# A Journey in Signal Processing with IPython

Jean-François Bercher

ESIEE-Paris

# Contents

```
%run nbinit.ipy
```

```
... Configuring matplotlib formats
... Configuring matplotlib with inline figures
... Importing numpy as np, scipy as sp, pyplot as plt, scipy.stats as stats
   ... scipy.signal as sig
... Importing widgets, display, HTML, Image, Javascript
... Some LaTeX definitions

... Defining figures captions

... Loading customized Javascript for interactive solutions (show/hide)
```

# 1

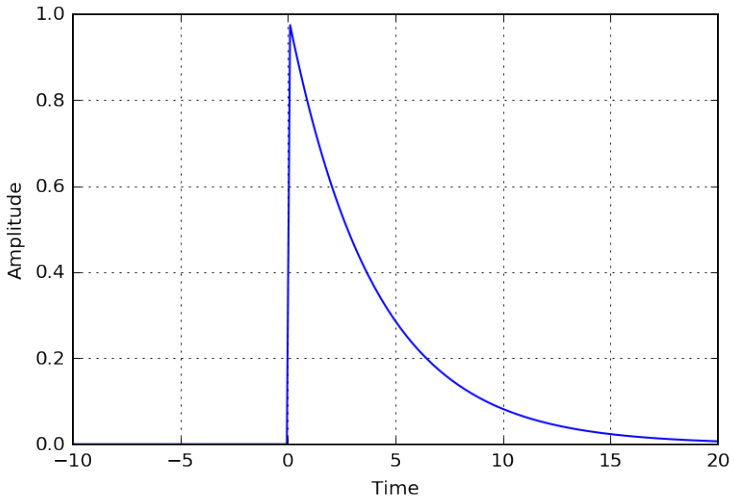
# A basic introduction to signals and systems

## 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.

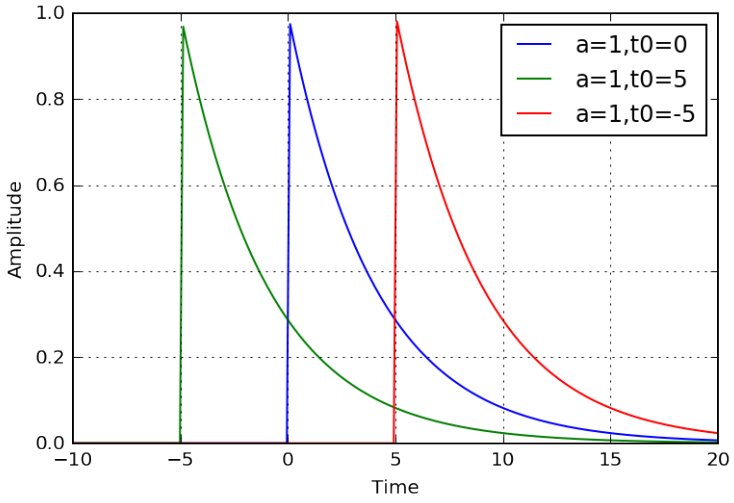
Study the code below and experiment with the parameters

```
# 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='on')


# 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 (...)
```



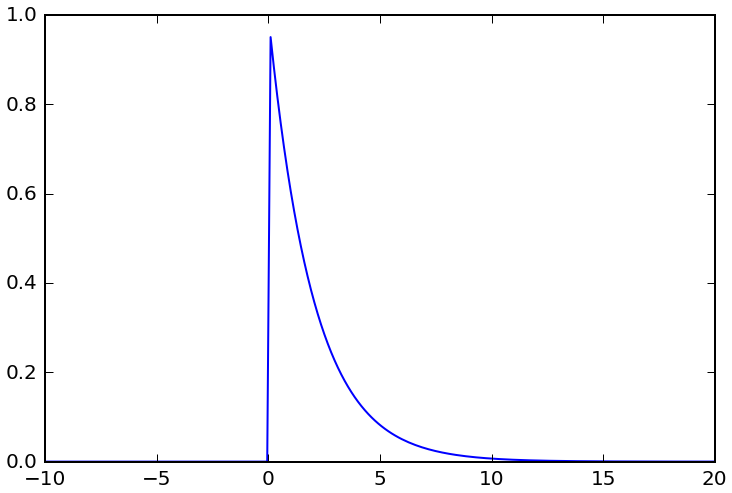

This to show that you do automatically several tests and plot the results all together.

```
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='on')
plt.legend()
```




And finally an interactive version

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

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)
        len(t)
        len(x)
        plt.plot(t,x)

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)
```




```
%run nbinit.ipy
```

```
... Configuring matplotlib formats
... Configuring matplotlib with inline figures
... Importing numpy as np, scipy as sp, pyplot as plt, scipy.stats as stats
   ... scipy.signal as sig
... Importing widgets, display, HTML, Image, Javascript
... Some LaTeX definitions

... Defining figures captions

... Loading customized Javascript for interactive solutions (show/hide)
```

## 1.2 A basic introduction to filtering

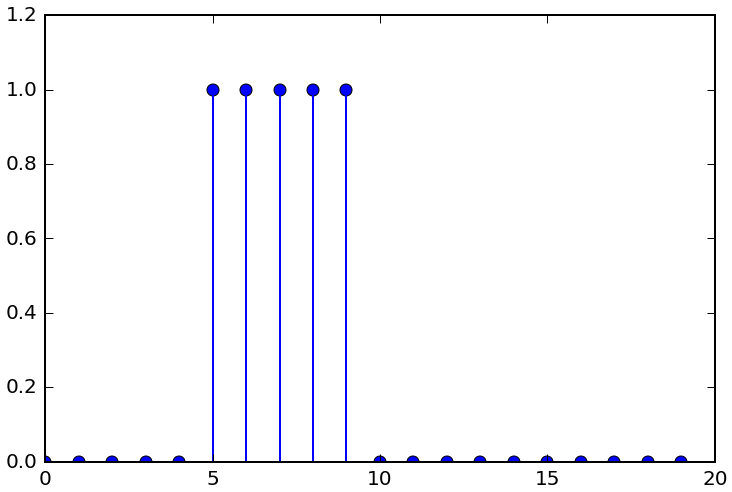
Through examples, we define several operations on signals and show how they transform them. Then we define what is a filter and the notion of impulse response.

- Transformations of signals - Examples of difference equations
- Filters
- Notion of impulse response

### 1.2.1 Transformations of signals - Examples of difference equations

We begin by defining a test signal.

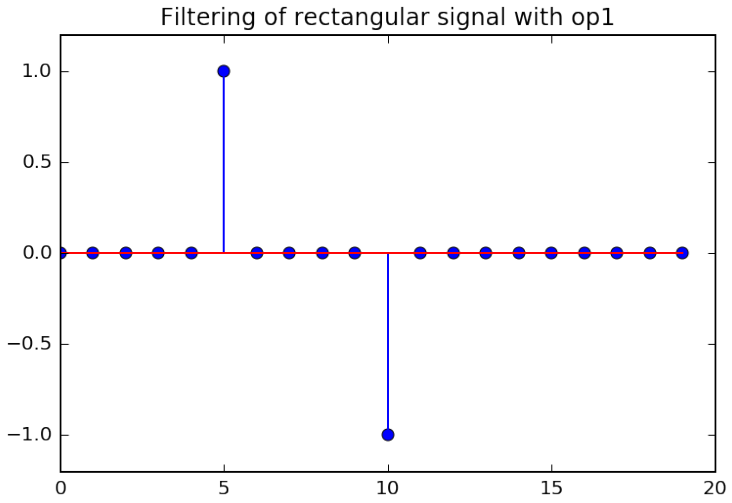
```
# rectangular pulse
N=20; L=5; M=10
r=np.zeros(N)

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

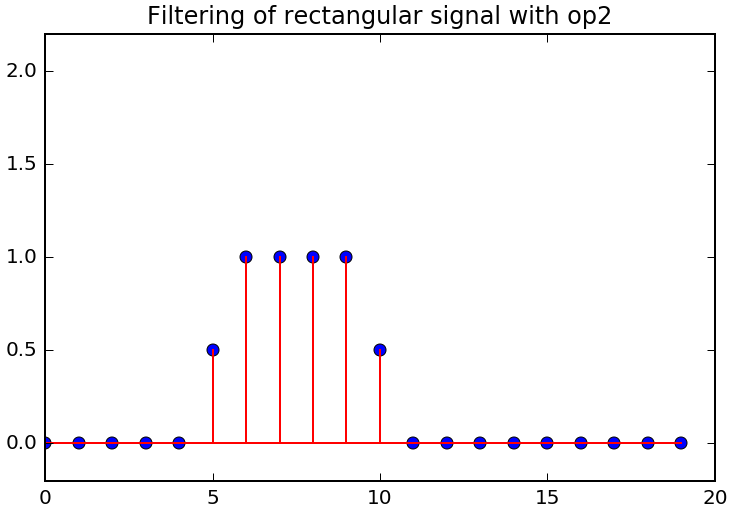



```
def op1(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]=signal[t]-signal[t-1]
    return transformed_signal

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])
```

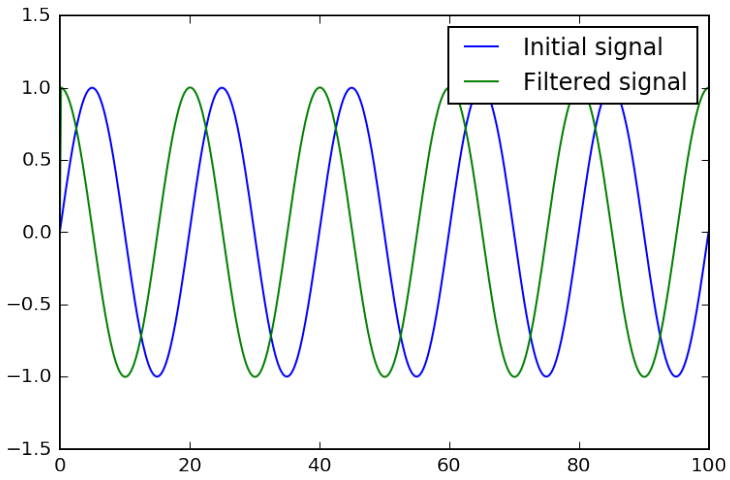
```
plt.title("Filtering of rectangular signal with op1")
plt.figure()
plt.stem(op2(r),'r')
_=plt.ylim([-0.2, 2.2])
plt.title("Filtering of rectangular signal with op2")
```



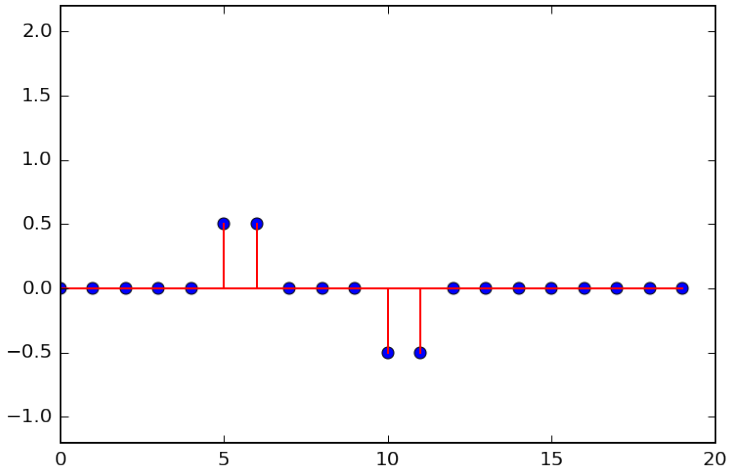

We define a sine wave and check that the operation implemented by "op1" seems to be a derivative...

```
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()
```

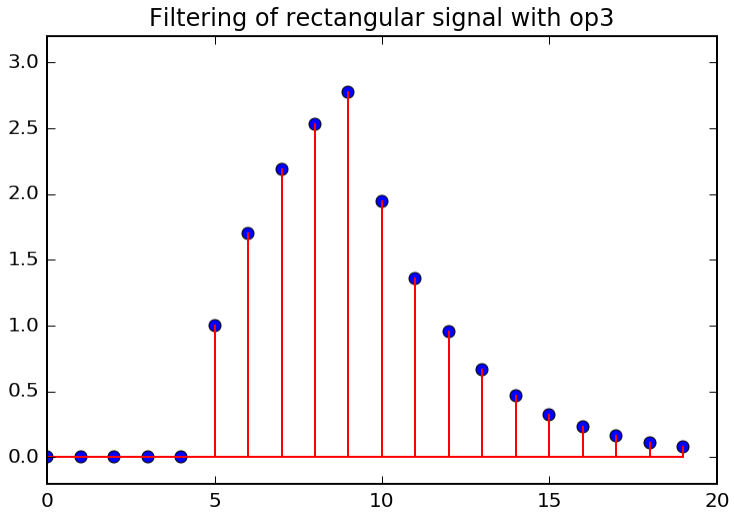



Composition of operations:

```
plt.stem(op1(op2(r)),'r')
_=plt.ylim([-1.2, 2.2])
```

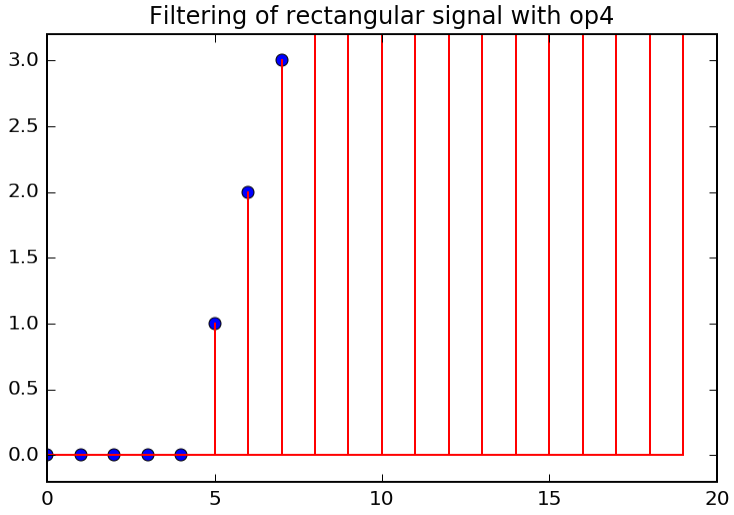


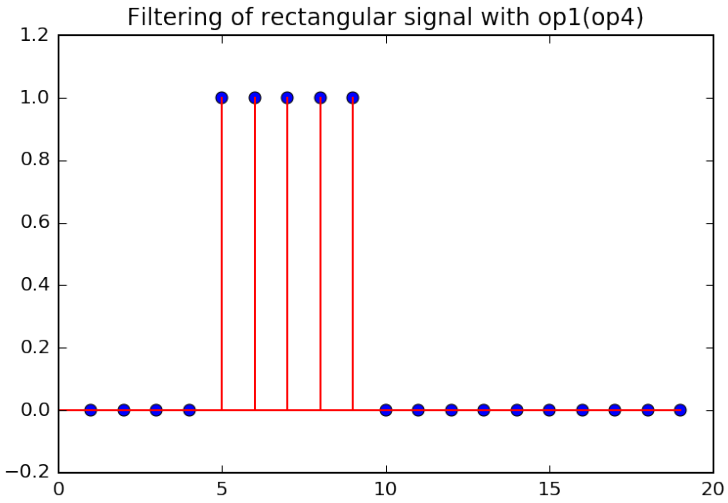

```
def op3(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]= 0.7*transformed_signal[t-1]+signal[t]
    return transformed_signal

plt.stem(op3(r),'r')
plt.title("Filtering of rectangular signal with op3")
_=plt.ylim([-0.2, 3.2])
```




### 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, 3.2])
# 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])
```






### 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** A Dirac impulse (or impulse for short) is defined by

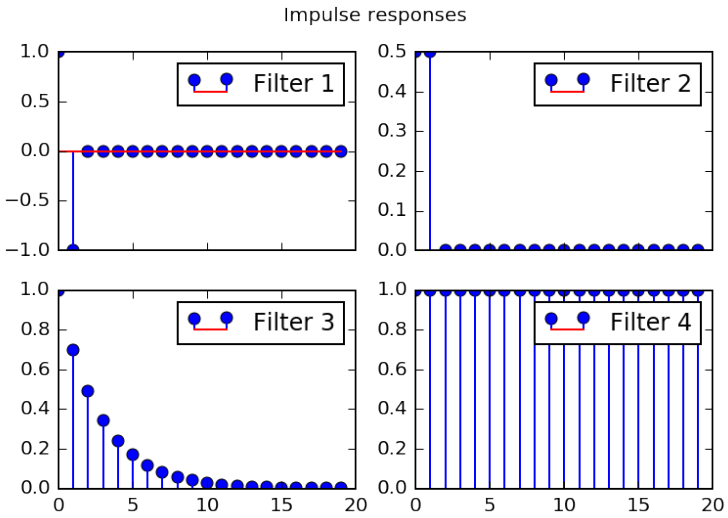
$$\delta(n) = \begin{cases} 1 & \text{if } n=0 \\ 0 & \text{elsewhere} \end{cases} \tag{1.1}$$

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

$$\delta(n) \to \text{System} \to h(n) \tag{1.2}$$

```
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
```

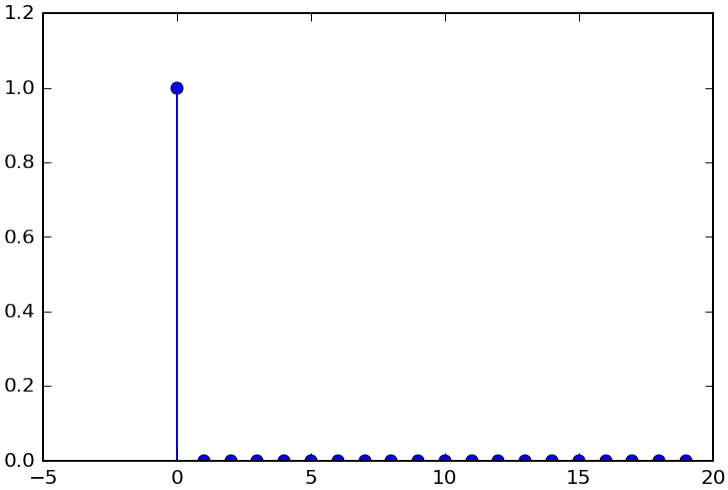
```
d=dirac_vector(20)
fig,ax=plt.subplots(2,2,sharex=True)
plt.subplot
ax[0][0].stem(op1(d),   label="Filter 1")
ax[0][0].legend()
ax[0][1].stem(op2(d),   label="Filter 2")
ax[0][1].legend()
ax[1][0].stem(op3(d), label="Filter 3")
ax[1][0].legend()
ax[1][1].stem(op4(d), label="Filter 4")
ax[1][1].legend()
plt.suptitle("Impulse responses")
```




### 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 impuse response of the indentity filter is a Dirac impulse.

