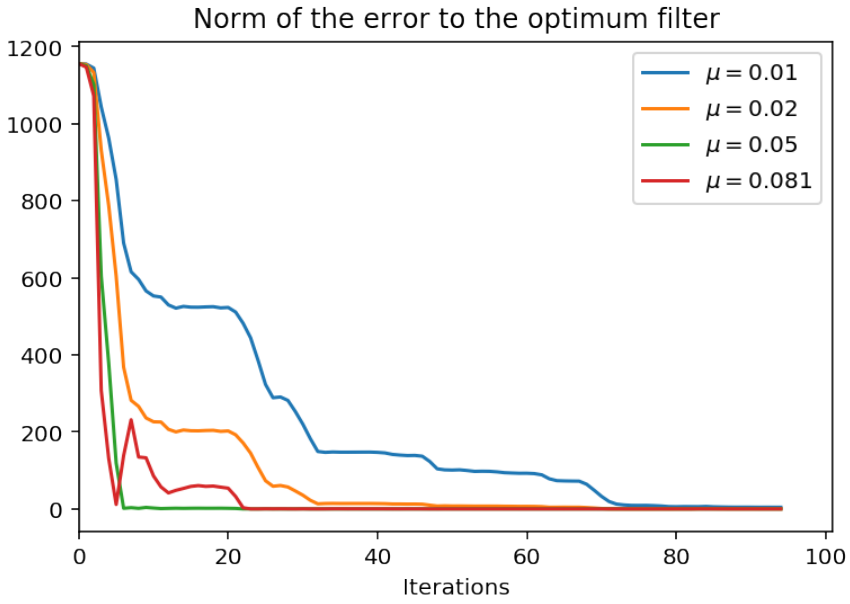
Study with respect to μ

It is really a simple task to study the behavior with respect to the choice of the stepsize μ. We just have to make a loop over possible values of μ, call the identification procedure and display the results.

```
 # Study with respect to $\mu$
p=6
NN=100
iter=np.arange(NN+1)-p

## Generate new datas
```

```
N=200
x=lfilter([1, 1], [1], np.random.randn(N))
htest=10*np.array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
y0=lfilter(htest,[1],x)
y=y0+0.1*randn(N)
# ------------------

for mu in [0.01, 0.02, 0.05, 0.081]:
    (w,erreur,yest)=ident(y,x,mu,p=p,h_initial=zeros(p))
    Errh=[sum(htest-w[:,n])**2 for n in range(NN+1)]
    plt.plot(iter,Errh, label="$\mu={}$".format(mu))
    plt.xlim([0, NN+1])

plt.legend()
plt.title("Norm of the error to the optimum filter")
_=plt.xlabel("Iterations")
```

```
/usr/local/lib/python3.5/site-packages/scipy/signal/signaltools.py:1344: FutureWarn
  out = out_full[ind]
```