```
%run nbinit.ipy
```

```
... Configuring matplotlib formats
... Configuring matplotlib with inline figures
... Importing numpy as np, scipy as sp, pyplot as plt, scipy.stats as stats
   ... scipy.signal as sig
... Importing widgets, display, HTML, Image, Javascript
... Some LaTeX definitions

... Defining figures captions

... Loading customized Javascript for interactive solutions (show/hide)
```

# 2

# Introduction to the Fourier representation

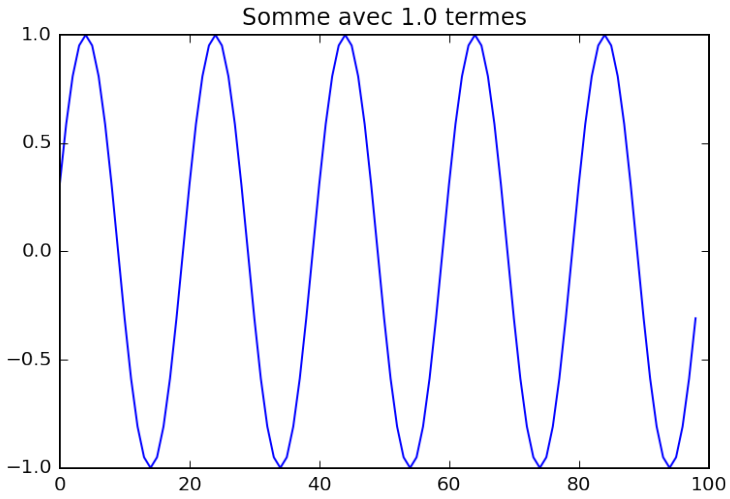
We begin by a simple example which shows that the addition of some sine waves, with special coefficients, converges constructively. We then explain that any periodic signal can be expressed as a sum of sine waves. This is the notion of Fourier series. After an illustration (denoising of a corrupted signal)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

- Simple examples
- Decomposition on basis - scalar producs
- Decomposition of periodic functions – Fourier series
- Complex Fourier series
- Computer experiment
- Towards Fourier transform

## 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(1,3,2):
    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")
```

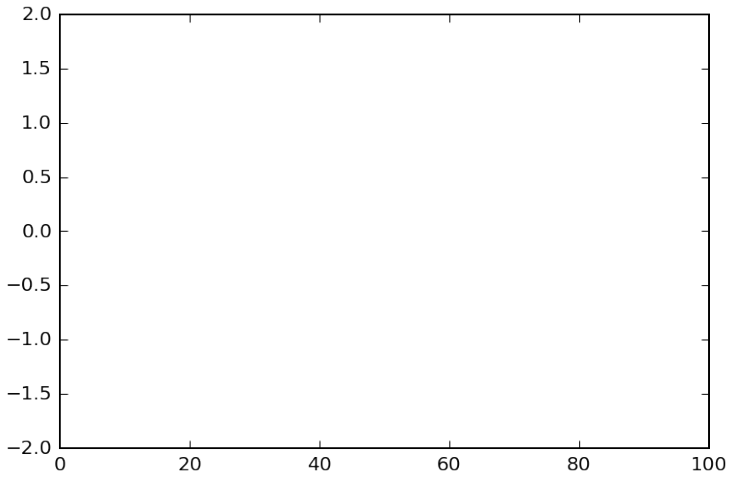



The next example is more involved in that it sums sin a cos of different frequencies and with different amplitudes. We also add widgets (sliders) which enable to interact more easily with the program.

```
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))

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

```
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=1)
val=widgets.IntText(value='1')

#----- Callbacks des widgets ------------
def sfou1_Km(name,Km):
        val.value=str(Km)
```

```
        sfou_exp(Km)

def sfou2_Km(name, Km):
        slide.value=Km
        #sfou_exp(Km.value)


# ---- Display ----------------
display(slide)
display(val)
slide.on_trait_change(sfou1_Km,'value')
val.on_trait_change(sfou2_Km,'value')
```

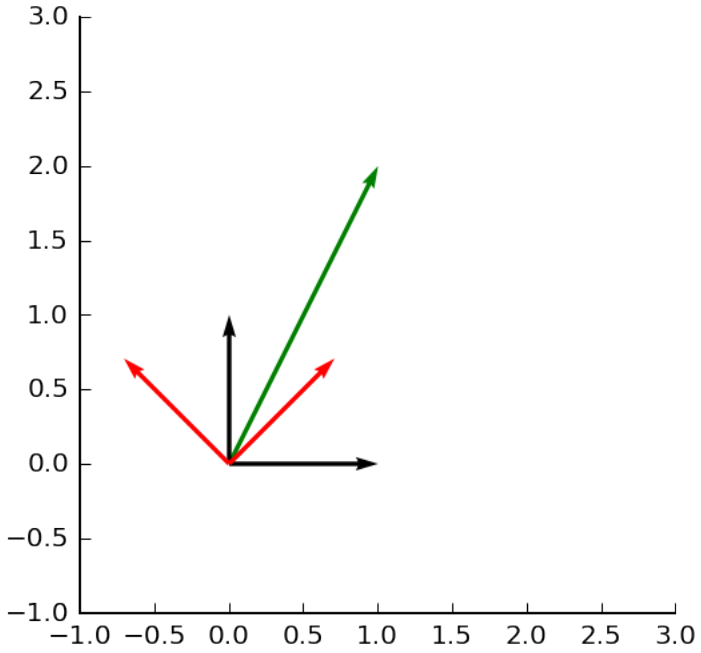



### 2.1.1 Decomposition on basis - scalar producs

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:  2.12132034356 0.707106781187
```

## 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)
```

```
2.0000000000000013
```

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

$$x(n) = \sqrt{\frac{2}{L}}\left(\frac{a_0}{2} + \sum_{k=1}^{+\infty} a_k \cos(2\pi k/Ln)\right) \tag{2.1}$$

with

$$a_k = \sqrt{\frac{2}{L}} \sum_{n\in[L]} x(n)\cos(2\pi k/Ln). \tag{2.2}$$

Regrouping the factors, the series can also be expressed as

$$x_{\text{even}}(n) = \left(\frac{a_0}{2} + \sum_{k=1}^{L-1} a_k \cos(2\pi k/Ln)\right) \tag{2.3}$$

with

$$a_k = \frac{2}{L} \sum_{n\in[L]} x(n)\cos(2\pi k/Ln), \tag{2.4}$$

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:

$$x_{\text{odd}}(n) = \sum_{k=0}^{L-1} b_k \sin(2\pi k/Ln) \tag{2.5}$$

with

$$b_k = \frac{2}{L} \sum_{n\in[L]} x(n)\sin(2\pi k/Ln), \tag{2.6}$$

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

$$x(n) = x_{\text{even}}(n) + x_{\text{odd}}(n) = \frac{x(n)+x(-n)}{2} + \frac{x(n)-x(-n)}{2}, \tag{2.7}$$

we have the sum of the decompositions:

$$\boxed{x(n) = \frac{a_0}{2} + \sum_{k=1}^{L-1} a_k \cos(2\pi k/Ln) + \sum_{k=1}^{+\infty} b_k \sin(2\pi k/Ln)} \tag{2.8}$$

with

$$\boxed{\begin{cases} a_k = \frac{2}{L}\sum_{n\in[L]} x(n)\cos(2\pi k/Ln), \\ b_k = \frac{2}{L}\sum_{n\in[L]} x(n)\sin(2\pi k/Ln), \end{cases}} \tag{2.9}$$

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 sl (i.e. norm of sk): ",np.vdot(sk,sk
    ))
```

```
scalar product between sk and sl:  (-1.04083408559e-17+3.25260651746e-18j)
scalar product between sk and sl (i.e. norm of sk):  (1+0j)
```

It is thus possible to decompose a signal as follows:

$$\boxed{\begin{aligned} x(n) &= \sum_{k=0}^{L-1} c_k e^{j2\pi\frac{kn}{L}} \\ \text{with } c_k &= \frac{1}{L}\sum_{n\in[L]} x(n) e^{-j2\pi\frac{kn}{L}} \end{aligned}} \tag{2.10}$$

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

$$x(n) = c_0 + \sum_{k=1}^{+\infty} c_k e^{j2\pi k/Ln} + c_{-k} e^{-j2\pi k/Ln}. \tag{2.11}$$

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

$$\begin{aligned} x(n) = c_0 + \sum_{k=1}^{+\infty} &\mathcal{R}\{c_k + c_{-k}\}\cos(2\pi k/Ln) + \mathcal{I}\{c_{-k} - c_k\}\sin(2\pi k/Ln) \\ &+ j\left(\mathcal{R}\{c_k - c_{-k}\}\sin(2\pi k/Ln) + \mathcal{I}\{c_k + c_{-k}\}\cos(2\pi k/Ln)\right). \end{aligned} \tag{2.12}$$

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

$$\begin{cases} a_k = \mathcal{R}\{c_k + c_{-k}\} \\ b_k = \mathcal{I}\{c_{-k} - c_k\} \end{cases} \tag{2.13}$$

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

$$\begin{cases} \mathcal{R}\{c_k\} = \mathcal{R}\{c_{-k}\} \\ \mathcal{I}\{c_k\} = -\mathcal{I}\{c_{-k}\}. \end{cases} \tag{2.14}$$

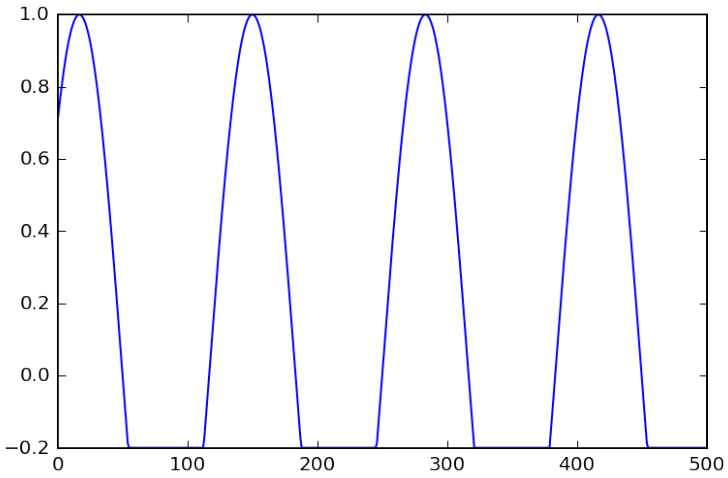
This symmetry is called 'Hermitian symmetry'.

### 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 0x7ff010046630>]
```




A function for computing the Fourier series coefficients

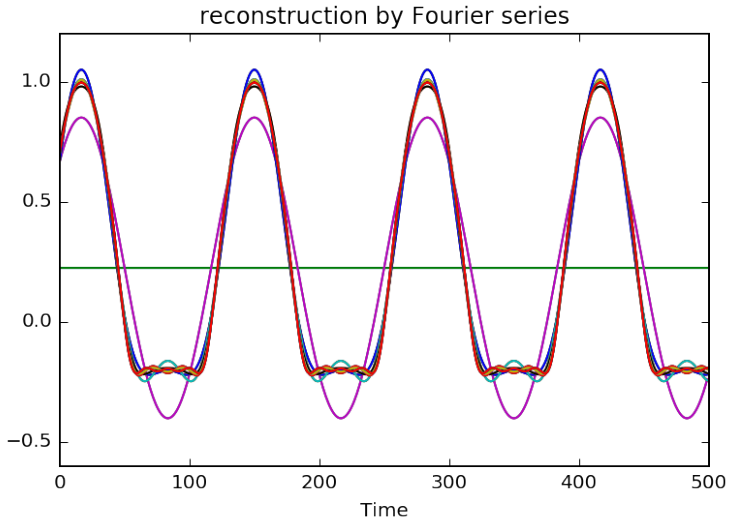
```
# compute the coeffs ck
def coeffck(x,L,k):
    assert np.size(x)==L, "input must be of length L"
    karray=[]
    res=[]
    if isinstance(k,int):
        karray.append(k)
    else:
        karray=np.array(k)

    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)
```

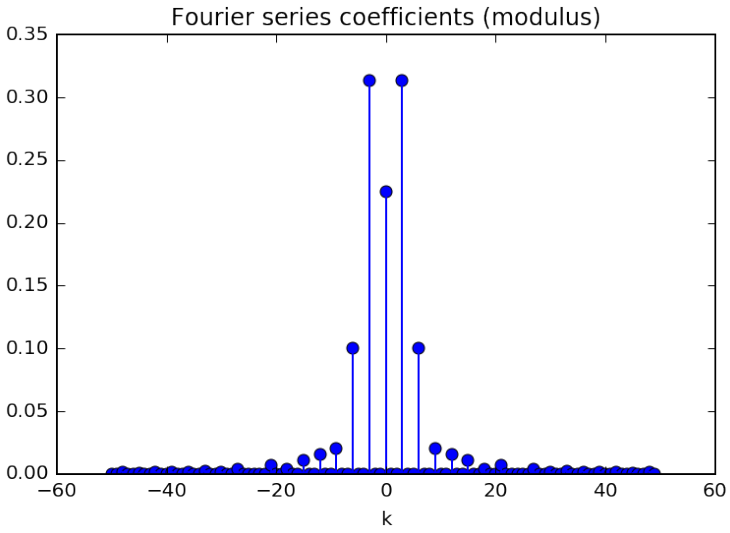
```
#test: coeffck(x[0:L],L,[-12,1,7])
# --> array([  1.51702135e-02 +4.60742555e-17j,
#      -1.31708229e-05 -1.31708229e-05j,   1.37224241e-05 -1.37224241e-05j
    ])
```

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")


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 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.
```



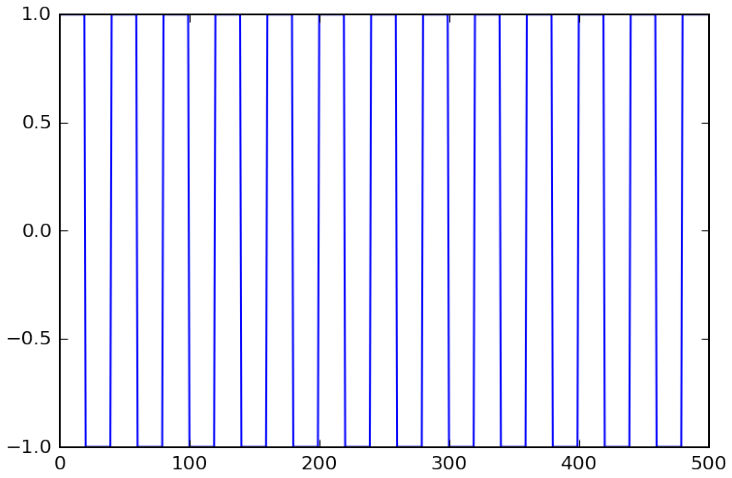




A pulse train corrupts our original signal

```
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)
```

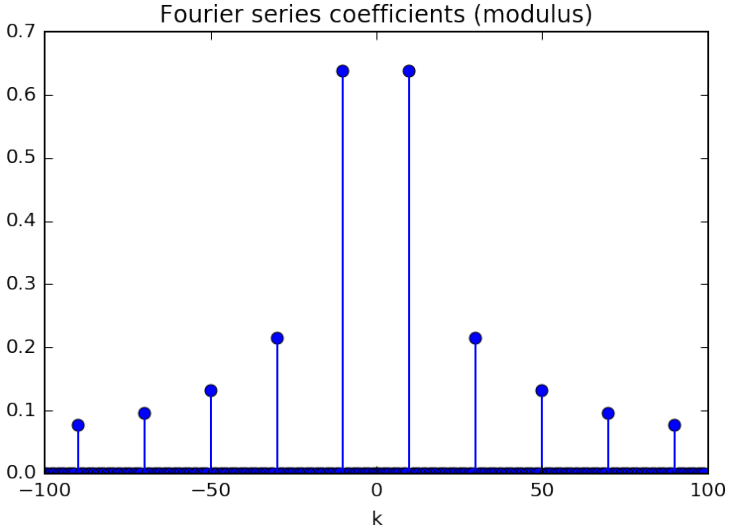
```
#sig=sign(sin(2*pi*10*t/L))

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)
```

```
[<matplotlib.lines.Line2D at 0x7ff00e393cc0>]
```



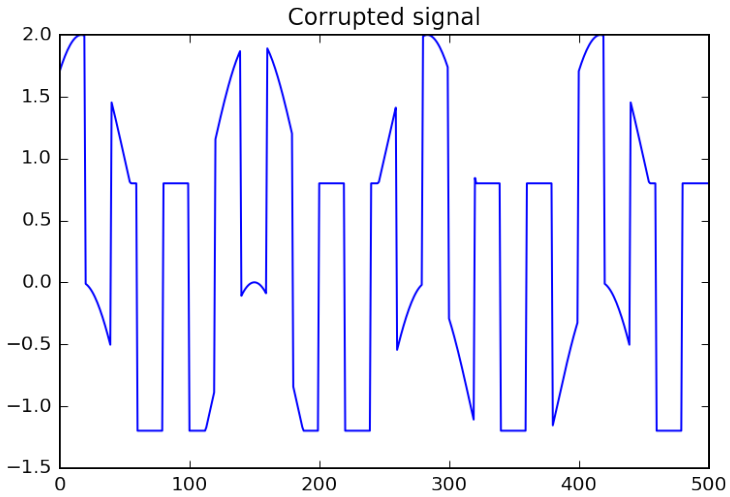

Compute and represent the Fourier coeffs of the pulse train

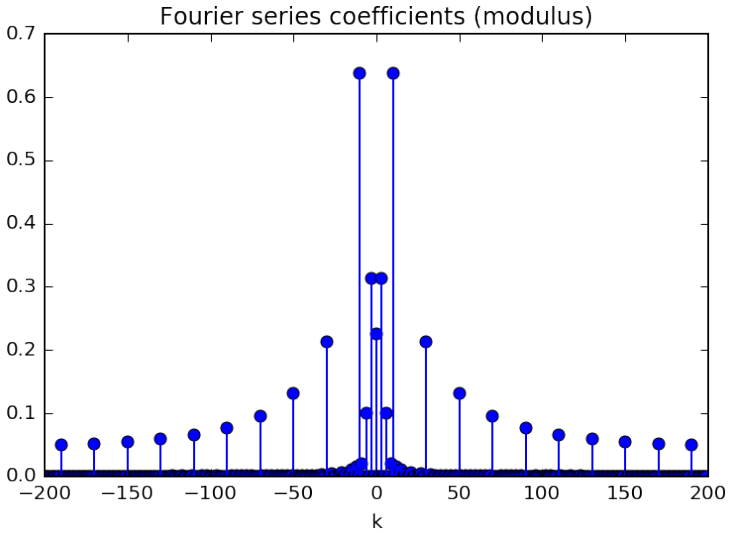
```
kk=np.arange(-100,100)
c=coeffck(sig[0:L],L,kk)
plt.figure()
plt.stem(kk,np.abs(c))
plt.title("Fourier series coefficients (modulus)")
plt.xlabel("k")
```



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).

```
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")
```

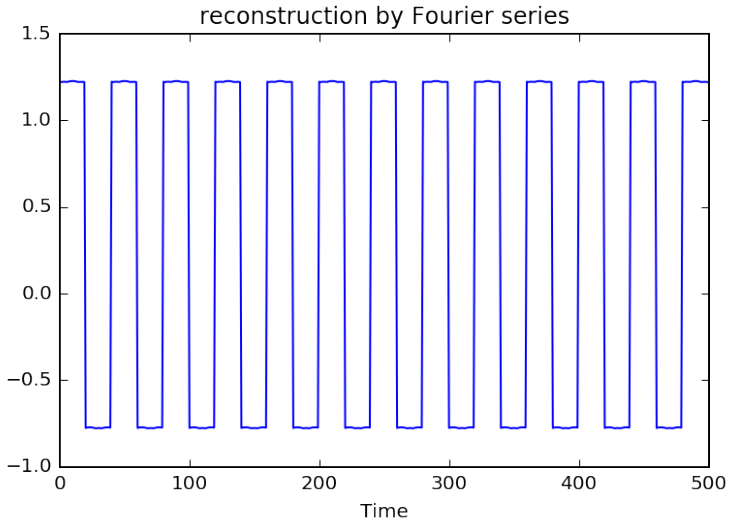


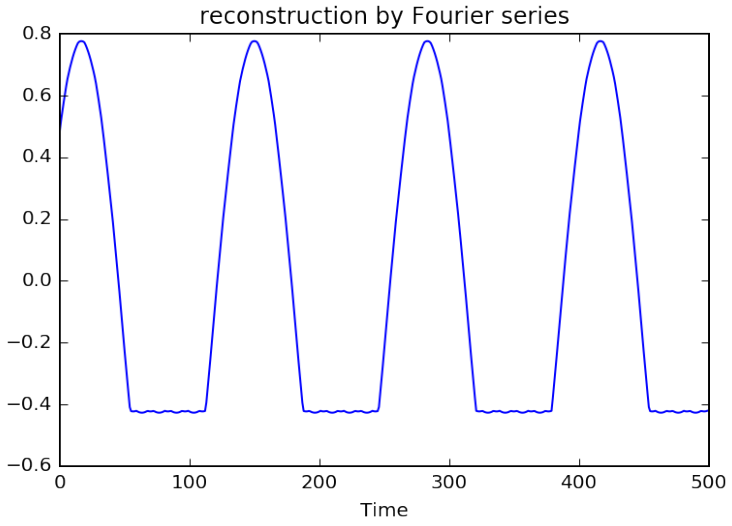




Now, we try to kill all the frequencies harmonics of 10 (the fundamental frequency of the pulse train), and reconstruct the resulting signal...

```
# 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")
```



# 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

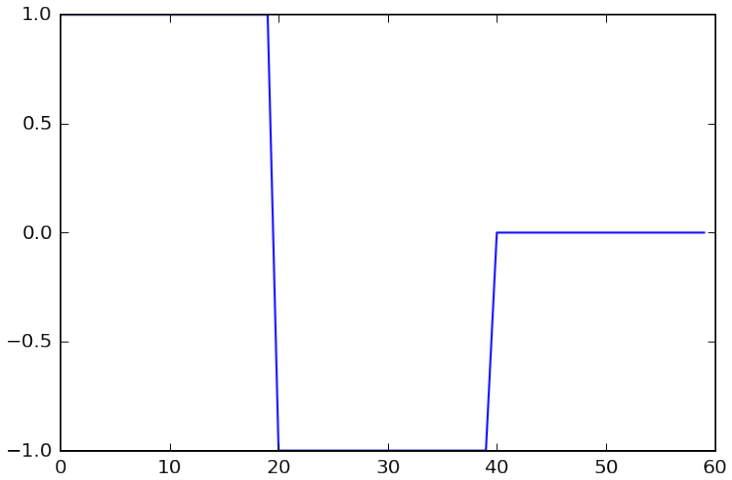
$$\begin{aligned} x(n) &= \sum_{k=0}^{N-1} X(k) e^{j2\pi \frac{kn}{N}} \\ \text{with } X(k) &= \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{kn}{N}} \end{aligned} \tag{3.1}$$

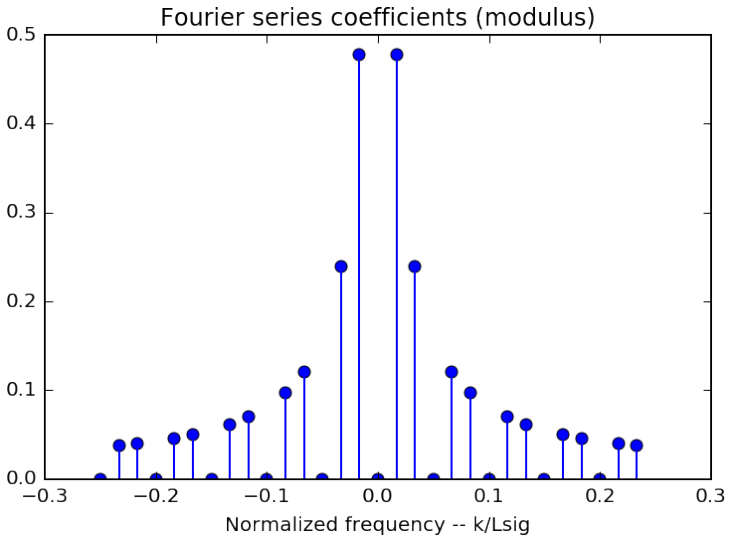
- we may also consider that there is nothing –that is zeros, outside of the observation interval. In such condition, we can still imagine that we have a periodic signal, but with an infinite period. Since the separation of two harmonics in the Fourier series is $\Delta f$=1/period, we see that $\Delta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






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:

$$\begin{aligned} x(n) = \sum_{k=0}^{N-1} c_k e^{j2\pi\frac{kn}{N}} = \sum_{k/N=0}^{1-1/N} NX(k)e^{j2\pi\frac{kn}{N}}\frac{1}{N} \\ \rightarrow x(n) = \int_0^1 X(f)e^{j2\pi fn}\mathrm{d}f \end{aligned} \tag{3.2}$$

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

$$\boxed{\begin{aligned} x(n) &= \int_0^1 X(f)e^{j2\pi fn}\mathrm{d}f \\ \text{with } X(f) &= \sum_{n=-\infty}^{\infty} x(n)e^{-j2\pi fn} \end{aligned}} \tag{3.3}$$

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.