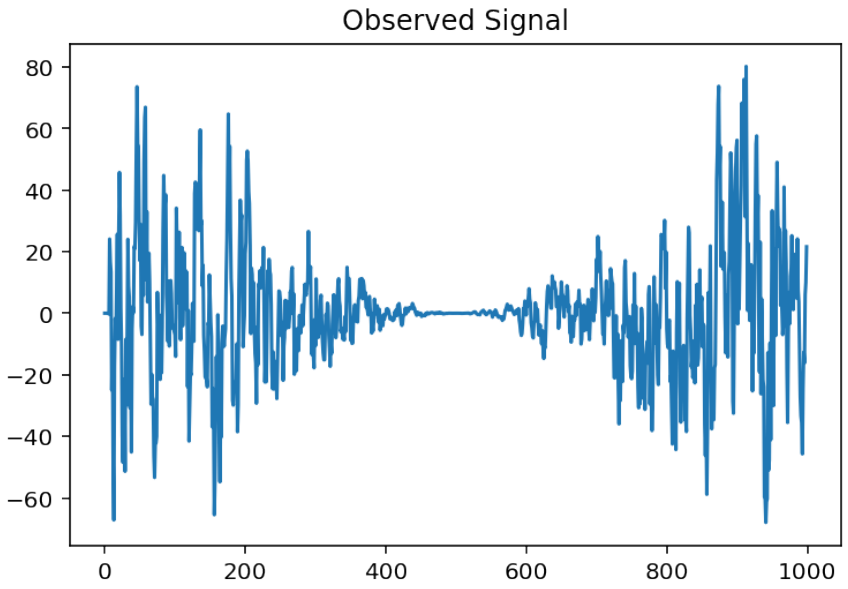


Tracking capabilities

With a constant step-size, the LMS never converge, since while an error exist, the filter is always updated. A consequence of this fact is that the LMS keeps tracking capabilities, which are especially useful in a non-stationary context. In the identification concept, it is possible that the filter to be identified varies during time. In such case, the algorithm must be able to track these modifications. Such an example is simulated below, where the impulse response is modulated by a $\cos()$, according to

$$h(t,\tau) = (1+\cos(2\pi f_0 t))\, h_{\text{test}}(\tau).$$

```
### Slow non-stationarity

N=1000
u=np.random.randn(N)
y=np.zeros(N)
htest=10*np.array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
L=size(htest)
for t in np.arange(L,N):
    y[t]=dot((1+cos(2*pi*t/N))*htest,u[t:t-L:-1])
y+=0.01*np.random.randn(N)
plt.figure()
plt.plot(y)
_=plt.title("Observed Signal")
```



Then, we can test the identification procedure for this non stationary signal. We check that the error indeed goes to zero, and that the identified filter seem effectively modulated with a cosine.

```
p=7
(w,err,yest)=ident(y,u,mu=0.1,p=p,h_initial=zeros(p))
#(w,err,yest)=ident(y,u,mu=1,p=p,h_initial=zeros(p),normalized=True)
plt.figure(1)
clf()
plt.plot(err)
plt.title('Identification error')
figcaption("Identification error in the nonstationary case", label="fig:
    error_ns_case")
plt.figure(2)
plt.clf()
t=np.arange(0,N+1)
true_ns_h=np.outer((1+cos(2*pi*t/N)),htest)
plt.plot(t,w.T,lw=1)
plt.plot(t,true_ns_h,lw=2,label="True values", alpha=0.4)
plt.title("Evolution of filter's coefficients")
figcaption("Evolution of filter's coefficients", label="fig:coeff_ns_case")
```