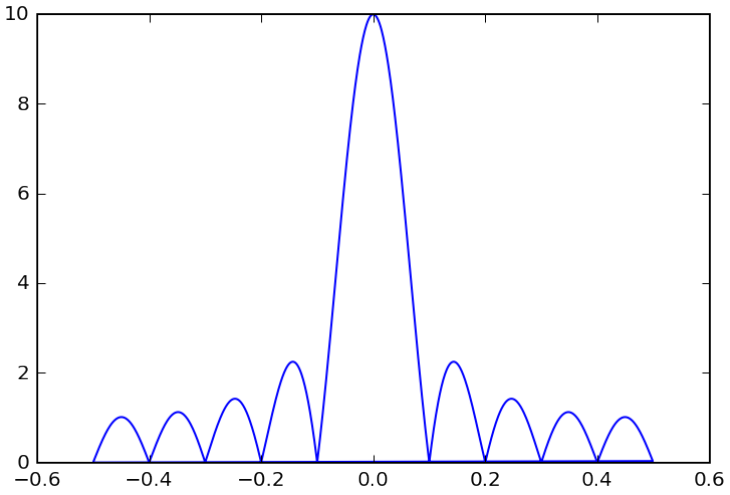
For the numerical experiments, import the fft (Fast Fourier Transform) function,

```
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))
```

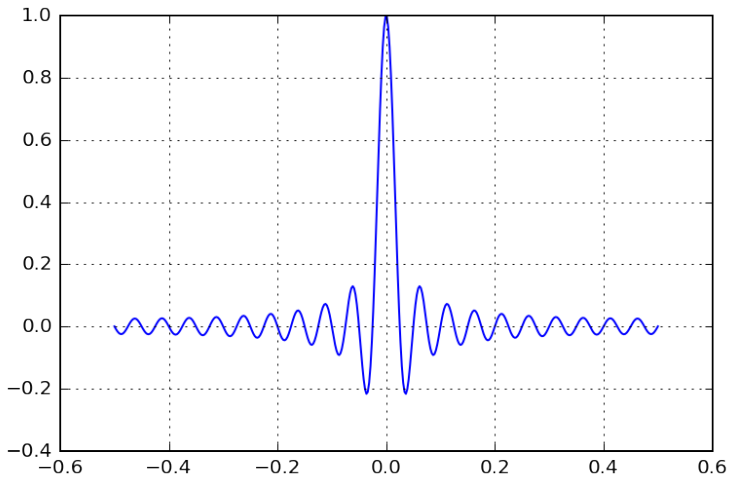
```
[<matplotlib.lines.Line2D at 0x7f3fd0e5bef0>]
```




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

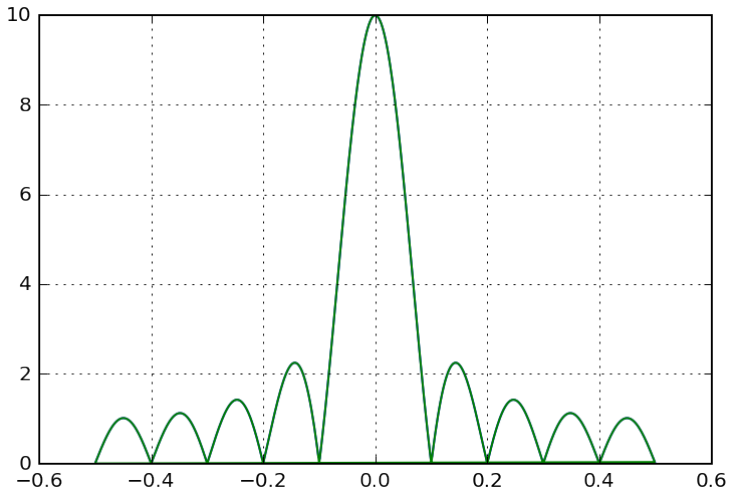
```
def dsinc(x,L):
    if isinstance(x,(int,float)): x=[x]
    x=np.array(x)
    out=np.ones(np.shape(x))
    I=np.where(x!=0)
    out[I]=np.sin(x[I])/(L*np.sin(x[I]/L))
    return out
```

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



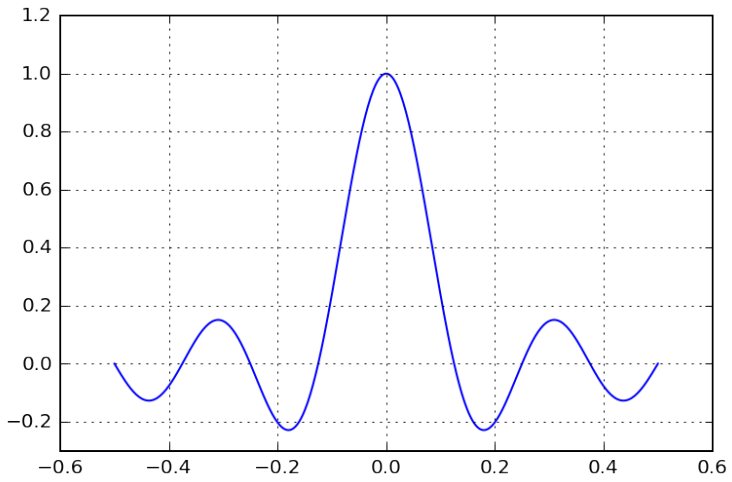

Comparison of the Fourier transform of a rectangle and a cardinal sine:

```
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='on')
```




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(name,L):
        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='on')

pltsinc('Width',8)
slider.on_trait_change(pltsinc,'value')
```



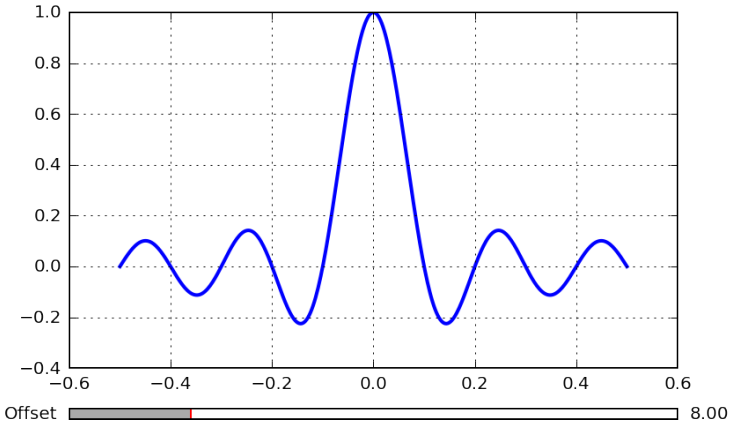

This is an example with matplotlib widgets interactivity, (instead of html widgets). The docs can be found at

```
%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='on')

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: agg
```

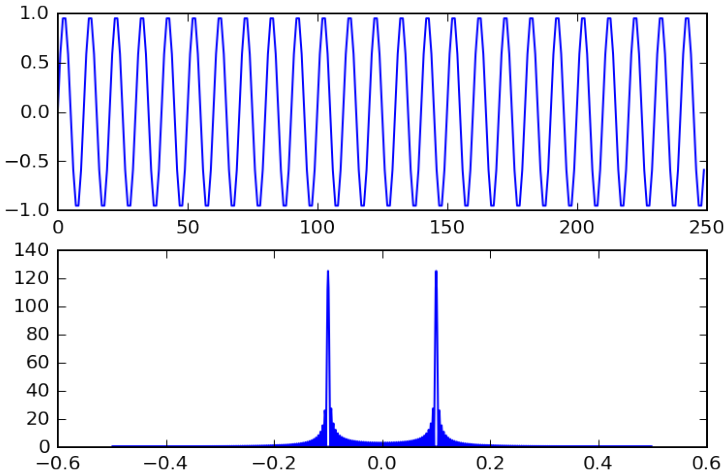
```
0
```




### 3.2.2 Fourier transform of a sine wave

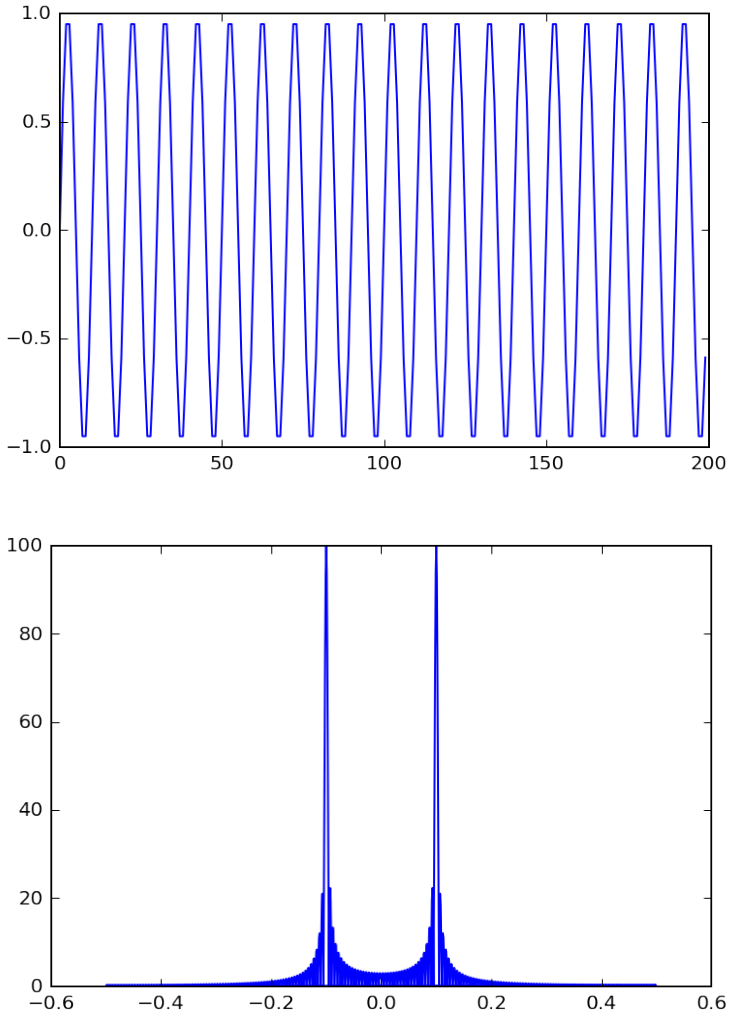
Again, the derivation will be done in class.

```
%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(min=1,max=1000,step=1,value=200)
sliderf0=widgets.FloatSlider(min=0,max=0.5,step=0.01,value=0.1)

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)


#------ Callbacks des widgets ----------------
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)
        plt.figure(1)
        plt.clf()
        plt.plot(t,s)
        plt.figure(2)
        plt.clf()
        plt.plot(f,np.abs(Sf))

pltsin(8)
sliderN.on_trait_change(pltsin)
sliderf0.on_trait_change(pltsin)
```

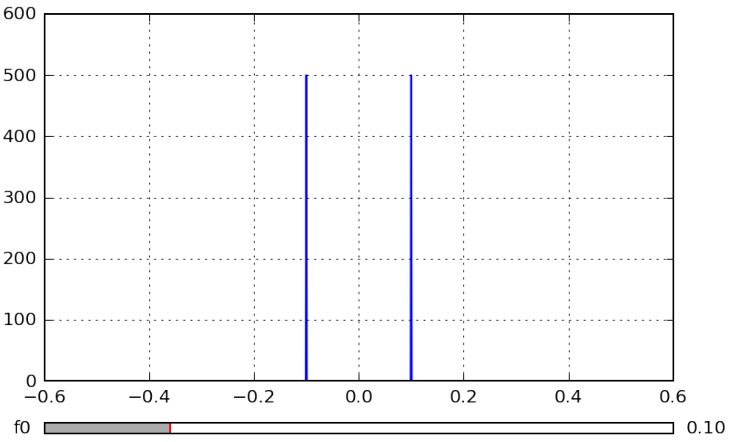



```
%matplotlib
from matplotlib.widgets import Slider

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, "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='on')

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)
```

```
Using matplotlib backend: agg
```

```
0
```




Some definitions

(3.4)

```
%run nbinit.ipy
js_addon()
```

```
... Configuring matplotlib formats
... Configuring matplotlib with inline figures
... Importing numpy as np, scipy as sp, pyplot as plt, scipy.stats as stats
   ... scipy.signal as sig
... Importing widgets, display, HTML, Image, Javascript
... Some LaTeX definitions

... Defining figures captions

... Loading customized Javascript for interactive solutions (show/hide)
```

## 3.3 Symmetries of the Fourier transform.

Consider the Fourier pair

$$x(n) \rightleftharpoons X(f). \tag{3.5}$$

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

$$\boxed{x^*(n) \rightleftharpoons X^*(-f)}. \tag{3.6}$$

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

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

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

$$\boxed{x(-n) \rightleftharpoons X(-f)}. \tag{3.7}$$

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

$$\mathrm{FT}\left\{x(-n)\right\} = \int_{-\infty}^{+\infty} x(-n)\ e^{-j2\pi ft}\ \mathrm{d}t, \tag{3.8}$$

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

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

using the two last emphasized relationships, we obtain

$$\boxed{x^*(-n) \rightleftharpoons X^*(f)}. \tag{3.9}$$

To sum it all up, we have

$$\boxed{\begin{array}{lcl} x(n) & \rightleftharpoons & X(f) \\ x(-n) & \rightleftharpoons & X(-f) \\ x^*(n) & \rightleftharpoons & X^*(-f) \\ x^*(-n) & \rightleftharpoons & X^*(f) \end{array}} \tag{3.10}$$

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

$$\boxed{X(f) = X^*(-f)} \tag{3.11}$$

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

$$\boxed{\begin{array}{llcll} \text{[even]} & x(n) = x(-n) & \rightleftharpoons & X(f) = X(-f) & \text{[even]} \\ \text{[odd]} & x(n) = -x(-n) & \rightleftharpoons & X(f) = -X(-f) & \text{[odd]} \end{array}} \tag{3.12}$$

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 |

(3.13)

Finally, we have

$$\boxed{\begin{array}{lcl} \text{Real even} + \text{imaginary odd} & \rightleftharpoons & \text{Real} \\ \text{Real odd} + \text{imaginary even} & \rightleftharpoons & \text{Imaginary} \end{array}} \tag{3.14}$$

## 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 | $x(n - n_0)$ | $X(f)e^{-j2\pi f n_0}$ |
| 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 | $\sum_{n=-\infty}^{\infty} \lvert x(n)\rvert^2$ | $\int_{[1]} \lvert X(f)\rvert^2 df$ |

(3.15)

Some examples of Fourier pairs are collected below:

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

(3.16)

```
%run nbinit.ipy
js_addon()
```

```
... Configuring matplotlib formats
... Configuring matplotlib with inline figures
... Importing numpy as np, scipy as sp, pyplot as plt, scipy.stats as stats
   ... scipy.signal as sig
... Importing widgets, display, HTML, Image, Javascript
... Some LaTeX definitions

... Defining figures captions

... Loading customized Javascript for interactive solutions (show/hide)
```

# 4 Filters and convolution

## 4.1 Representation formula

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

$$\boxed{x(n) = \sum_{m=-\infty}^{+\infty} x(m)\delta(n-m).} \tag{4.1}$$

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

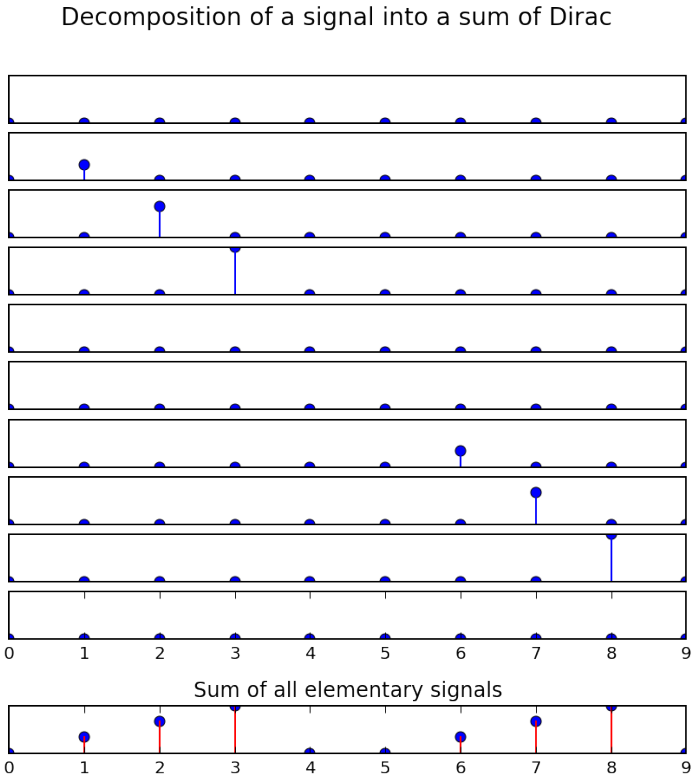
- Since $\delta(n-m) = 1$ if and only if $n = m$, then all the terms in the sum cancel, excepted the one with $n = m$ and therefore we arrive at the identity $x(n) = x(n)$.
- The set of delayed Dirac impulses $\delta(n-m)$ form a basis of the space of discrete signals. Then the coordinate of a signal 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,'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.]
```




## 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.

$$x(m)\delta(n-m) \to x(m)h(n-mu). \tag{4.2}$$

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

$$x(n) = \sum_{m=-\infty}^{+\infty} x(m)\delta(n-m), \tag{4.3}$$

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

$$\boxed{y(n) = \sum_{m=-\infty}^{+\infty} x(m)h(n-m) = [x*h](n).} \tag{4.4}$$

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.

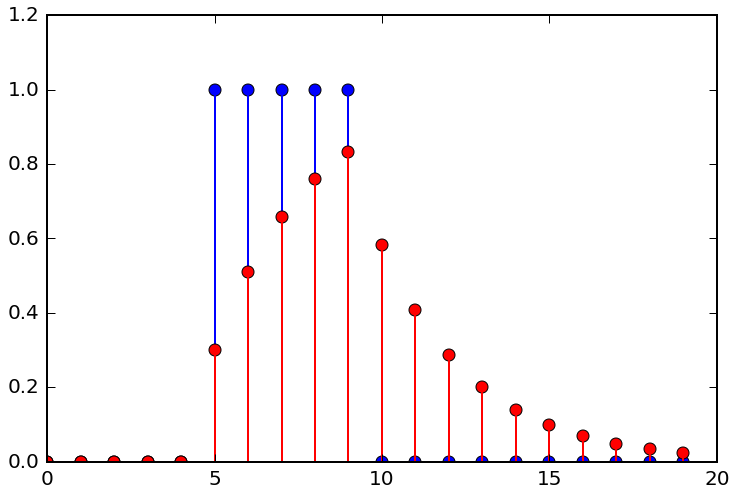
The convolution operation is important since it enables to compute the output of the system using only its impulse response. It is not necessary to know the way the system is build, its internal design and so on. The only thing one must have is its impulse response. Thus we see that the knowledge of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

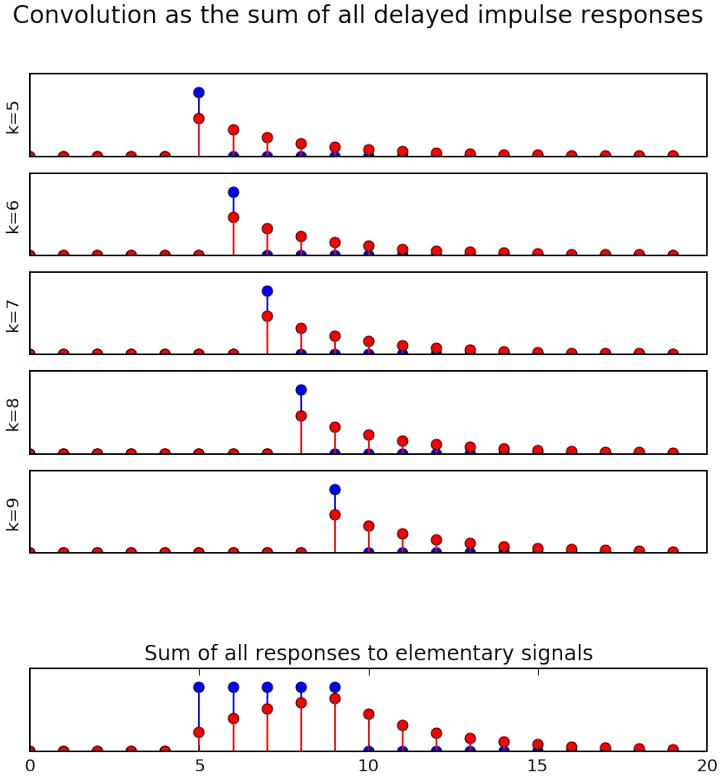
```
def op3(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]= 0.7*transformed_signal[t-1]+0.3*signal[t]
    return transformed_signal
#
# rectangular pulse
N=20; L=5; M=10
r=np.zeros(N)
r[L:M]=1
#
plt.stem(r)
plt.stem(op3(r),linefmt='r-',markerfmt='ro')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



### 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 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)
    #
    ## -> FILL IN
    #
    return z
# test it:
z=myconv(np.ones(L),np.ones(L))
print('z=',z)
```

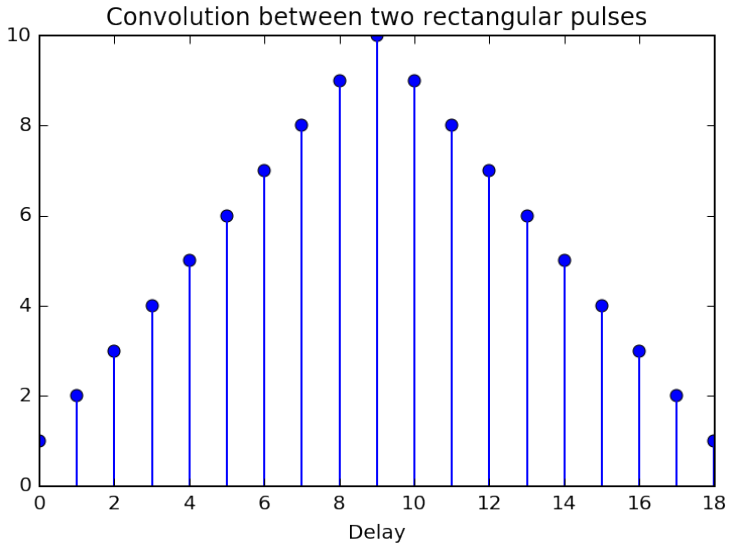
```
z= [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

```
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=10
z=sig.convolve(np.ones(L),np.ones(L))
plt.stem(z)
plt.title("Convolution between two rectangular pulses")
plt.xlabel("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

$$x(n) = X_0 e^{j2\pi f_0 n}. \tag{5.1}$$

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 $f_0$:

$$\boxed{H(f_0) = \sum_m h(m) e^{-j2\pi f_0 m}}. \tag{5.2}$$

Hence, the output can be simply written as

$$y(n) = X_0 e^{j2\pi f_0 n} H(f_0). \tag{5.3}$$

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);

$$x(n) = \int_{[1]} X(f) e^{j2\pi fn} \mathrm{d}f. \tag{5.4}$$

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

$$y(n) = \int_{[1]} X(f)H(f) e^{j2\pi fn} \mathrm{d}f. \tag{5.5}$$

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

$$\boxed{Y(f) = X(f)H(f)}. \tag{5.6}$$

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

$$\boxed{[x{*}y](t) \rightleftharpoons X(f)Y(f)}. \tag{5.7}$$

It is easy to check that reciprocally,

$$\boxed{x(n)y(n) \rightleftharpoons [X{*}Y](f)}. \tag{5.8}$$

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:

$$\begin{aligned}
\mathrm{FT}[x(n)y(n)] &= \sum_n x(n)y(n)e^{-j2\pi fn} && (5.9)\\
&= \sum_n \int X(f_1)e^{j\pi f_1 n}\mathrm{d}f_1 \int Y(f_2)e^{j\pi f_2 n}\mathrm{d}f_2 e^{-j2\pi fn} && (5.10)\\
&= \iint X(f_1)Y(f_2) \sum_n e^{j\pi f_1 n} e^{j\pi f_2 n} e^{-j2\pi fn} \mathrm{d}f_1 \mathrm{d}f_2 && (5.11)
\end{aligned}$$

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

$$\sum_n e^{j\pi f_1 n} e^{j\pi f_2 n} e^{-j2\pi fn} = \delta(f - f_1 - f_2). \tag{5.12}$$

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

$$\mathrm{FT}[x(n)y(n)] = \int X(f_1)Y(f - f_1)\mathrm{d}f_1 = [X * Y](f). \tag{5.13}$$

(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:

$$\boxed{\begin{array}{lcl} [x * y](t) & \rightleftharpoons & X(f)Y(f), \\ x(t)y(t) & \rightleftharpoons & [X * Y](f). \end{array}} \tag{5.14}$$

This theorem has several important consequences.

## 5.2 Consequences

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

$$x(n)y(n)^* \rightleftharpoons \int_{[1]} X(u)Y(u-f)^* \mathrm{d}u, \tag{5.15}$$

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

$$\mathrm{FT}x(n)y(n)^* = \int_{[1]} X(u)Y(u-f)^* \mathrm{d}u, \tag{5.16}$$

that is, for $f = 0$,

$$\boxed{\sum_{-\infty}^{+\infty} x(n)y^*(n) = \int_{[1]} X(u)Y^*(u)\mathrm{d}u}. \tag{5.17}$$

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

$$\boxed{\sum_{-\infty}^{+\infty} |x(n)|^2 = \int_{[1]} |X(f)|^2 \mathrm{d}f}, \tag{5.18}$$

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

(5.19)

(5.20)

(5.21)

# 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

$$y(n) = ay(n-1) + x(n) \tag{6.1}$$

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 $f_0 = 3$, sampled at $Fe = 32$ on $N = 128$ points

2. Fitler 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

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'
#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

$$y(n) = ay(n-1) + x(n) \tag{8.1}$$

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.

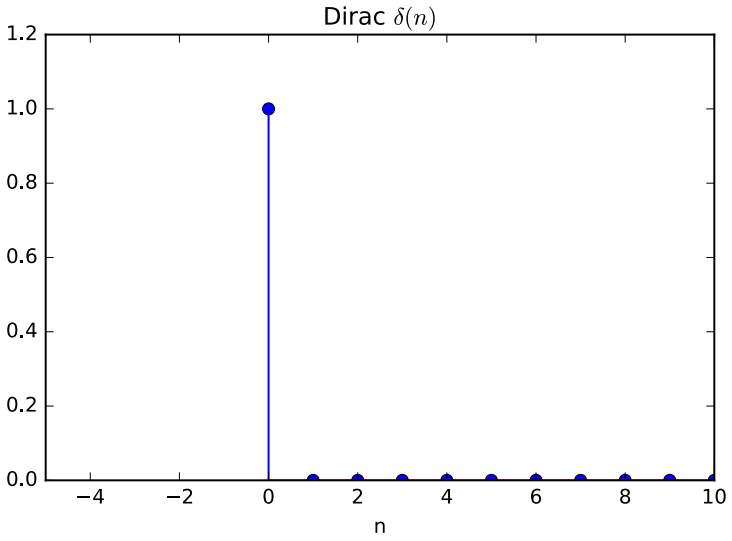
```
from pylab import *
```

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.

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[nb]*x[n-nb]
                         - a[1]*y[n-1] - ... - a[na]*y[n-na]

using the following difference equations::

     y[m] = b[0]*x[m] + z[0,m-1]
     z[0,m] = b[1]*x[m] + z[1,m-1] - a[1]*y[m]
     ...
     z[n-3,m] = b[n-2]*x[m] + z[n-2,m-1] - a[n-2]*y[m]
     z[n-2,m] = b[n-1]*x[m] - a[n-1]*y[m]

where m is the output sample number and n=max(len(a),len(b)) is the
model order.

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

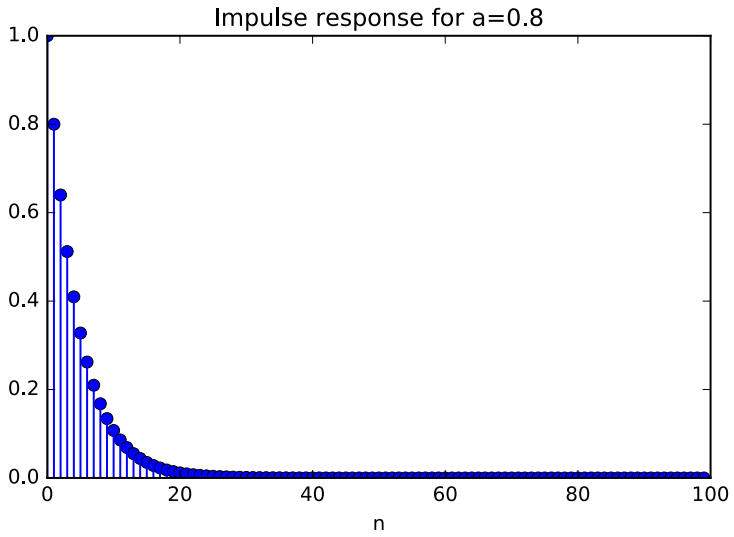
```
                        -1                -nb
                b[0] + b[1]z  + ... + b[nb] z
      Y(z) = ---------------------------------- X(z)
                        -1                -na
                a[0] + a[1]z  + ... + a[na] z
```

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")
```

```
(<matplotlib.text.Text at 0x7f3aa3e34828>,
 <matplotlib.text.Text at 0x7f3aa3e21898>)
```




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 $h(n) = a^n$.

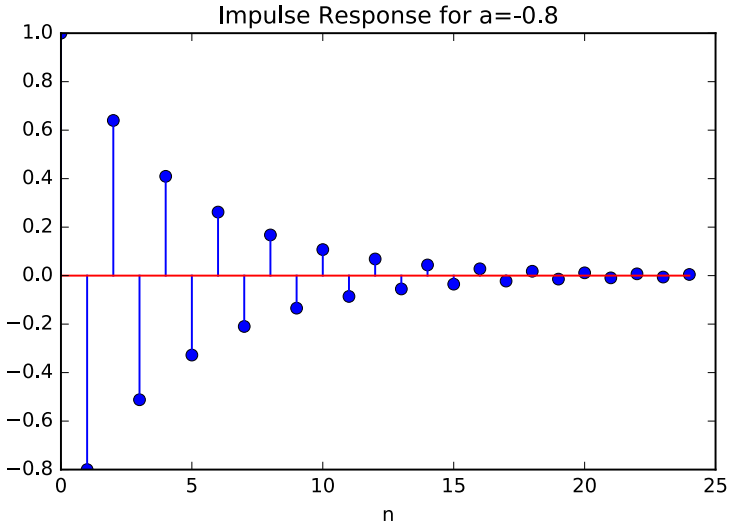
We will check this for some other values of $a$.

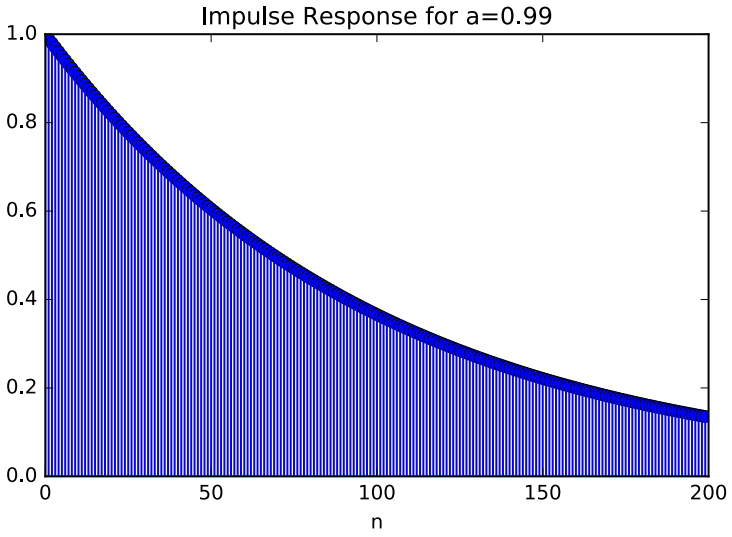
To ease our explorations, we will first define a function that returns the impulse reponse, for two vectors [b] and [a] describing any rational filter. It suffices to compute the filter's output, with a Dirac at its input, on a specified length:

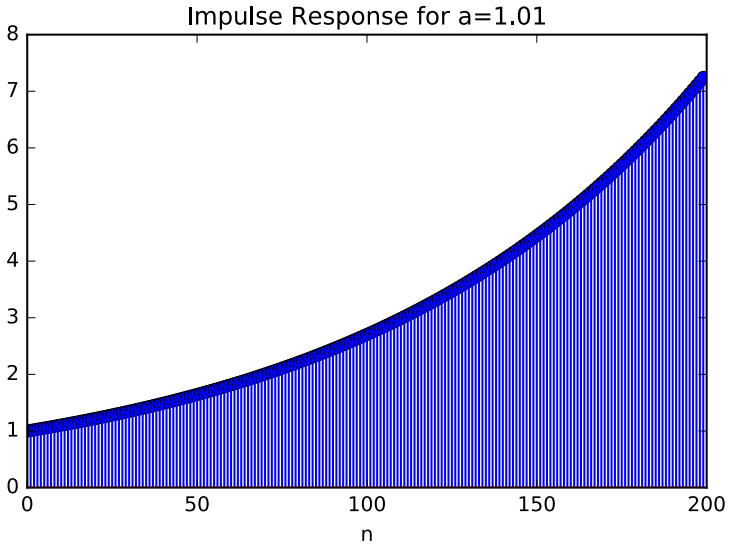
```
def ri(b,a,n):
    """ Returns an impulse response of length n (int)
    of a filter with coefficients a and b
    """
    return lfilter(array(b),array(a),dirac(n))
```

## 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")
```





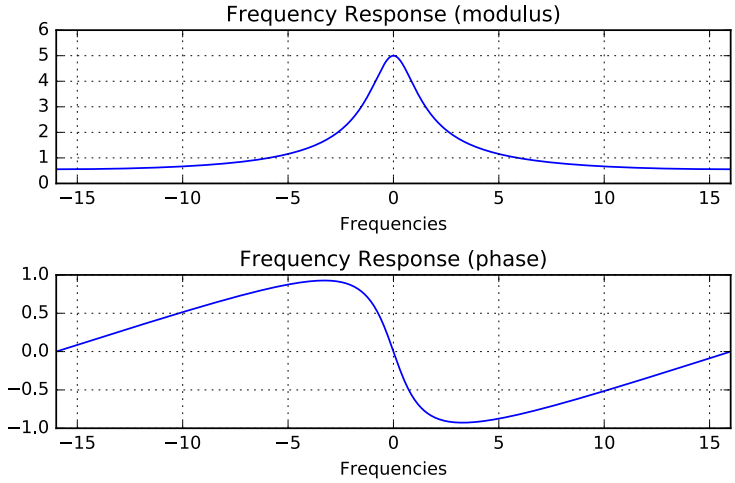

**Conclusions:**

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

## 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);

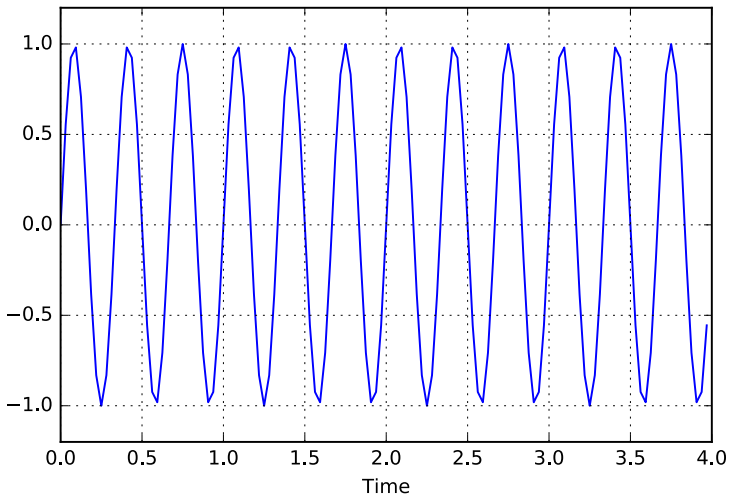
– using a convolution, y2=lfilter(h,1,x); with $h$ the impulse resonse of the filter for $a = 0.8$

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

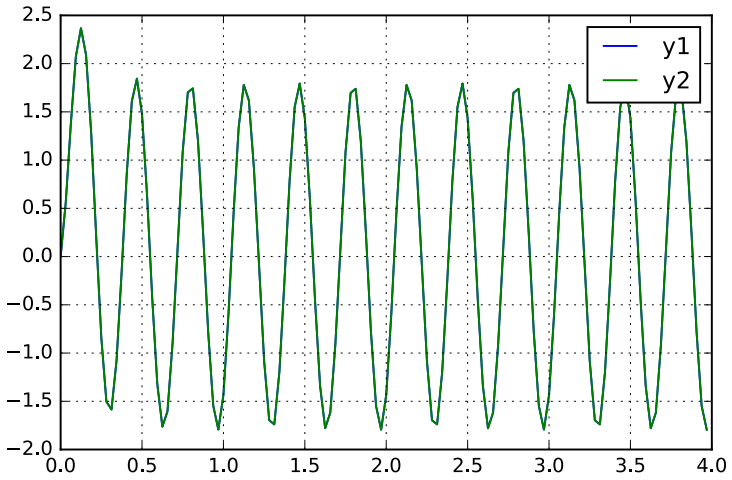
### 8.4.1 Analysis in the time domain

```
# 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)
```

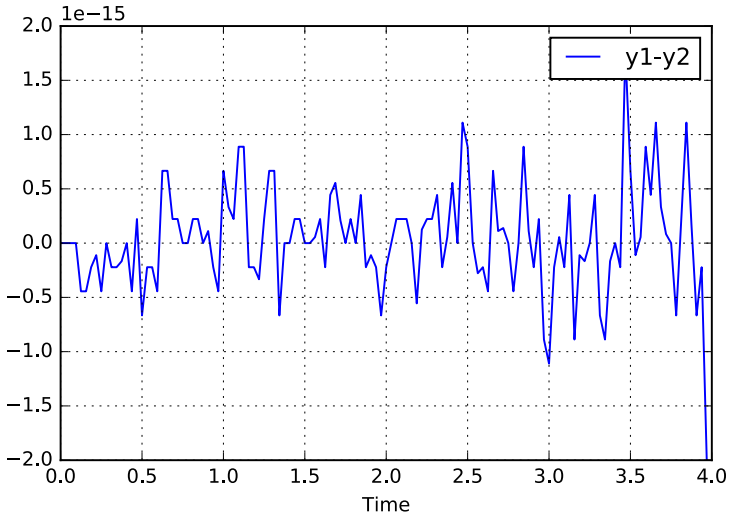



```
# Filtering with filter h
a=0.8
h=ri([1],[1, -a],N) # h computed again, but on N points
y1=lfilter([1],[1, -0.8],x)
y2=lfilter(h,[1],x)
figure()
plot(t,y1,label='y1')
plot(t,y2,label='y2')
grid(b=True)
legend()
show()
```



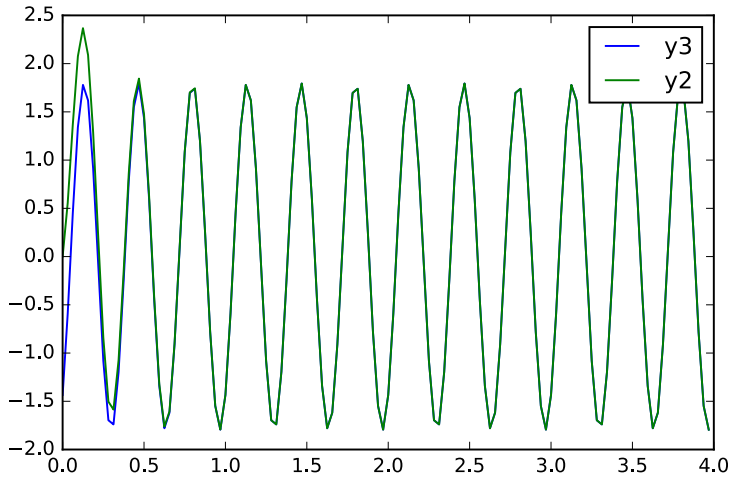

One can also plot the difference between the two signals, so things are clear!

```
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()
```

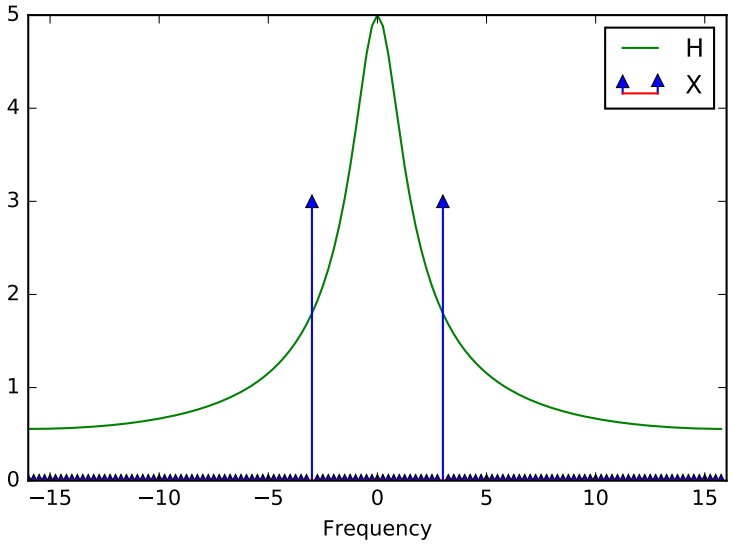



The difference observed at the beginning of the two plots comes from a different assumption on the values of the signals at negative (non observed) times. Actually, function`lfilter` assumes that the signal is zero where non observed, which implies a transient response at the output of the filter. The 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 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()
```



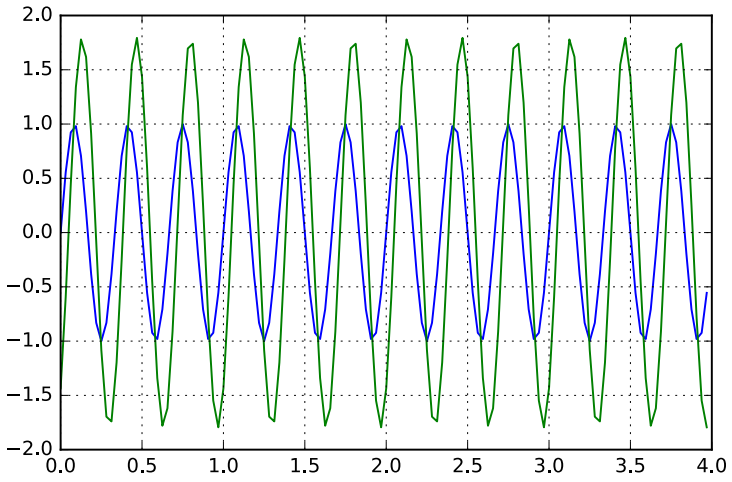

The sine wave has frequency $fo = 3$. let us measure the values of gain and phase at this frequency:

```
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]
```

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

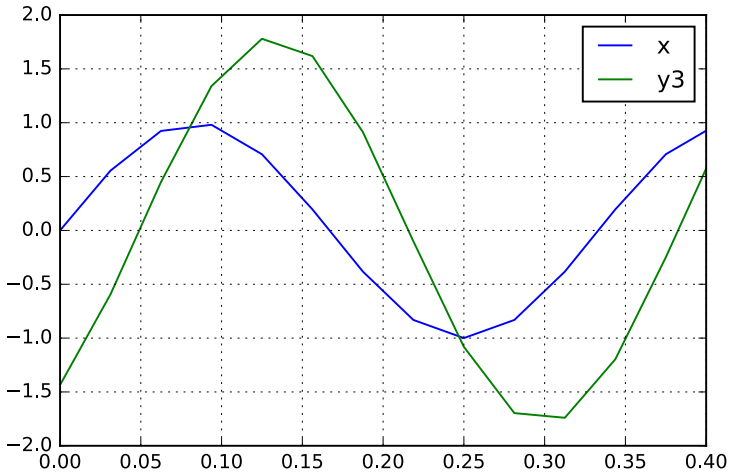
```
figure()
plot(t,x,t,y3)
grid('on')
```



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])
```

```
(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
```

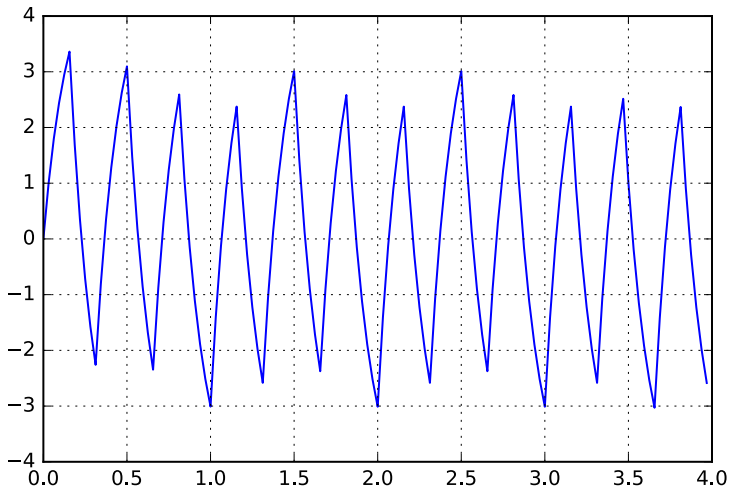
**Observations** : We see that if the input is a sine wave, then the output is **also** a sine wave, up to a gain and phase shift. These gain and phase corresponds exactly to the gain and phase given by the tranfer function.