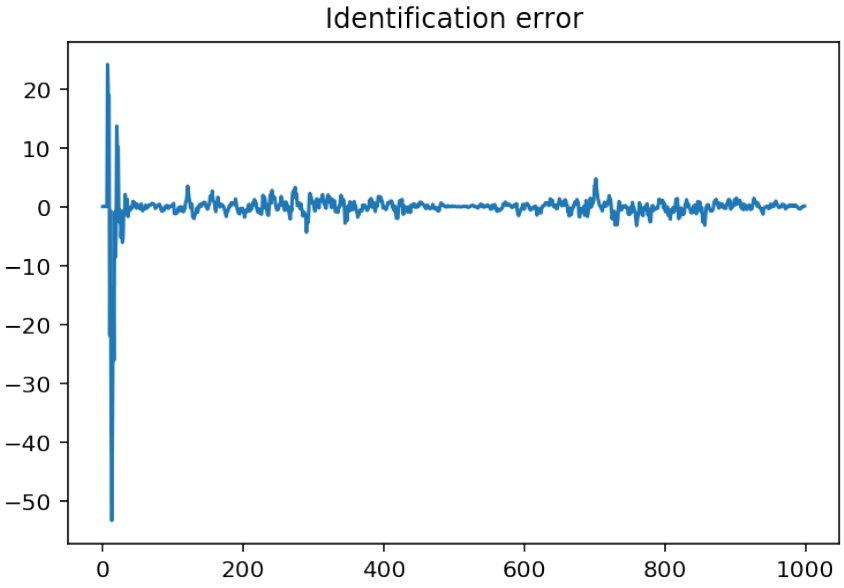


Figure 13.10: Evolution of filter's coefficients

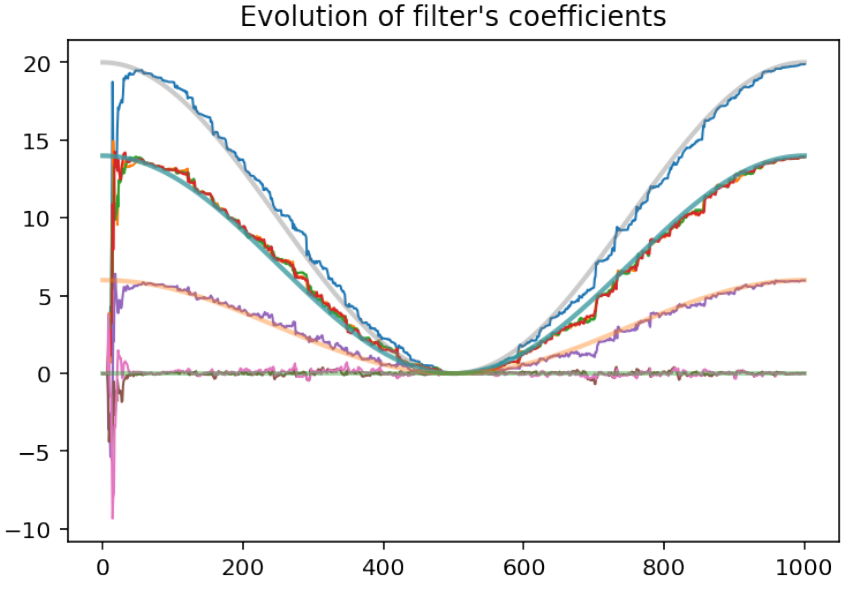


Figure 13.11: Identification error in the nonstationary case

13.4.3 Convergence properties of the LMS

As we realize with these numerical experiments, since the LMS directly depends of the data, the algorithm itself is stochastic; the learning curves have a random character but the mean trajectories still converge to the correct solution. The correct characterization of stochastic algorithms is difficult – actually, the first correct analysis is due to Eweda and Macchi (1983). The traditional analysis relies on a false hypothesis, the *independence assumption*, which still gives a good idea of what happens.

The idea is simply that the average algorithm

$$\mathbb{E}[\mathbf{w}(n+1)] = \mathbb{E}\left[\mathbf{w}(n) - \mu\mathbf{u}(n)\left(\mathbf{u}(n)^T\mathbf{w}(n) - d(n)\right)\right] \tag{13.39}$$

$$= \mathbb{E}[\mathbf{w}(n)] - \mu\mathbb{E}\left[\mathbf{u}(n)\left(\mathbf{u}(n)^T\mathbf{w}(n) - d(n)\right)\right] \tag{13.40}$$

$$\approx \mathbb{E}[\mathbf{w}(n)] - \mu\left(\mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^T\right]\mathbb{E}[\mathbf{w}(n)] - \mathbb{E}[\mathbf{u}(n)d(n)]\right) \tag{13.41}$$

is exactly the true gradient algorithm. Thus, we would have exactly the same conditions for convergence as for the gradient algorithm. However, this is only an approximation. Indeed, in the third line the equality $\mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^T\mathbf{w}(n)\right] = \mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^T\right]\mathbb{E}[\mathbf{w}(n)]$ is incorrect since obviously $\mathbf{w}(n)$ depends on $\mathbf{u}(n)$ through the components at times $n-1, n-2$, etc.

Furthermore, it must be stressed that the learning curves are now **random**. Thus, we can understand that the convergence conditions are more strict than for the gradient algorithm. A practical rule for the choice of μ is

$$\mu = \frac{2}{\alpha \mathrm{Tr}[\mathbf{R}_{uu}]} = \frac{2}{\alpha p R_{uu}(0)},$$

where α is a scalar between 2 and 3, $R_{uu}(0) = \mathbb{E}\left[|u(n)|^2\right]$ and p is the dimension of the correlation matrix.