---

We 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. The sine (and cosine) are the *only* signals which are invariant by linear filtering – they are the eigenfunctions of linear systems. This is a profound justification of the interest of decomposing signals on sine (and cosine) functions; that is of employing the Fourier representation.

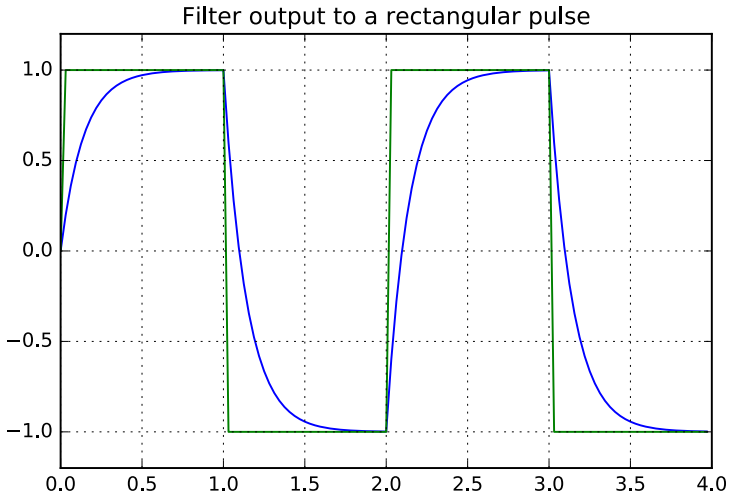
```
def rectpulse(x):
    """rectpulse(x): \n
    Returns a pulse train with period  2pi"""
    return sign(sin(x))
```

```
x=rectpulse(2*pi*fo*t)
y=lfilter(h,[1],x)
figure()
plot(t,y)
grid(b=True)
```




Finally, and for the *fun*, we examin what happens with a larger rectangular pulse. We see in particular the rise and fall time to attain a stable level, like in electrical circuits.

```
f1=0.5
x=rectpulse(2*pi*f1*t)
y=(1-a)*lfilter(h,[1],x)      # Supplementary question: explain why:
                              # we introduce here a factor (1-a)figure()
plot(t,y,label='y')
plot(t,x,label='x')
ylim([-1.2, 1.2])
grid(b=True)
title('Filter output to a rectangular pulse')
```




```
%run nbinit.ipy
```

```
... Configuring matplotlib formats
... Configuring matplotlib with inline figures
... Importing numpy as np, scipy as sp, pyplot as plt, scipy.stats as stats
   ... scipy.signal as sig
```

```
... Importing widgets, display, HTML, Image, Javascript
... Some LaTeX definitions

... Defining figures captions

... Loading customized Javascript for interactive solutions (show/hide)
```

# 9 The continuous time case

- The continuous time case
  \begin{itemize}
- The continuous time Fourier transform

```
\begin{itemize}
\itemsep1pt\parskip0pt\parsep0pt
\item
  \hyperref[Definition]{Definition}
\item
  \hyperref[Example----The-Fourier-transform-of-a-rectangular-pulse]{Example
  - The Fourier transform of a rectangular pulse}
\item
  \hyperref[Table-of-Fourier-transform-properties]{Table of Fourier
  transform properties}
\item
  \hyperref[Symmetries-of-the-Fourier-transform.]{Symmetries of the
  Fourier transform.}
```

Dirac impulse, representation formula and convolution

## 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

$$x(t) = \int_{-\infty}^{+\infty} X(f)\ e^{j2\pi ft}\ \mathrm{d}f, \tag{9.1}$$

where

$$X(f) = \int_{-\infty}^{+\infty} x(t)\ e^{-j2\pi ft}\ \mathrm{d}t. \tag{9.2}$$

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

$$\int_{-\infty}^{+\infty} |x(t)| \, \mathrm{d}t < +\infty. \tag{9.3}$$

Indeed, if $x(t)$ is absolutely integrable, then

$$\int_{-\infty}^{+\infty} |x(t) \; e^{-j2\pi ft}| \, \mathrm{d}t < \int_{-\infty}^{+\infty} |x(t)| \, \mathrm{d}t < +\infty \tag{9.4}$$

(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* $\mathrm{rect}_T(t)$ *the rectangular pulse defined by*

$$\mathrm{rect}_T(t) = \begin{cases} 1 & \text{if } t \in [-T/2, T/2], \\ 0 & \text{elsewhere.} \end{cases} \tag{9.5}$$

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

$$X(f) = \mathrm{FT}\left\{A\mathrm{rect}_T(t)\right\} = A\int_{-T/2}^{T/2} e^{-j2\pi ft}\mathrm{d}t, \tag{9.6}$$

*that is*

$$X(f) = A\left[\frac{e^{-j2\pi ft}}{-j2\pi f}\right]_{-\frac{T}{2}}^{\frac{T}{2}} = A\frac{1}{j2\pi f}\left[e^{j\pi fT} - e^{-j\pi fT}\right] \tag{9.7}$$

*so that finally*

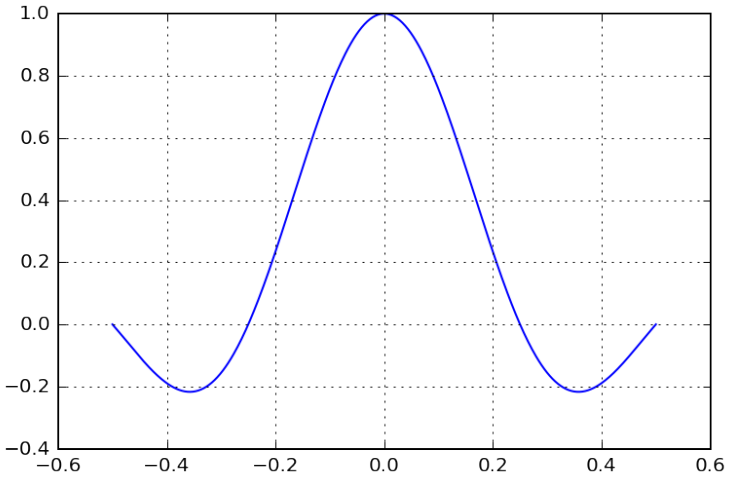
$$\boxed{X(f) = AT\frac{\sin(\pi fT)}{\pi fT} \triangleq AT\mathrm{sinc}\,(\pi fT).} \tag{9.8}$$

*where* $\mathrm{sinc}\,(.)$ *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* $\mathrm{sinc}\,(\pi fT)$ *vanishes for* $\pi fT = k\pi$*, that is for* $f = k/T$*; except for* $k = 0$*, since* $\mathrm{sinc}\,(x) = 1$ *for* $x \to 0$.

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

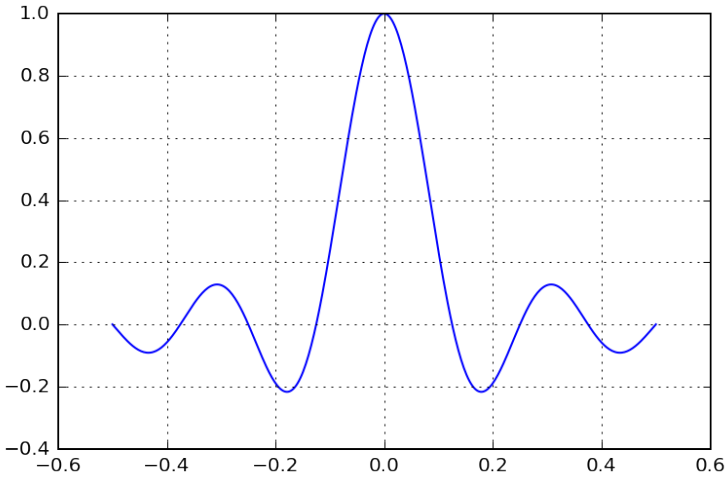
```
%matplotlib inline
def sinc(x):
    if isinstance(x,(int,float)): x=[x]
    x=np.array(x)
    out=np.ones(np.shape(x))
    I=np.where(x!=0)
    out[I]=np.sin(x[I])/x[I]
    return out
```

```
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*4*f))
plt.grid(b='on')
```




Playing with values using a slider.

```
N=1000
f=np.linspace(-0.5,0.5,400)
def pltsinc(value, T):
    clear_output(wait=True)
    plt.plot(f,sinc(pi*T*f))
    plt.grid(b='on')
s=widgets.FloatSlider(min=0, max=20, step=0.1, value=8)
pltsinc('Width',8)
s.on_trait_change(pltsinc,'value')
display(s)
#alternatively
#interact(pltsinc,value=fixed(1), T=[0.1,10,0.1])
```




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

$$\int_0^{+\infty} \mathrm{sinc}\,(x)\mathrm{d}x = \frac{\pi}{2}, \tag{9.9}$$

the symmetry of $\mathrm{sinc}\,()$, and a change of variable, we obtain that

$$\int_{-\infty}^{+\infty} T\mathrm{sinc}\,(\pi fT)\mathrm{d}f = 1. \tag{9.10}$$

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, $T\mathrm{sinc}\,(\pi fT)$ 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 $T\mathrm{sinc}\,(\pi fT)$ 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 $\delta$ function we had in discrete time.It is called *Dirac distribution* (or function) and we end with the following pair

$$1 \rightleftharpoons \delta(f) \tag{9.11}$$

- 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 (9.8), we obtain that the Fourier transform of a Dirac function is a unit constant

$$\delta(t) \rightleftharpoons 1 \tag{9.12}$$

```
%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)
```

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

ImportError                               Traceback (most recent call last)

<ipython-input-4-c72328a46101> in <module>()
----> 1 get_ipython().magic('matplotlib tk')
      2 from matplotlib.widgets import Slider
      3
      4 fig, ax = plt.subplots()
      5 fig.subplots_adjust(bottom=0.2, left=0.1)


/usr/local/lib/python3.5/site-packages/IPython/core/interactiveshell.py in magic(self, a
   2334         magic_name, _, magic_arg_s = arg_s.partition(' ')
   2335         magic_name = magic_name.lstrip(prefilter.ESC_MAGIC)
-> 2336         return self.run_line_magic(magic_name, magic_arg_s)
   2337
   2338     #-------------------------------------------------------------------------


/usr/local/lib/python3.5/site-packages/IPython/core/interactiveshell.py in run_line_magi
   2255                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2256             with self.builtin_trap:
-> 2257                 result = fn(*args,**kwargs)
   2258             return result
   2259


/usr/local/lib/python3.5/site-packages/IPython/core/magics/pylab.py in matplotlib(self,


/usr/local/lib/python3.5/site-packages/IPython/core/magic.py in <lambda>(f, *a, **k)
```

```
    191     # but it's overkill for just that one bit of state.
    192     def magic_deco(arg):
--> 193         call = lambda f, *a, **k: f(*a, **k)
    194
    195         if callable(arg):


    /usr/local/lib/python3.5/site-packages/IPython/core/magics/pylab.py in matplotlib(self,
     98             print("Available matplotlib backends: %s" % backends_list)
     99         else:
--> 100             gui, backend = self.shell.enable_matplotlib(args.gui)
    101             self._show_matplotlib_backend(args.gui, backend)
    102


    /usr/local/lib/python3.5/site-packages/IPython/core/interactiveshell.py in enable_matplo
   3130                 gui, backend = pt.find_gui_and_backend(self.pylab_gui_select)
   3131
-> 3132         pt.activate_matplotlib(backend)
   3133         pt.configure_inline_support(self, backend)
   3134


    /usr/local/lib/python3.5/site-packages/IPython/core/pylabtools.py in activate_matplotlib
    273
    274     import matplotlib.pyplot
--> 275     matplotlib.pyplot.switch_backend(backend)
    276
    277     # This must be imported last in the matplotlib series, after


    /usr/local/lib/python3.5/site-packages/matplotlib/pyplot.py in switch_backend(newbackend
    222     matplotlib.use(newbackend, warn=False, force=True)
    223     from matplotlib.backends import pylab_setup
--> 224     _backend_mod, new_figure_manager, draw_if_interactive, _show = pylab_setup()
    225
    226


    /usr/local/lib/python3.5/site-packages/matplotlib/backends/__init__.py in pylab_setup()
     30     # imports. 0 means only perform absolute imports.
     31     backend_mod = __import__(backend_name,
---> 32                              globals(),locals(),[backend_name],0)
     33
     34     # Things we pull in from all backends


    /usr/local/lib/python3.5/site-packages/matplotlib/backends/backend_tkagg.py in <module>(
```

```
      4
      5 from matplotlib.externals import six
----> 6 from matplotlib.externals.six.moves import tkinter as Tk
      7 from matplotlib.externals.six.moves import tkinter_filedialog as FileDialog
      8


    /usr/local/lib/python3.5/site-packages/matplotlib/externals/six.py in __get__(self, obj,
     88
     89     def __get__(self, obj, tp):
---> 90         result = self._resolve()
     91         setattr(obj, self.name, result) # Invokes __set__.
     92         try:


    /usr/local/lib/python3.5/site-packages/matplotlib/externals/six.py in _resolve(self)
    111
    112     def _resolve(self):
--> 113         return _import_module(self.mod)
    114
    115     def __getattr__(self, attr):


    /usr/local/lib/python3.5/site-packages/matplotlib/externals/six.py in _import_module(nam
     78 def _import_module(name):
     79     """Import module, returning the module after the last dot."""
---> 80     __import__(name)
     81     return sys.modules[name]
     82


    /usr/local/lib/python3.5/tkinter/__init__.py in <module>()
     33 import sys
     34
---> 35 import _tkinter # If this fails your Python may not be configured for Tk
     36 TclError = _tkinter.TclError
     37 from tkinter.constants import *


    ImportError: No module named '_tkinter'
```

### 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 | $\int_{-\infty}^{\infty} \lvert x(t)\rvert^2 \mathrm{d}t$ | $\int_{-\infty}^{\infty} \lvert X(f)\rvert^2 \,\mathrm{d}f.$ |

(9.13)

**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:*

$$x(t-t_0) \rightleftharpoons X(f)e^{-j2\pi f t_0}. \tag{9.14}$$

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

$$\mathrm{FT}\left\{x(t-t_0)\right\} = \int_{-\infty}^{+\infty} x(t-t_0)\; e^{-j2\pi f t}\; \mathrm{d}t; \tag{9.15}$$

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

$$\mathrm{FT}\left\{x(t-t_0)\right\} = \int_{-\infty}^{+\infty} x(t-t_0) e^{-j2\pi f(t-t_0)} e^{-j2\pi f t_0} \mathrm{d}t, \tag{9.16}$$

that is

$$\mathrm{FT}\left\{x(t-t_0)\right\} = e^{-j2\pi f t_0} \int_{-\infty}^{+\infty} x(t-t_0) e^{-j2\pi f(t-t_0)} \mathrm{d}t = e^{-j2\pi f t_0} X(f). \tag{9.17}$$

□

### 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 |

(9.18)

(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

$$\delta(t) = \begin{cases} 0 & \text{if } t \neq 0, \\ +\infty & \text{for } t = 0, \end{cases} \tag{9.19}$$

and is such that

$$\int_{-\infty}^{+\infty} \delta(t)\mathrm{d}t = 1. \tag{9.20}$$

### 9.2.2 Representation formula

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

$$x(t)\delta(t - t_0) = x(t_0)\delta(t - t_0). \tag{9.21}$$

Consequently,

$$\int_{-\infty}^{+\infty} x(t)\delta(t - t_0)\mathrm{d}t = x(t_0)\int_{-\infty}^{+\infty} \delta(t - t_0)\mathrm{d}t = x(t_0). \tag{9.22}$$

Therefore, we always have

$$\begin{cases} x(t) = \int_{-\infty}^{+\infty} x(\tau)\delta(t - \tau)\mathrm{d}\tau \\ \text{with } x(\tau) = \int_{-\infty}^{+\infty} x(t)\delta(t - \tau)\mathrm{d}t. \end{cases} \tag{9.23}$$

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

$$x(\tau) =< x(t), \delta_\tau(t) >= \int_{-\infty}^{+\infty} x(t)\delta(t - \tau)\mathrm{d}t, \tag{9.24}$$

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

$$x(t) = \int_{-\infty}^{+\infty} x(\tau)\delta(t - \tau)\mathrm{d}\tau. \tag{9.25}$$

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

$$\boxed{y(t) = \int_{-\infty}^{+\infty} x(\tau)h(t - \tau)\mathrm{d}\tau = [x * h](t).} \tag{9.26}$$

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

$$x(n) = \sum_{n=0}^{L_0-1} c_k e^{j2\pi k f_0 n}, \tag{10.2}$$

where $f_0 = 1/L_0$ and

$$c_k = \frac{1}{L_0} \sum_{[L_0]} x_{L_0}(n) e^{-j2\pi k f_0 n}. \tag{10.3}$$

From this relation, we immediately have

$$c_k = \frac{1}{L_0} X_{L_0}(k f_0), \tag{10.4}$$

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

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

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

$$\mathrm{FT}\left\{x(n)\right\} = \frac{1}{L_0} \sum_{k=0}^{L_0-1} X_{L_0}(k f_0) \mathrm{FT}\left\{e^{j2\pi k f_0 n}\right\}, \tag{10.6}$$

that is

$$\boxed{X(f) = \mathrm{FT}\{x(n)\} = \frac{1}{L_0}\sum_{k=0}^{L_0-1} X_{L_0}(kf_0)\delta(f-kf_0)}. \tag{10.7}$$

### 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*:

$$\boxed{\sum_{m=-\infty}^{+\infty} \delta(n-mL_0) = \frac{1}{L_0}\sum_{k=0}^{L_0-1} e^{j2\pi k f_0 n}}. \tag{10.8}$$

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

$$w_{L_0}(n) = \sum_{m=-\infty}^{+\infty} \delta(n-mL_0). \tag{10.9}$$

Taking the Fourier transforms of the two sides of (10.8), we obtain

$$\boxed{\sum_{m=-\infty}^{+\infty} e^{j2\pi f m L_0 n} = \frac{1}{L_0}\sum_{k=0}^{L_0-1} \delta(f-kf_0)}; \tag{10.10}$$

that is the second *Poisson's formula*:

$$\boxed{\sum_{m=-\infty}^{+\infty} \delta(n-mL_0) \rightleftharpoons \frac{1}{L_0}\sum_{k=0}^{L_0-1} \delta(f-kf_0)}. \tag{10.11}$$

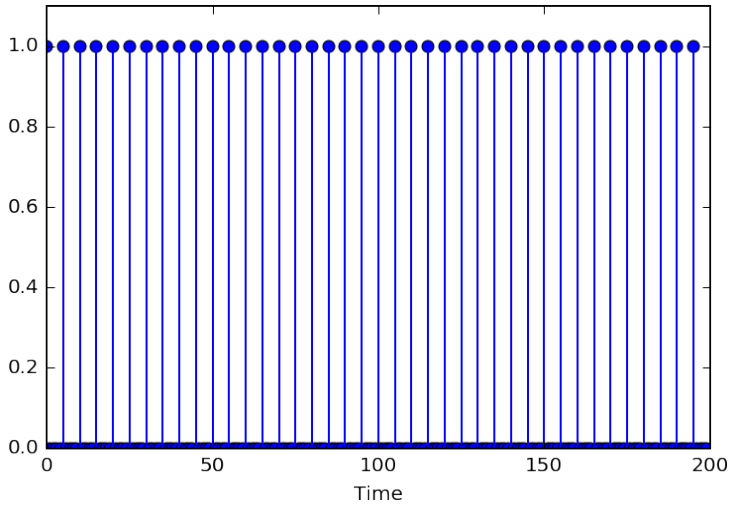
This last relation shows that the Fourier transform of a Dirac comb is also a Dirac comb, these two combs having an inverse spacing.

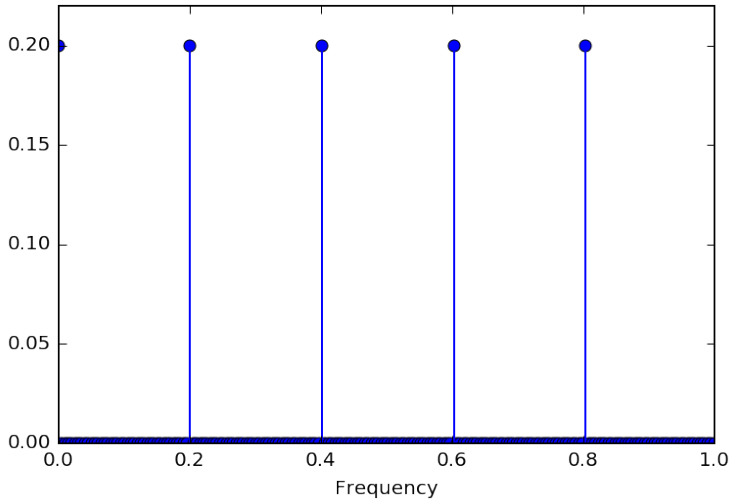
**Exercise 4.** *Let us check this numerically. This is very easy: define a Dirac comb, take its Fourier transform using the* `fft` *function, and look at the result.*