... to be continued...

Eweda, E., and Macchi, O.. "Quadratic mean and almost-sure convergence of unbounded stochastic approximation algorithms with correlated observations." Annales de l'institut Henri Poincaré (B) Probabilités et Statistiques 19.3 (1983): 235-255. <http://eudml.org/doc/77211>.

@articleEweda1983, author = Eweda, E., Macchi, O., journal = Annales de l'institut Henri Poincaré (B) Probabilités et Statistiques, keywords = almost-sure convergence; correlated observations; quadratic mean convergence; stochastic gradient algorithm; finite memory; finite moments, language = eng, number = 3, pages = 235-255, publisher = Gauthier-Villars, title = Quadratic mean and almost-sure convergence of unbounded stochastic approximation algorithms with correlated observations, url = http://eudml.org/doc/77211, volume = 19, year = 1983,

13.4.4 The normalized LMS

A simple variant of the LMS relies on the idea of introducing an non constant step-size μ_n and to determine an optimum value for the step-size at each iteration. A simple way to show the result is as follows. - The standard error, before updating the LMS from $\mathbf{w}(n)$ into $\mathbf{w}(n+1)$, is

$$e(n|n) = \mathbf{w}(n)^T\mathbf{u}(n) - d(n)$$

- After having updated the filter, we can recompute the error, as

$$e(n|n+1) = \mathbf{w}(n+1)^T\mathbf{u}(n) - d(n).$$

This error is called *a posteriori* error, since it is calculated with the updated filter. This is also indicated by the notation $.|n+1$ which means "computed using the filter at time $n+1$. The standard error is thus qualified of *a priori* error.

Since $\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \mathbf{u}(n) e(n|n)$, we immediately get that

$$e(n|n+1) = \mathbf{w}(n+1)^T \mathbf{u}(n) - d(n) \tag{13.42}$$
$$= (\mathbf{w}(n) - \mu_n \mathbf{u}(n) e(n|n))^T \mathbf{u}(n) - d(n) \tag{13.43}$$
$$= e(n|n) - \mu_n \mathbf{u}(n)^T \mathbf{u}(n) e(n|n) \tag{13.44}$$
$$= \left(1 - \mu_n \mathbf{u}(n)^T \mathbf{u}(n)\right) e(n|n) \tag{13.45}$$

Evidently, updating must decrease the error. Thus, we must have

$$|e(n|n+1)| \leq |e(n|n)|$$

that is

$$\left|\left(1 - \mu_n \mathbf{u}(n)^T \mathbf{u}(n)\right)\right| \leq 1.$$

This yields the condition

$$\boxed{0 \leq \mu_n \leq \frac{2}{\mathbf{u}(n)^T \mathbf{u}(n)}}.$$

The optimum value of the step-size corresponds to the minimum of |e(n|n+1)|, which is simply given by

$$\boxed{\mu_n = \frac{1}{\mathbf{u}(n)^T \mathbf{u}(n)}}.$$

However, the **normalized LMS algorithm** is often given with an auxiliary factor, say $\tilde{\mu}$, which adds a tuning parameter the algorithm

$$\boxed{\mathbf{w}(n+1) = \mathbf{w}(n) - \frac{\tilde{\mu}}{\mathbf{u}(n)^T \mathbf{u}(n)} \mathbf{u}(n) \left(\mathbf{w}(n)^T \mathbf{u}(n) - d(n)\right)}$$

The condition (13.4.4) directly gives

$$\boxed{0 \leq \tilde{\mu} \leq 2},$$

which is a very simple rule.

Implementation of the normalized LMS is a simple modification of the standard LMS. Note that it is useful to introduce a small positive constant in the definition of the step-size

$$\mu_n = \frac{1}{\mathbf{u}(n)^T \mathbf{u}(n) + \varepsilon}$$

in order to avoid division by zero errors.

```
def normalized_lms(d,u,w,mu):
    """
    Implements a single iteration of the stochastic gradient (LMS)\n
    :math:`w(n+1)=w(n)+\\mu u(n)\\left(d(n)-w(n)^T u(n)\\right)`

    Input:
    ======
        d : desired sequence at time n
        u : input of length p
        w : wiener filter to update
        mu : adaptation step for the NLMS; mu <2

    Returns:
    =======
        w : upated filter
```

```
        err : d-dest
        dest : prediction = :math:`u(n)^T w`
    """
    assert mu<2, "In the normalized case, mu must be less than 2"
    u=squeeze(u) #Remove single-dimensional entries from the shape of an
        array.
    w=squeeze(w)
    dest=u.dot(w)
    err=d-dest
    mun=mu/(dot(u,u)+1e-10)
    w=w+mun*u*err
    return (w,err,dest)
```

13.4.5 Other variants of the LMS

The stochastic gradient algorithm is obtained from the theoretical gradient algorithm by approximating the exact statistical quantities by their instantaneous values. This approach can be extended to arbitrary cost functions. Indeed, if we consider a cost function $J(\mathbf{w}) = \mathbb{E}[f(e(n))]$, with f a positive even function, then the steepest descent algorithm leads to

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \frac{\mathrm{d}\mathbb{E}[f(e(n))]}{\mathrm{d}\mathbf{w}(n)} \tag{13.46}$$

$$= \mathbf{w}(n) - \mu \mathbb{E}\left[\mathbf{u}(n)\frac{\mathrm{d}f(e(n))}{\mathrm{d}\mathbf{w}(n)}\right], \tag{13.47}$$

$$\tag{13.48}$$

where we used the chain rule for derivation.

The corresponding stochastic gradient algorithm is then immediately given by

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \mathbf{u}(n)\frac{\mathrm{d}f(e(n))}{\mathrm{d}e(n)}.$$

Let us look at some examples:

- if $f(e) = |e|$, then $f'(e) = \mathrm{sign}(e)$ and we obtain the so-called **sign-error** algorithm:

  $$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \mathbf{u}(n)\mathrm{sign}(e(n)).$$

  This is an early algorithm with very low complexity, which can be implemented without any multiplications (if μ is a power of 2, then the step-size multiplication can be implemented as a bit shift).

- for $f(e) = |e|^k$, then $f'(e) = k|e|^{k-1}\mathrm{sign}(e)$, and the stochastic gradient algorithm has the form

  $$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \mathbf{u}(n)|e(n)|^{k-1}\mathrm{sign}(e(n)).$$

See Mathews, ece6550 -chapter4, page 22, for an example of a piecewise linear cost function leading to a quantization of the error.

13.4.6 Recursive Least Squares

Instead of taking an instantaneous estimate of the correlation matrix and vector, it is still possible to go on with the exponential mean estimates

$$\begin{cases}\hat{\mathbf{R}}_{uu}(n+1)=\sum_{l=0}^{n+1}\lambda^{l-n-1}\mathbf{u}(l)\mathbf{u}(l)^H=\lambda\hat{\mathbf{R}}_{uu}(n)+\mathbf{u}(n+1)\mathbf{u}(n+1)^H\\ \hat{\mathbf{R}}_{du}(n+1)=\lambda\hat{\mathbf{R}}_{du}(n)+d(n+1)\mathbf{u}(n+1).\end{cases}$$

It remains to compute the solution

$$\hat{\mathbf{w}}(n+1)=\left[\hat{\mathbf{R}}_{uu}(n+1)\right]^{-1}\hat{\mathbf{R}}_{du}(n+1). \tag{13.49}$$

The main problem is the inversion, for each n, of the correlation matrix. Fortunately, it is possible to obtain a recursive solution which do not need a matrix inversion at all... The key here is to invoke the matrix inversion lemma

$$[\mathbf{A}+\mathbf{BD}]^{-1}=\mathbf{A}^{-1}-\mathbf{A}^{-1}\mathbf{B}[\mathbf{I}+\mathbf{DA}^{-1}\mathbf{B}]^{-1}\mathbf{DA}^{-1}. \tag{13.50}$$