```
## 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")
```




We may now go back to the exploration of the links between Fourier series and transform, using the second Poisson formula (10.11).

**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

$$[x * \delta_{n_0}](n) = \sum_m x(m)\delta_{n_0}(n-m) \tag{10.12}$$
$$= \sum_m x(m)\delta(n-m-n_0) \tag{10.13}$$
$$= x(n-n_0) \tag{10.14}$$
$$\tag{10.15}$$

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:

$$x(n) = [x_L * w_{L_0}](n) = \left[x_L * \sum_k \delta_{kL_0}\right](n) \tag{10.16}$$
$$= \sum_k [x_L * \delta_{kL_0}](n) \tag{10.17}$$
$$= \sum_k x_L(n-kL_0). \tag{10.18}$$
$$\tag{10.19}$$

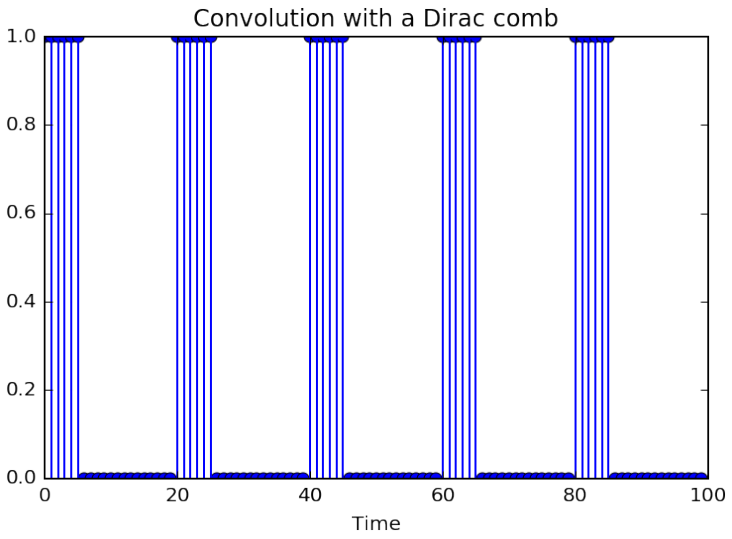
This is nothing but the expression of a periodic signal. If $L_0$ is larger than the support $L$ of $x_L(n)$, then $x(n)$ is simply the repetition, with a period $L_0$, of the pattern $x_L(n)$.

*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')
```




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,

$$x(n) = [x_L * w_{L_0}](n) \rightleftharpoons X_L(f).\mathrm{FT}\{w_{L_0}(n)\}. \tag{10.20}$$

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

$$x(n) = [x_L * w_{L_0}](n) \rightleftharpoons X_L(f).\frac{1}{L_0} w_{\frac{1}{L_0}}(f), \tag{10.21}$$

or

$$X(f) = X_L(f).\frac{1}{L_0} w_{\frac{1}{L_0}}(f) = \frac{1}{L_0}\sum_k X_L(kf_0)\delta(f - kf_0), \tag{10.22}$$

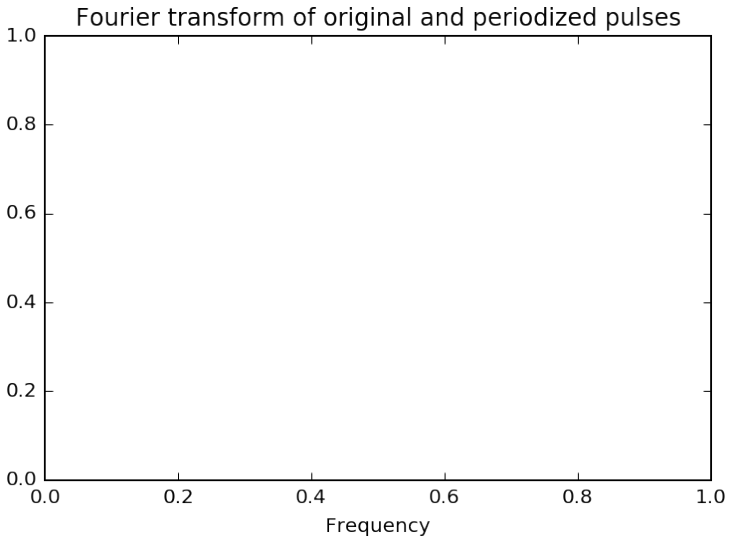
with $f_0 = 1/L_0$. We see that the Fourier transform of the periodized signal is the product of the Fourier transform of the initial pattern with a Dirac comb in frequency. Hence, periodization in the time domain results in a discretization of the frequency axis, yielding a Fourier transform constituted 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.*

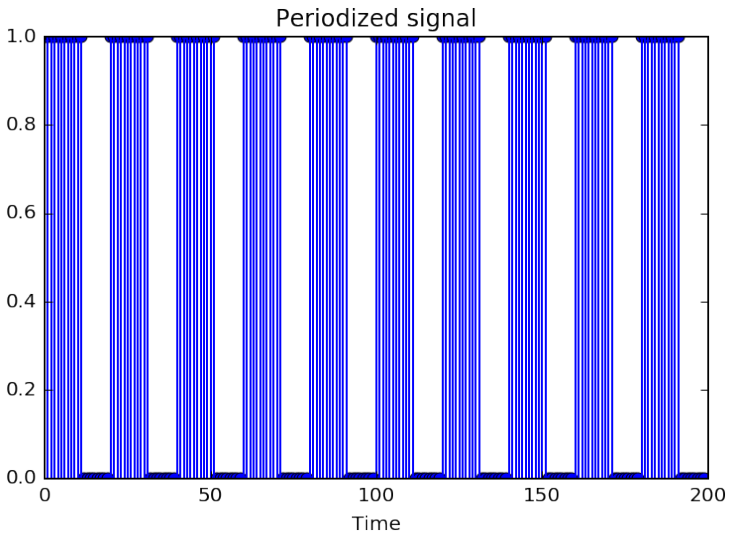
**Exercise 6.** *Continue the exercise 5 by an analysis of what happens in the Fourier domain: compute the Fourier transforms of the original and periodized signals and compare them on the same plot. The Fourier transform of the periodized signal should be computed* without *zero padding, ie exactly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

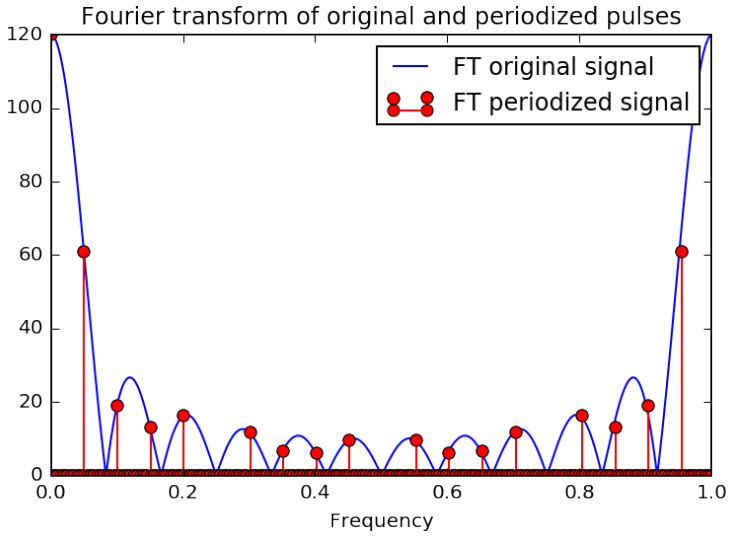
*You will have to introduce a factor to account for the fact that there is more signal in the periodized one yan in the initial - the factor to consider is simply the number of periods.*

```
#
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

$$\boxed{\begin{aligned} x(n) &= \sum_{k=0}^{N-1} X(k) e^{j2\pi \frac{kn}{N}} \\ \text{with } X(k) &= \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{kn}{N}}. \end{aligned}} \tag{10.23}$$

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 $n \in [0, N)$,

$$x(n) = [x_0 * w_N](n) \tag{10.24}$$

and

$$\begin{aligned} X(f) &= X_0(f).\frac{1}{N} w_{\frac{1}{N}}(f) && (10.25)\\ &= X_0(f).\frac{1}{N}\sum_{k=0}^{N-1} \delta\left(f - \frac{k}{N}\right) && (10.26)\\ &= \frac{1}{N}\sum_{k=0}^{N-1} X_0\left(\frac{k}{N}\right)\delta\left(f - \frac{k}{N}\right) && (10.27)\\ & && (10.28) \end{aligned}$$

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

$$\begin{aligned} x(n) &= \int_{[1]} X(f) e^{j2\pi f n} \mathrm{d}f && (10.29)\\ &= \frac{1}{N}\sum_{k=0}^{N-1} X_0\left(\frac{k}{N}\right)\int_{[1]} e^{j2\pi f n}\delta\left(f - \frac{k}{N}\right)\mathrm{d}f && (10.30)\\ &= \frac{1}{N}\sum_{k=0}^{N-1} X_0\left(\frac{k}{N}\right) e^{j2\pi\frac{kn}{N}} && (10.31) \end{aligned}$$

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

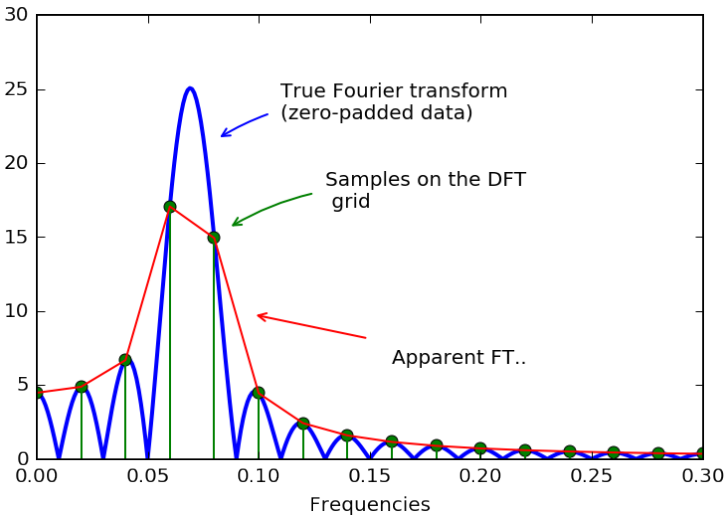
$$\begin{aligned} X_0\left(f = \frac{k}{N}\right) &= \sum_{n=-\infty}^{+\infty} x_0(n) e^{-j2\pi\frac{kn}{N}} && (10.32)\\ &= \sum_{n=0}^{N-1} x_0(n) e^{-j2\pi\frac{kn}{N}} && (10.33)\\ & && (10.34) \end{aligned}$$

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 (10.23) 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])
```

```
(0, 0.3)
```




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

$$\mathbf{x} = \begin{bmatrix} x(0) \\ x(1) \\ \vdots \\ x(N-1) \end{bmatrix} = x(0) \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + x(1) \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} + \ldots + x(N-1) \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}. \tag{10.35}$$

The vectors of complex exponentials

$$\mathbf{e_k} = \frac{1}{\sqrt{N}} \left[ 1, e^{-j2\pi \frac{k}{N}}, \ldots, e^{-j2\pi \frac{kl}{N}}, \ldots, e^{-j2\pi \frac{k(N-1)}{N}} \right]^T \tag{10.36}$$

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

$$\mathbf{F} = \left[ \mathbf{e_0}, \mathbf{e_1}, \ldots \mathbf{e_{N-1}} \right] \tag{10.37}$$

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

$$\mathbf{X} = \mathbf{F}^+ \mathbf{x} \tag{10.38}$$

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

$$\mathbf{x} = \mathbf{F}\mathbf{X}. \tag{10.39}$$

Developing line $k$ of (10.38), we obtain

$$X(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{kn}{N}}, \tag{10.40}$$

and developing line $n$ of (10.39), we obtain

$$x(n) = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X(k) e^{j2\pi \frac{kn}{N}}. \tag{10.41}$$

Up to a simple factor (let eg $X'(k) = \frac{1}{\sqrt{N}} X(k)$) we recover the formulas (10.23) of the DFT.

### 10.2.3 Time-shift property

... to be completed

### 10.2.4 Circular convolution

... to be completed

```
%run nbinit.ipy
```

```
... Configuring matplotlib formats
... Configuring matplotlib with inline figures
... Importing numpy as np, scipy as sp, pyplot as plt, scipy.stats as stats
   ... scipy.signal as sig
... Importing widgets, display, HTML, Image, Javascript
... Some LaTeX definitions

... Defining figures captions

... Loading customized Javascript for interactive solutions (show/hide)
```

## 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':

$$X_s(f) = \sum_n x_s(n) e^{-j2\pi fn}. \tag{10.42}$$

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

$$X_s(f) = \sum_k x_d(k) e^{-j2\pi f k N_0}. \tag{10.43}$$

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

$$X_d(f) = X_s\left(\frac{f}{N_0}\right). \tag{10.44}$$

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

$$x_s(n) = x(n).w_{N_0}(n). \tag{10.45}$$

By Plancherel's theorem, we have

$$X_s(f) = \left[X * \mathrm{FT}\left\{w_{N_0}(n)\right\}\right](f) \tag{10.46}$$

$$= \left[X * \frac{1}{N_0} w_{\frac{1}{N_0}}\right](f) \tag{10.47}$$

$$= \frac{1}{N_0} \sum_k \left[X * \delta_{\frac{k}{N_0}}\right](f) \tag{10.48}$$

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

$$\boxed{X_s(f) = \frac{1}{N_0} \sum_k X(f - \frac{k}{N_0}).} \tag{10.49}$$

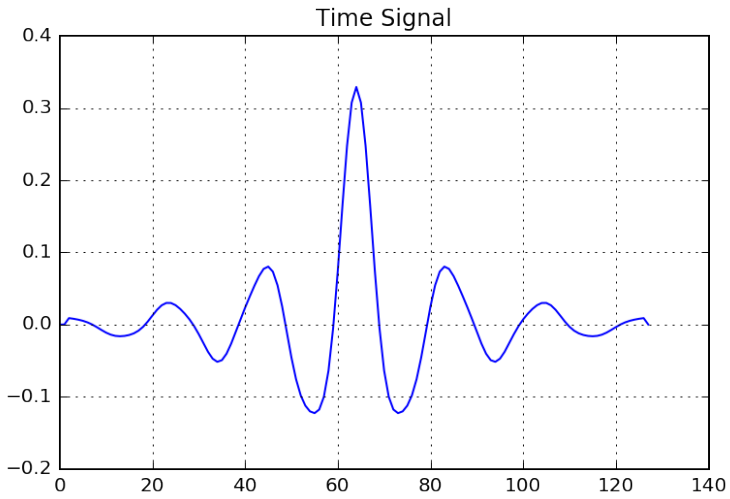
This is a fundamental result:

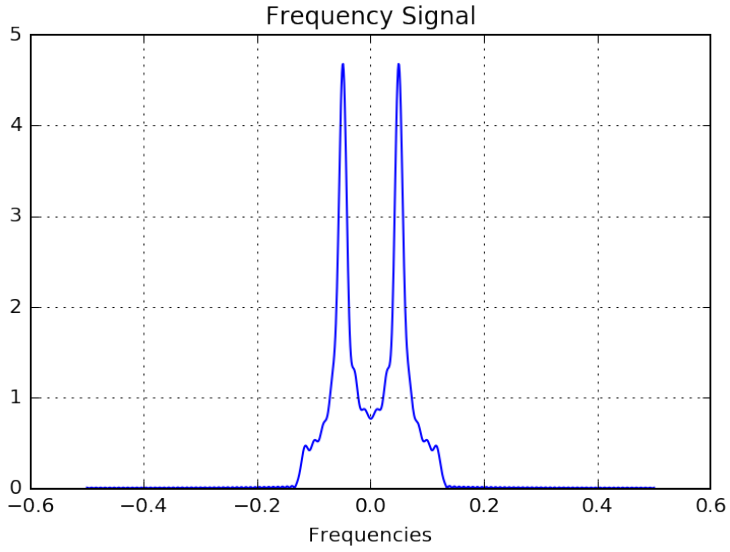
*Sampling in the time domain yields periodization in the frequency domain.*

Let us illustrate this on our test signal:

```
%matplotlib inline
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('on')
plt.figure()
#plot frequency signal
xf=fftshift(fft(x,M))
```

```
plot(f,abs(xf))
title('Frequency Signal')
xlabel('Frequencies')
plt.grid('on')
```



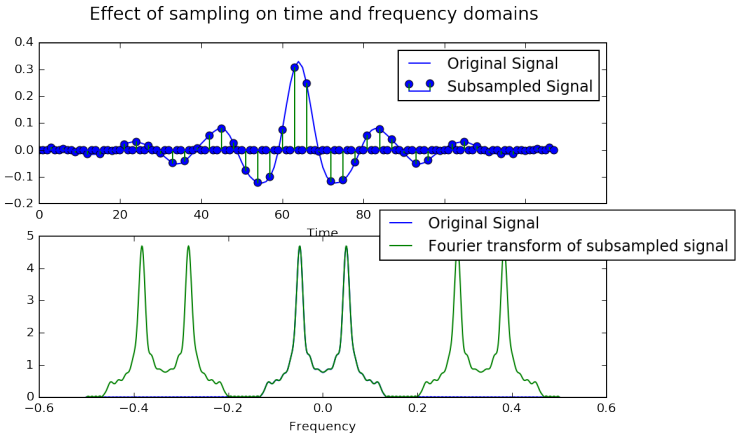




We first define a subsampler function, that takes for argument the signal $x$ and the subsampling factor $k$.

```
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)
```

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

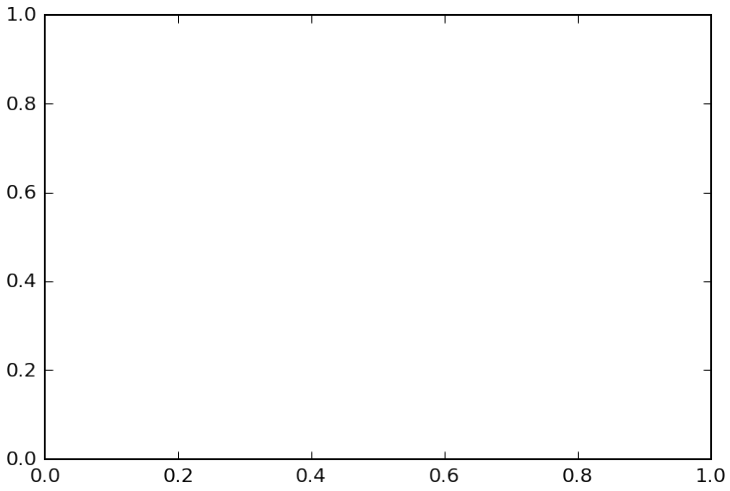
```
def sampling_experiment(val,k):
    fig,bx=plt.subplots(2,1,figsize=(8,5))
    clear_output(wait=True)
    bx[0].plot(t,x, label='Original Signal')
    (xs,xsf)=subsampb(x,k,M)
    bx[0].stem(t,xs,linefmt='g-',markerfmt='bo', basefmt='b-',label='
        Subsampled Signal')
    bx[0].set_xlabel('Time')
    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()

display(slide_k)
sampling_experiment('',3)
slide_k.on_trait_change(sampling_experiment,'value')
```




matplotlib external figure version:

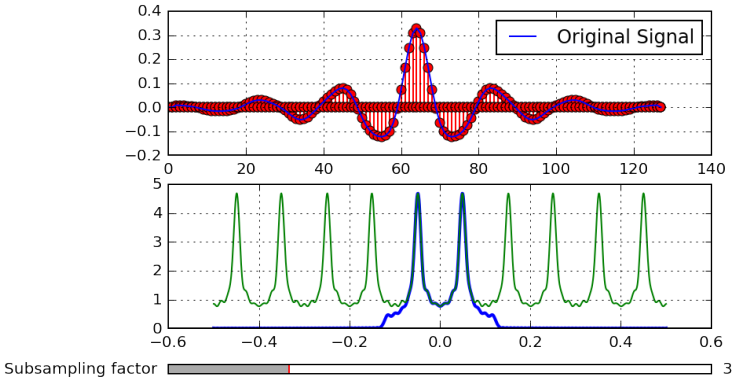
```
%matplotlib
def plt_stem(t,x,*args, ax=gca(), **kwargs):
    xx=zeros(3*len(x))
    xx[1:-1:3]=x
    xx=xx[:3*len(x)]
    tt=np.repeat(t,3)
    out=ax.plot(tt,xx,*args, **kwargs)
    return out
#plt_stem(t,x,'-or')
```

```
Using matplotlib backend: agg
```




```
%matplotlib
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='on')
ax[1].grid(b='on')

def on_change(k):
    k=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)
```

```
Using matplotlib backend: agg
```




## 10.4 The sampling theorem

### 10.4.1 Derivation in the case of discrete-time signals

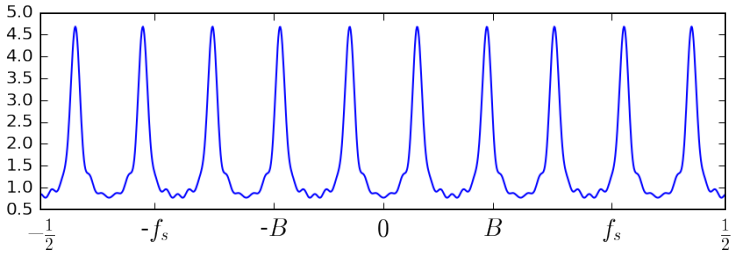
As a consequence, we will obtain a sufficient condition for the reconstruction of the original signal from its samples. Assume that $x(n)$ **is a real band-limited signal with a maximum frequency** $B$.

$$X(f) = 0 \text{ for } |f| > B \tag{10.50}$$

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.

$$X_s(f) = f_s \sum_k X(f - kf_s). \tag{10.51}$$

```
%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 $[-f_s/2, f_s/2]$:

$$\begin{cases} H(f) = 1 & \text{for } |f| < f_s/2 \\ H(f) = 0 & \text{for } |f| > f_s/2 \end{cases} \tag{10.52}$$

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

$$X_s(f).H(f) = f_s X(f) \tag{10.53}$$

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

$$x(n) = T_s[x_s * h](n), \tag{10.54}$$

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 $f_s$:

$$h(n) = \int_{[1]} \text{rect}_{f_s}(f) e^{j2\pi fn} \mathrm{d}f \tag{10.55}$$

$$= \int_{-\frac{f_s}{2}}^{\frac{f_s}{2}} e^{j2\pi fn} \mathrm{d}f \tag{10.56}$$

$$= f_s \frac{\sin(\pi f_s n)}{\pi f_s n} \tag{10.57}$$

In developed form, the convolution then expresses as

$$\boxed{x(n) = \sum_{k=-\infty}^{+\infty} x(kT_s) \frac{\sin(\pi f_s (n - kT_s))}{\pi f_s (n - kT_s)}.} \tag{10.58}$$

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*

$$f_s > 2B. \tag{10.59}$$

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

$$x(n) = \sum_{k=-\infty}^{+\infty} x(kT_s) \frac{\sin(\pi f_s (n - kT_s))}{\pi f_s (n - kT_s)}. \tag{10.60}$$

### 10.4.2 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:

$$X_s(f) = f_s \sum_k X(f - kf_s). \tag{10.61}$$

Aliasing is avoided if the sampling frequency $f_s$ is such that

$$f_s > 2B. \tag{10.62}$$

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

$$x(t) = \sum_{k=-\infty}^{+\infty} x(kT_s) \frac{\sin(\pi f_s(t - kT_s))}{\pi f_s(t - kT_s)}. \tag{10.63}$$

### 10.4.3 Illustrations

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

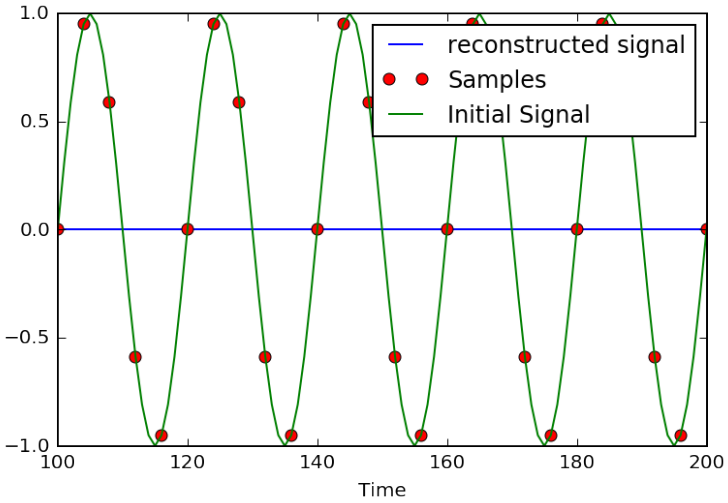
$$x(n) = \sum_{k=-\infty}^{+\infty} x(kT_s) \frac{\sin(\pi f_s(n - kT_s))}{\pi f_s(n - kT_s)}. \tag{10.64}$$

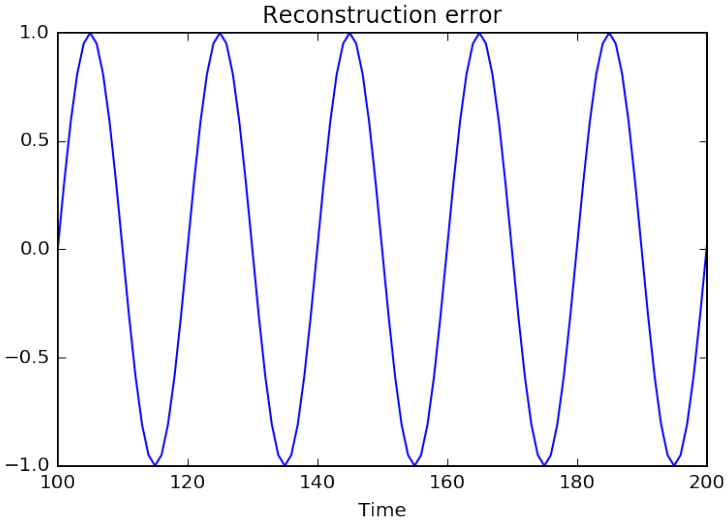
*In order to do that, you will first create a sinusoid with frequency $f_0$ (eg $f_0 = 0.05$). You will sample this sine wave at 4 samples per period ($f_s = 4f_0$). Then, you will implement the interpolation formula and will compare the approximation (finite number of terms in the sum) to the intial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

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")
```

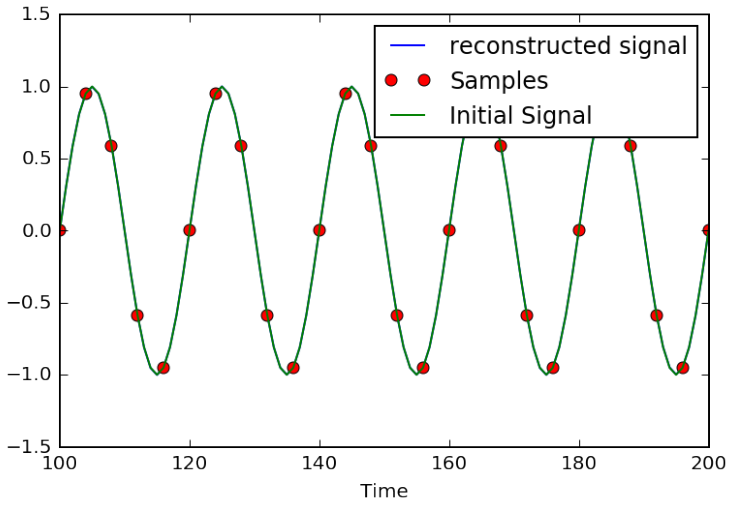
```
plt.plot(ts,xs,'ro',label="Samples")
plt.plot(t,x,'-g',label="Initial Signal")
plt.xlabel("Time")
plt.xlim([100,200])
plt.legend()

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

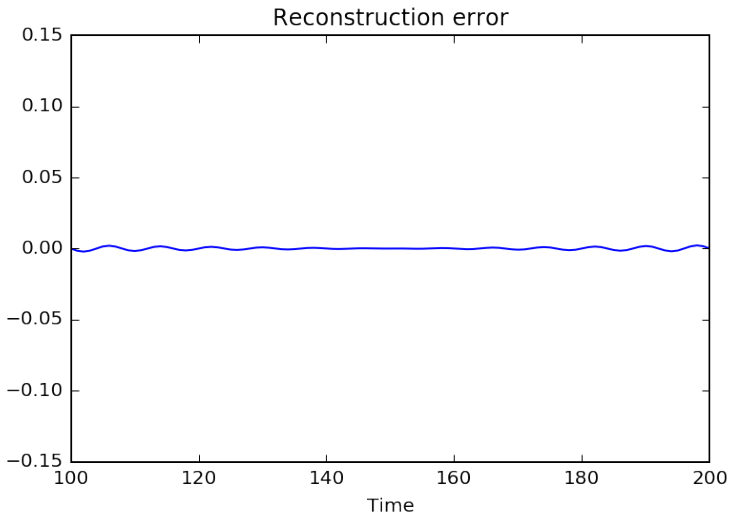



```
N=300
t=np.arange(N)
fo=0.05 #--> 1/fo=20 samples per period
x=sin(2*pi*fo*t)
ts=np.arange(0,N,4) # 5 samples per period
num=np.size(ts) # number of samples
xs=x[::4] #downsampling, 5 samples per period
```

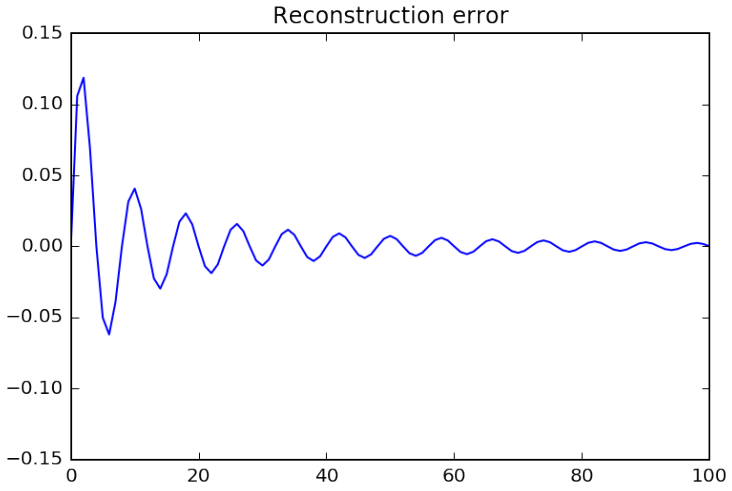
```
Ts,Fs=4,1/4
x_rec=zeros(N) #reconstructed signal
for k in range(num):
    x_rec=x_rec+xs[k]*np.sinc(Fs*(t-k*Ts)) #! The sinc includes the pi

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 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.

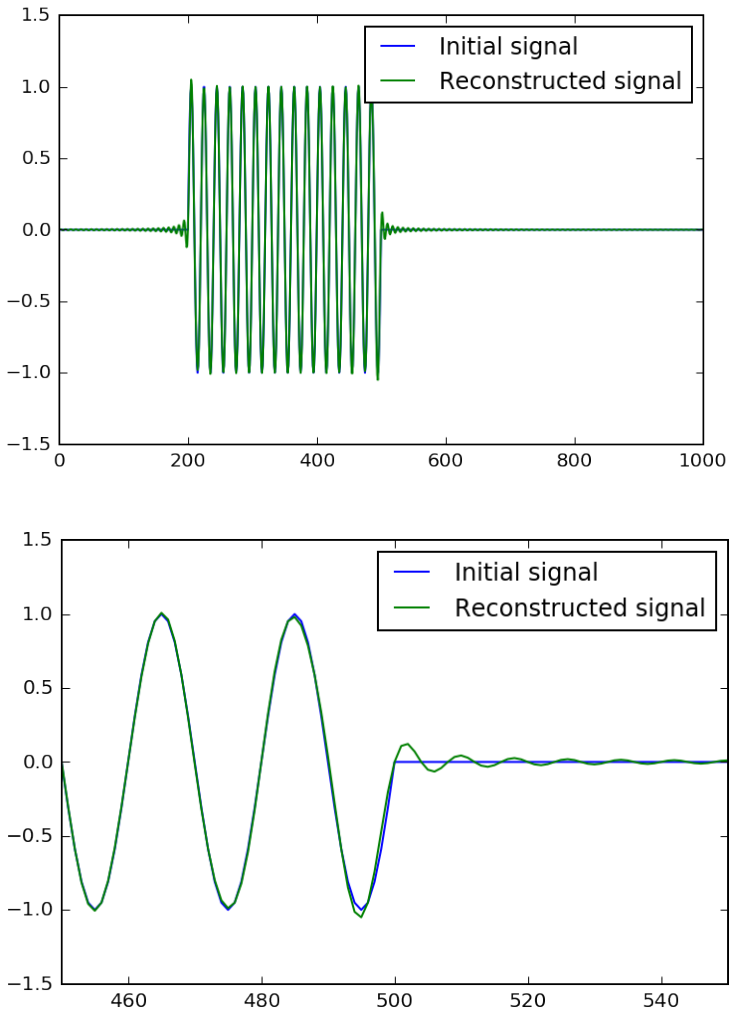
**Analysis of the aliasing due to time-limited support.**

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()
```

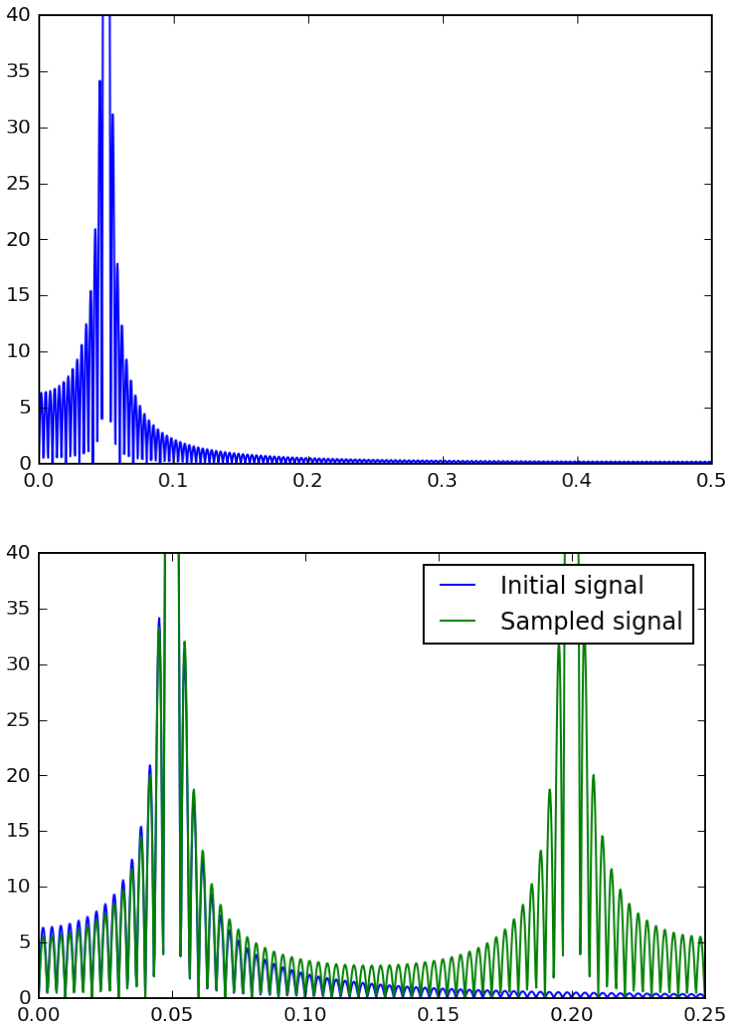
```
plt.figure()
plt.plot(x_extended,label="Initial signal")
plt.plot(t,x_rec,label="Reconstructed signal")
plt.xlim([450,550])
_=plt.legend()
```




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 $f_s/2 = 0.125$.

### 10.4.4 Sampling of band-pass signals

to be continued...

## 10.5 Lab on basics in image processing

### 10.5.1 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.

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.

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 `%matplotlib`. To return to the inline mode, use `%matplotlib inline`.

```
#This was for Python v. 2.7
#from __future__ import division
#from __future__ import print_function

import matplotlib.pyplot as plt
import numpy as np
from numpy import abs, where, sin, cos, ones, zeros, log, real, imag
from scipy import ndimage as ndi
from scipy import signal as sig
from numpy.fft import fft, ifft, fft2, ifft2, fftshift
from scipy.ndimage import convolve as convolvend
# For 3D representations
from mpl_toolkits.mplot3d.axes3d import Axes3D

# change the dpi for a better vizualisation of images
import matplotlib
savedpi=matplotlib.rcParams['savefig.dpi']
%matplotlib inline
```

Listing of offered utility functions

```
# from defs_tp_img import *
#
## Definitions
# ===========

def isodd(num):
        return num & 1 and True or False
# or num%2==1

##
def rect2(L,N):
    """ Returns a (2L+1)x(2L+1) rectangle centered over a
    NxN matrix"""
    x=np.zeros((N,N))
    center=(round((N+1)/2),round((N+1)/2)) if isodd(N) else (round(N/2+1),
        round(N/2+1))
```

```
    x[center[0]-L:center[0]+L+1,center[1]-L:center[1]+L+1]=1
    return x


##
def bandpass2d(position,half_width,total_dim):
    """ Returns the frequency response of a bandpass filter, correctly
        centered on
    `position`, of half-width `half_width` and symmeterized in frequency
        domain
    (zero at the center of the resulting matrix)
    Parameters
    _________
    position: array
        center-frequency of the band-pass response
        (in samples -- the corresponding frequencies are 2*position/
            total_dim)
    half_width: integer
         half frequency band
    total_dim: integer
          image dimension
    Author : jfb"""
    N=total_dim
    center=(round((N+1)/2),round((N+1)/2)) if isodd(N) else (round(N/2+1),
        round(N/2+1))
    #
    H=zeros((N,N))
    H[center[0]+position[0]-half_width:center[0]+position[0]+half_width+1,
        center[1]+position[1]-half_width:center[1]+position[1]+half_width
        +1]=1
    position=-np.array(position)  # symmetric frequencies
    H[center[0]+position[0]-half_width:center[0]+position[0]+half_width+1,
        center[1]+position[1]-half_width:center[1]+position[1]+half_width
        +1]=1
    return H

## 2D fequency representation
def showfft2(X,Zero='uncentered', Freq='normalized', num=None,cmap='hot'):
    """Display an image, in the 2D Fourier domain
    The representation is centered and in normalized frequencies.
    **/!\\   *showfft2* **does not compute the Fourier transform**.
    The user is responsible for supplying correct data.
    Parameters:
    _________
        X : 2D data in the Fourier domain
        Num: Figure number(optional)
        Zero='uncentered' (default), the input Fourier transform are
            supposed uncenterd and a
        fftshift is applied
        Freq='normalized' (default); otherwise axis are in samples
    ``Author: jfb``
    """
    N,M=np.shape(X)
    plt.figure(num)
    if Zero!='centered':
        X=fftshift(X)
    if Freq=='normalized':
        extent=(-0.5, 0.5, -0.5, 0.5)
    else:
        extent=(-N/2, N/2, -N/2, N/2)
```

```
    im=plt.imshow(X,aspect='auto',origin='lower', extent=extent, cmap=cmap
        )
    if Freq=='normalized':
        ticks=np.arange(-0.5,0.6,0.1)
        plt.xticks(ticks)
        plt.yticks(ticks)
    plt.colorbar()



def mesh(obj3D,numfig=None,subplt=(1,1,1),cmap='hot'):
    """
    Emulates Matlab's mesh
    Author: jfb
    """
    fig = plt.figure(numfig)
    ax = fig.add_subplot(subplt[0],subplt[1],subplt[2], projection='3d')
    m,n=np.shape(obj3D)
    mm=np.arange(m)
    nn=np.arange(n)
    X,Y=np.meshgrid(nn,mm)
    ax.plot_surface(X, Y, obj3D, rstride=max([round(m/50), 1]),
                    cstride=max([round(n/50), 1]), cmap=cmap)

def saltpepper(percent=85,shape=None):
        s=np.random.random(size=shape)
        d=np.where(s>percent/100)
        out=np.ones(shape=shape)
        out[d]=0
        return out
```

### 10.5.2 Frequency representation – Filtering in the frequency domain

**A pretty sine wave**

- Generate a sine wave $f(x,y) = \sin(2\pi(f_x x + f_y y))$ , with $f_x$ an $f_y$ between 0.02 à 0.2, on $N \times N$ points (e.g. $N = 512$). For the implementation, you may use a `for` loop, or a double comprehension list, or even the smart function `fromfunction()` and an anonymous `lambda` function. Display the result using `imshow`.

Note that for a matrix, usually the first index denotes the rows and the second the columns. Therefore modifying something with respect to x results in a variation along the rows of the matrix (usually the "y" axis)..