Applying this with $\mathbf{A}=\lambda\hat{\mathbf{R}}_{uu}(n-1)$, $\mathbf{B}=\mathbf{u}(n)$ and $\mathbf{C}=\mathbf{u}(n)^H$, and denoting

$$\mathbf{K}_{n+1}=\left[\hat{\mathbf{R}}_{uu}(n+1)\right]^{-1}$$

we readily obtain

$$\mathbf{K}(n+1)=\frac{1}{\lambda}\mathbf{K}(n)-\frac{1}{\lambda^2}\frac{\mathbf{K}(n)\mathbf{u}(n+1)\mathbf{u}(n+1)^H\mathbf{K}(n)}{1+\frac{1}{\lambda}\mathbf{u}(k+1)^H\mathbf{K}(n)\mathbf{u}(k+1)}, \tag{13.51}$$

and after several lines of calculations, we arrive at the updating formula

$$\hat{\mathbf{w}}(n+1)=\hat{\mathbf{w}}(n)+\mathbf{K}(n+1)\mathbf{u}(n+1)[d(n+1)-\mathbf{w}(n)^H\mathbf{u}(n+1)]. \tag{13.52}$$

Note that there are some notational differences between the LMS and the RLS. For the LMS, the filter $\mathbf{w}(n+1)$ is calculated based on the data available at time n. For the RLS, $\mathbf{w}(n+1)$ is computed using data available at time $(n+1)$. This is just a notational difference – we could easily rename $\mathbf{w}(n+1)$ into say $\mathbf{v}(n)$ and obtain similar indexes. However these notations are traditional, so we follow the classical developments and equations. What is important however is to note that both filters are calculated using the *a priori* error, that is the error using the data at time n and the filter computed using the data at time $n-1$.

Initialization - The initialization of the algorithm requires the specification of an initial $\mathbf{w}(0)$ which is usually taken as a null vector. It also requires specifying $\mathbf{K}(0)$. Since $\mathbf{K}(0)$ is the inverse of the correlation matrix before the beginning of the iterations, we usually choose $\mathbf{R}_{uu}(0)=\delta\mathbf{I}$, with δ very small. So the inverse is $\mathbf{K}(0)=\delta^{-1}\mathbf{I}$, a large value which disappears during the iterations of the algorithm.

An implementation of the RLS algorith is proposed below, using the standard `numpy array` type as well as the `matrix` type. Casting from one type to the other is done by `np.matrix` or `np.array` keywords (which make a copy), or using `np.asmatrix` or `np.asarray` keywords.

```
# Implementation using the array type
def algo_rls(u,d,M,plambda):
    N=size(u)
# initialization
    e=zeros(N)
    wrls=zeros((M,N+1))
    Krls=100*eye(M)
    u_v=zeros(M)
    for n in range(N):
```

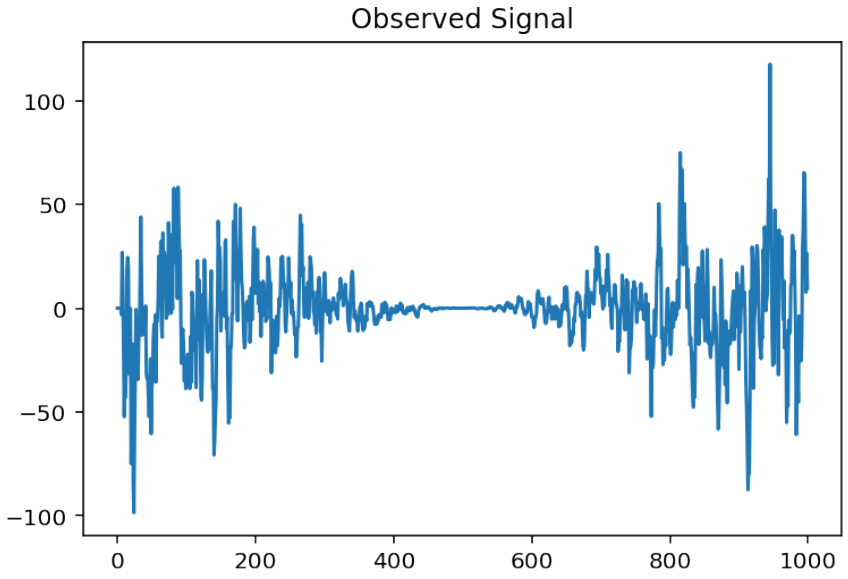
```
        u_v[0]=u[n]
        u_v[1:M]=u_v[0:M-1]#concatenate((u[n], u_v[1:M]), axis=0)
        e[n]=conj(d[n])-dot(conj(u_v),wrls[:,n])
        # print("n={}, Erreur de {}".format(n,e[n]))
        Kn=Krls/plambda
        Krls=Kn-dot(Kn,dot(outer(u_v,conj(u_v)),Kn))/(1+dot(conj(u_v),dot(Kn
            ,u_v)))
        wrls[:,n+1]=wrls[:,n]+dot(Krls,u_v)*conj(e[n])
    return (wrls,e)

## RLS, matrix version

def col(v):
    """ transforms an array into a column vector \n
    This is the equivalent of  x=x(:) under Matlab"""
    v=asmatrix(v.flatten())
    return reshape(v,(size(v),1))

def algo_rls_m(u,d,M,plambda):
    """
    Implementation with the matrix type instead of the array type
    """
    N=size(u)
        # initialization
    e=zeros(N)
    wrls=matrix(zeros((M,N+1)))
    Krls=100*matrix(eye(M))
    u=col(u)
    u_v=matrix(col(zeros(M)))

    for n in range(N):
        u_v[0]=u[n]
        u_v[1:M]=u_v[0:M-1]
                #u_v=concatenate(u[n], u_v[:M], axis=0)
        e[n]=conj(d[n])-u_v.H*wrls[:,n]
        Kn=Krls/plambda
        Krls=Kn-Kn*(u_v*u_v.H*Kn)/(1+u_v.H*Kn*u_v)
        wrls[:,n+1]=wrls[:,n]+Krls*u_v*conj(e[n])

    return (wrls,e)
```

At this point, it would be useful to do again the previous experimentations (identification with non stationary data) with the RLS algorithm. Then to compare and conclude.

```
def ident_rls(observation,input_data,factor_lambda=0.95,p=20):
    """ Identification of an impulse response from an observation
    'observation' of its output, and from its input 'input_data' \n
    'mu' is the adaptation step\n
    Inputs:
    =======
    observation: array
        output of the filter to identify
    input_data: array
        input of the filter to identify
    factor_lambda: real (defaut value=0.95)
        forguetting factor in the RLS algorithm
    p: int (default =20)
        order of the filter
    Outputs:
```

```
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
    N=np.size(input_data)
    input_data=squeeze(input_data) #reshape(input_data,(N))
    observation=squeeze(observation)
    (wrls,e)= algo_rls(input_data,observation,p,factor_lambda)
#   (w[:,t+1],erreur[t],yest[t])=lms(input_data[t:t-p:-1],w[:,t],mun)
    return (wrls,e)
```

```
### Slow non-stationarity

N=1000
u=np.random.randn(N)
y=np.zeros(N)
htest=10*np.array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
L=size(htest)
for t in np.arange(L,N):
    y[t]=dot((1+cos(2*pi*t/N))*htest,u[t:t-L:-1])
y+=0.01*np.random.randn(N)
plt.figure()
plt.plot(y)
_=plt.title("Observed Signal")
```




```
p=7
lamb=0.97
(w,err)=ident_rls(y,u,factor_lambda=lamb,p=10)
plt.figure(1)
clf()
plt.plot(err)
```

```
plt.title('Identification error')
figcaption("Identification error in the nonstationary case", label="fig:
    error_ns_case")
plt.figure(2)
plt.clf()
t=np.arange(0,N+1)
true_ns_h=np.outer((1+cos(2*pi*t/N)),htest)
plt.plot(t,w.T,lw=1)
plt.plot(t,true_ns_h,lw=2,label="True values", alpha=0.4)
plt.title("Evolution of filter's coefficients")
figcaption("Evolution of filter's coefficients", label="fig:coeff_ns_case")
```

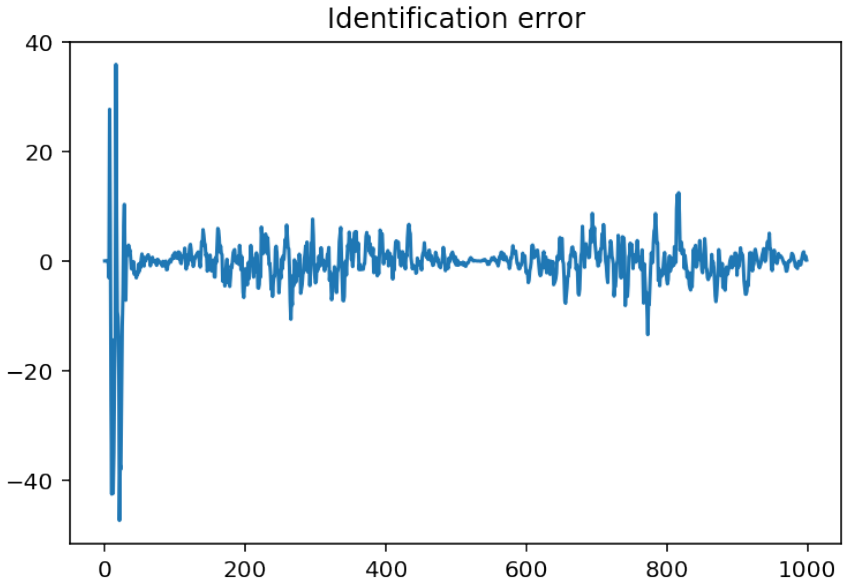


Figure 13.12: Evolution of filter's coefficients

References:

- http://www.ece.utah.edu/~mathews/ece6550/chapter10.pdf
- http://www.cs.tut.fi/~tabus/course/ASP/LectureNew10.pdf
- Recursive Least Squares at wikipedia
- Adaptive Filtering Applications (open access book at intechopen).

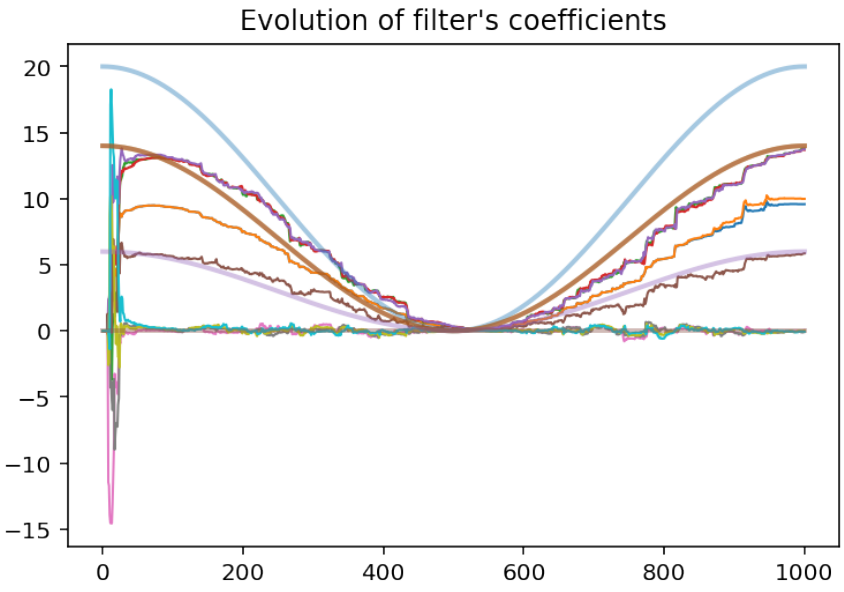


Figure 13.13: Identification error in the nonstationary case