```
?np.fromfunction
```

```
fx,fy= 0.02, 0.01
#
# DO IT YOURSELF...
#
# Z=
#
#
plt.imshow(Z,cmap='hot',origin='upper')
_=plt.xlabel('y')
_=plt.ylabel('x')
```

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

<ipython-input-4-9a5744efda6a> in <module>()
      6 #
      7 #
----> 8 plt.imshow(Z,cmap='hot',origin='upper')
      9 _=plt.xlabel('y')
     10 _=plt.ylabel('x')


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

- Transform your program into a function `pltsin` which takes the two frequencies as inputs. Then implement a interactive version, using the lines

```
from IPython.html.widgets import interact, interactive, fixed
interact(pltsin,fx=[-0.1,0.1,0.01], fy=[-0.1,0.1,0.01])
```

Then play with the parameters, and look at what happens when you change the values of the frequencies.

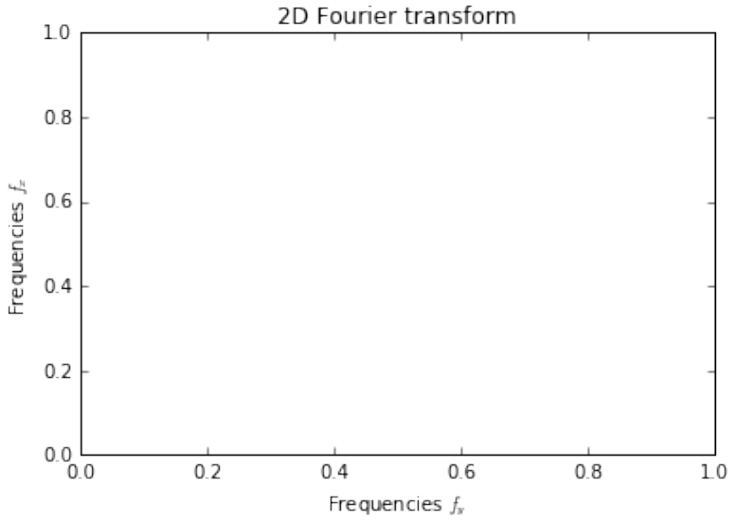
```
from IPython.html.widgets import interact, interactive, fixed
def pltsin(fx,fy):
# DO IT YOURSELF...
#
#
interact(pltsin,fx=[-0.1,0.1,0.01], fy=[-0.1,0.1,0.01])
```

```
  File "<ipython-input-5-c5ca0b1f9ad7>", line 6
interact(pltsin,fx=[-0.1,0.1,0.01], fy=[-0.1,0.1,0.01])
       ^
IndentationError: expected an indented block
```

- Compute the 2D Fourier transform of $f(x, y)$ (using function fft2) and display the modulus (function abs) of the result, via `showfft2`. What are the significations of the axes? What are the spatial frequencies of the sine wave. Experiment with several values of $f_x, f_y$.

```
# We change the frequencies for ease of visualization in the frequency
    domain
fx,fy= 0.1, 0.2
#S=
```

```
# Display
#
plt.title('2D Fourier transform')
plt.xlabel('Frequencies $f_y$')
plt.ylabel('Frequencies $f_x$')
```




- Show theoretically, and then check it numerically, that the 2D Fourier transform can be obtained as the succession of two 1-dimensional transforms, applied respectively to the rows and then to the columns (or vice versa). You will take advantage of the fact that the standard `fft` has a parameter axis which is the axis over which the `fft` is actually computed.

```
# DO IT!
#
plt.title('2D-Fourier transform as the succession of two 1D transforms')
plt.xlabel('Frequencies $f_y$')
plt.ylabel('Frequencies $f_x$')
```

### 10.5.3 Playing with Barbara – Filtering in the frequency domain

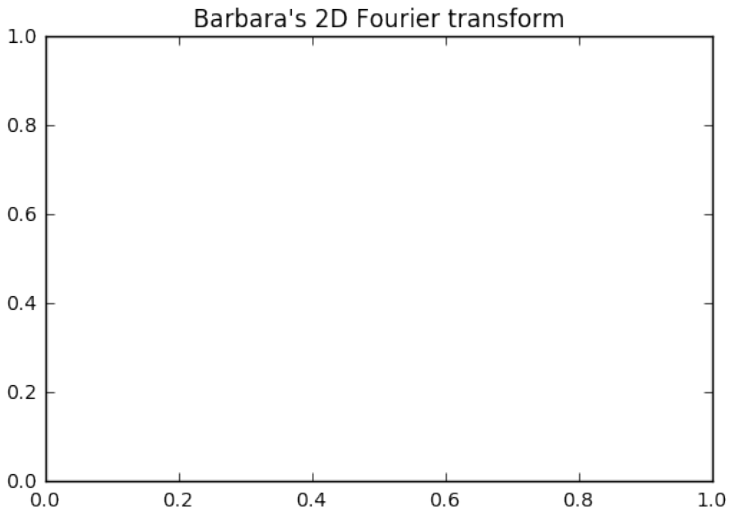
*Rappelle-toi Barbara Il pleuvait sans cesse sur Brest ce jour-là Et tu marchais souriante Épanouie ravie ruisselante Sous la pluie* (Prévert, 1946)

- Load the image Barbara, using `plt.imread(barbara.png)` and display it (in gray levels).

```
matplotlib.rcParams['savefig.dpi'] = 2*savedpi # double dpi <-- This is to
    increase resolution
#
#%% reading and displaying the image
#B=
```

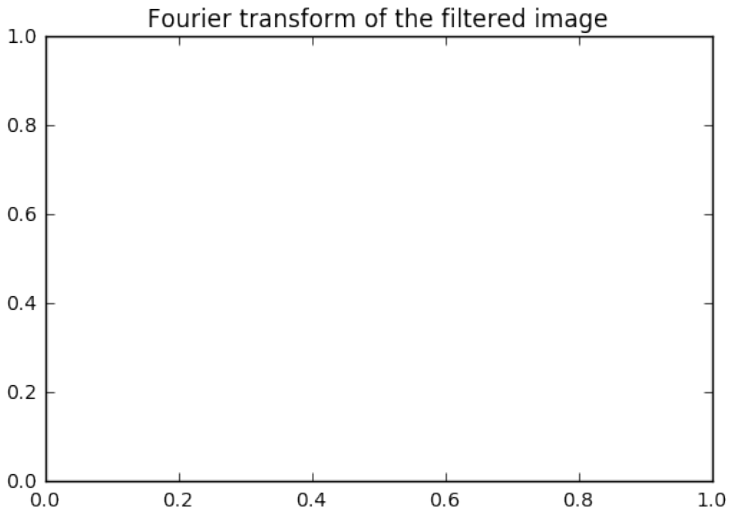
- Display the corresponding frequency representation `showfft2` , in log scale (simply take `log(abs())`!)

```
matplotlib.rcParams['savefig.dpi'] = savedpi #restore dpi
# Do it!
plt.title("Barbara's 2D Fourier transform")
```



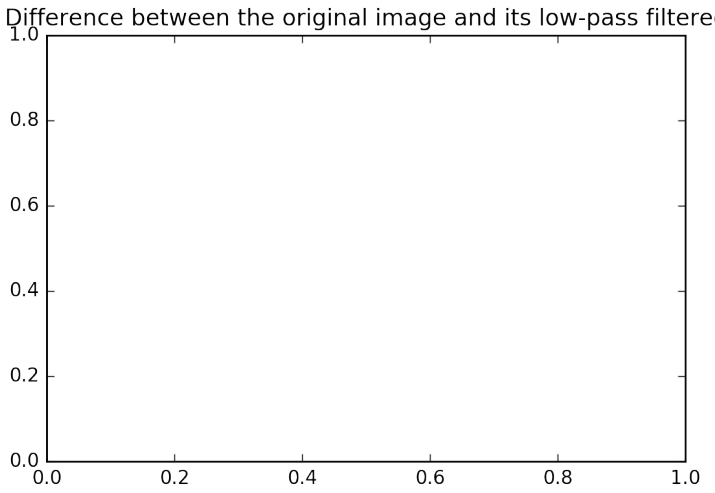

- Filter this image using a low-pass filter with a rectangular frequency response (use the function `rect2`), for rectangles of half-width 40, 80 ,100. Display the different resulting images, as well as the differences with the initial image. Observations, conclusions.

```
L=40
#
#
#
#
plt.title('Fourier transform of the filtered image')
```




The corresponding results in the spatial domain are:

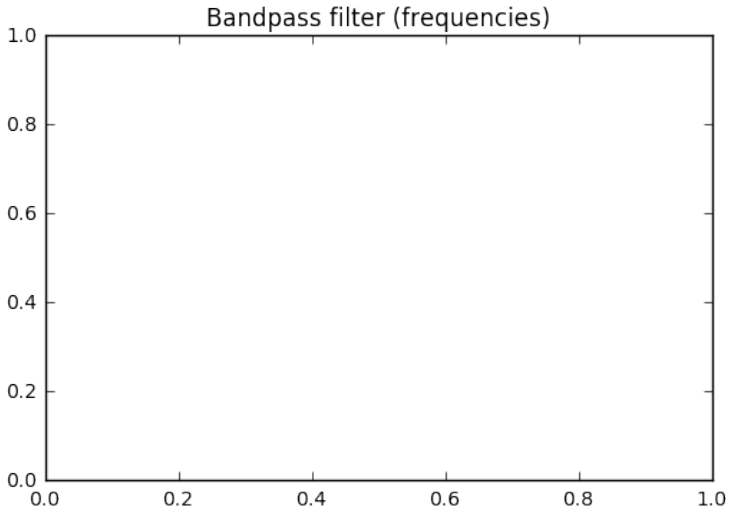
```
matplotlib.rcParams['savefig.dpi'] = 2*savedpi
#And in the spatial domain
#
#
plt.title("Filtered image (spatial domain)")
# Also experiment with L=80, L=100
# Differences :
#
#
plt.title("Difference between the original image and its low-pass filtered
    ")
```



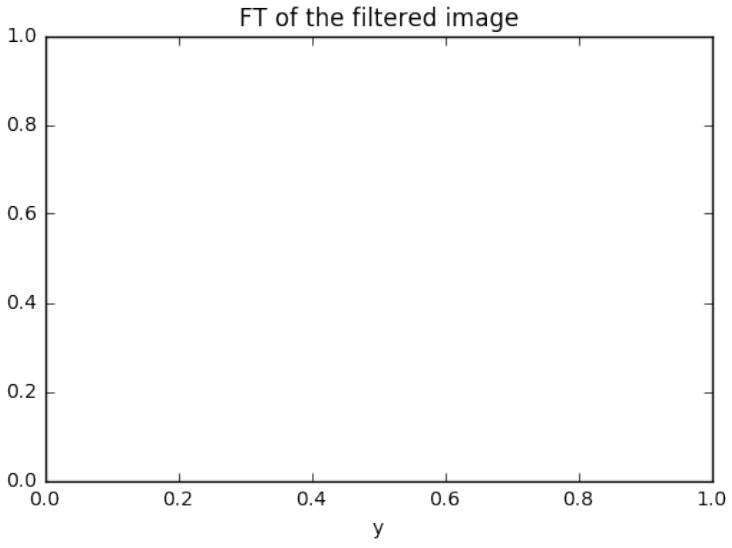

- design a frequency response that kills selectively the frequencies around points (46,54) and (-70,79), e.g.˜on a neighborhood of ± 10 points. In order to do that, you may use the function `bandpass2d`, and you will create a frequency rejector by `1-bandpass2d`. Look again at the Fourier transform of Barbara, but wit the axes in points, so as to understand what you do. Apply the filter to the initial image and look at the result in the spatial domain. Try to

understand the modifications. In particular, **look at the tablecloth**. Also look at the difference image.

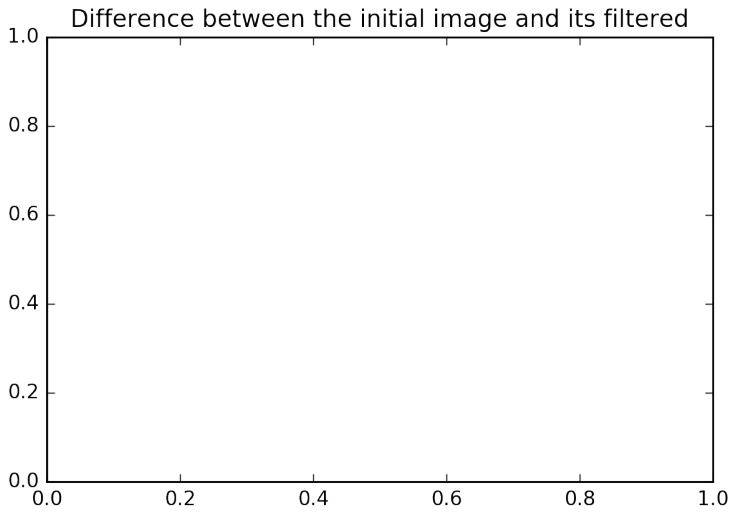
```
matplotlib.rcParams['savefig.dpi'] = savedpi  # restore initial dpi
# Rejector filter
#
#
#
plt.title('Bandpass filter (frequencies)')
```




```
matplotlib.rcParams['savefig.dpi'] = savedpi
#
#
#
plt.title("FT of the filtered image")
plt.xlabel('y')
```



```
# in the spatial domain we than have
matplotlib.rcParams['savefig.dpi'] = 2*savedpi
#B_filtered
#
#
plt.title("Filtered image by the frequency rejector")
# Differences:
#
#
plt.title("Difference between the initial image and its filtered")
```




### 10.5.4 Filtering by convolution

The function that will be useful in this part is the function `convolve` from the module scipy.ndimage (get it by `ndi.convolve` if ndimage has been imported under the name ndi). We will alo use a `np.random.normal(size=(...))` for adding gaussian noise (with typically a scale 0.1); or `saltpepper` for a salt and pepper noise.

- begin by implementing a convolution in two dimensions, by studying the following code:

```
h=ones((2*ll+1,2*ll+1)) # h the impulse response
for m in range(ll,M-ll):
    for n in range(ll,N-ll):
        B_filtered[m,n]=sum(sum(h*B[m-ll:m+ll+1,n-ll:n+ll+1]))
```

Compute a low-pass filtering of Barbara (h constant over an half width of 3 to 10), and examine the result.

```
(N,M)=np.shape(B)
#
#
#
plt.imshow(B_filtered,cmap='gray',origin='upper')
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-15-4b7705c0e90f> in <module>()
----> 1 (N,M)=np.shape(B)
      2 #
      3 #
      4 #
      5 plt.imshow(B_filtered,cmap='gray',origin='upper')


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

As we see, it is very simple. Nothing but a sum of products. A better implementation would take care of edge effects.

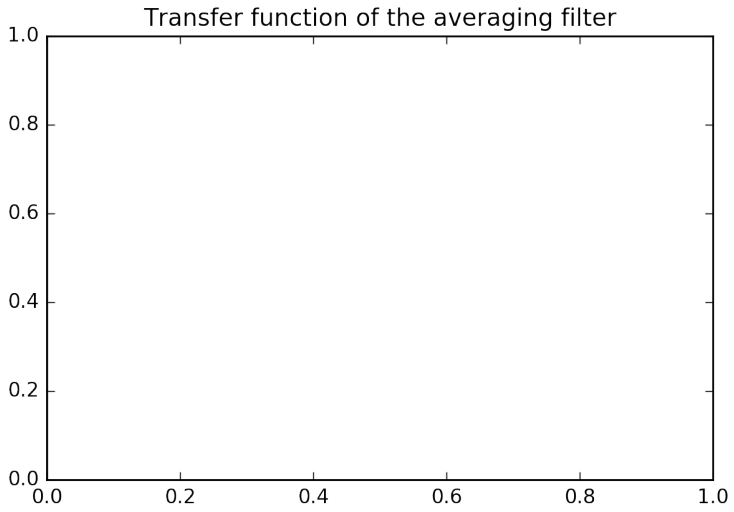
- Do this filtering again, but using the function `ndi.convolve`. Examine the effect of the filtering with an additive gaussian noise `np.random.normal(size=(512,512)` with scale 0.1 or a salt and pepper noise. Check that the filtering is indeed a low-pass filter by looking at its transfer function – use a zero-padding when computing the Fourier transform `fft2(h,s=(1000,1000))`. You can also use the `mesh` function for the representation.

```
B=plt.imread('barbara.png')
#
#
#
imshow(B_filtered,cmap='gray',origin='upper')
title('Filtered image')
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-16-f7e61fb77c67> in <module>()
      3 #
      4 #
----> 5 imshow(B_filtered,cmap='gray',origin='upper')
      6 title('Filtered image')


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

Transfer function

```
# Lets us look at the transfer function associated with this impulse
    response..
#
#
#
plt.title('Transfer function of the averaging filter')
```




- On the Barbara image, or on the image of cells, or of bacterias, test a Prewit or a Sobel gradient, with impulse responses

```
dx=np.array([[1.0, 0.0, -1.0],[1.0, 0.0, -1.0],[1.0, 0.0, -1.0],])
# dx=np.array([[1.0, 0.0, -1.0],[2.0, 0.0, -2.0],[1.0, 0.0, -1.0],]) #Sobel
dy=np.transpose(dx)
```

applied over the two directions (x,y), by `ndi.convolve` and build the gradient magnitude image.

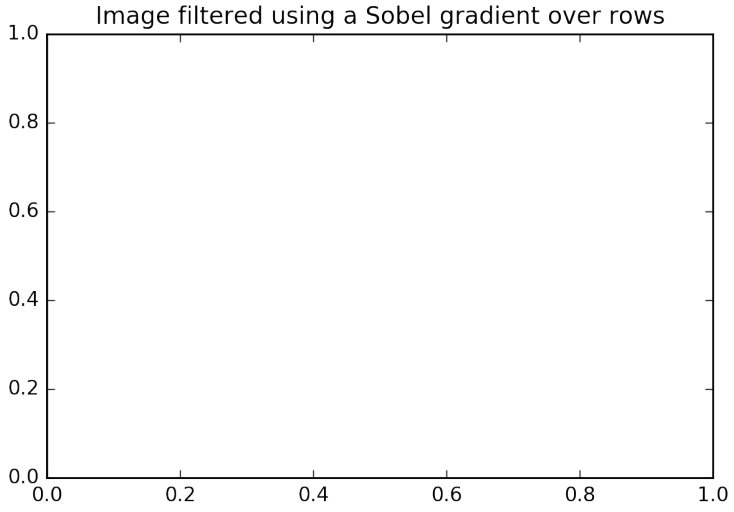
NB: if Dx and Dy are the gradient images obtained in directions x, y, then the gradient magnitude image is $\sqrt{Dx^2 + Dy^2}$.

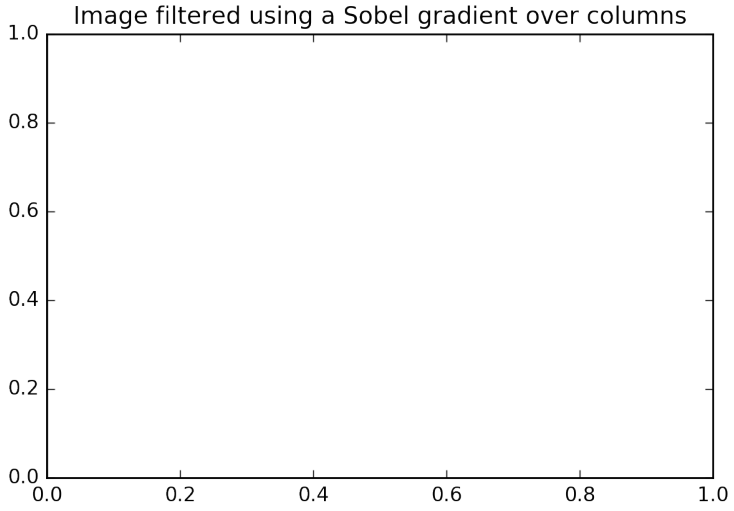
The ndimage module contains many predefined filters, such as `scipy.ndimage.filters.sobel`. However, we use here the direct convolution, for pedagogical purposes, instead of these functions.

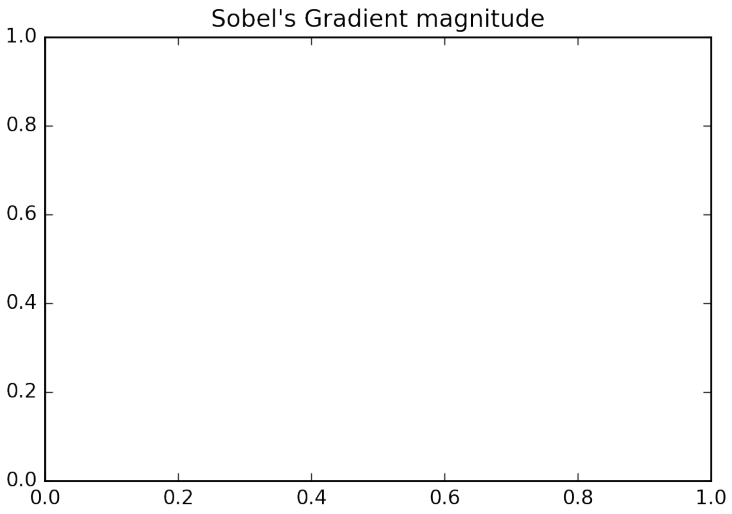
```
# Computation of gradient images and magnitude image
#
#
# Sobel Filter
#dx=np.array([[1.0, 0.0, -1.0],[2.0, 0.0, -2.0],[1.0, 0.0, -1.0],])
#dy=np.transpose(dx)
#
#
#
### Prewitt Filter
#dx=np.array([[1.0, 0.0, -1.0],[1.0, 0.0, -1.0],[1.0, 0.0, -1.0],])
#dy=np.transpose(dx)
```

```
plt.figure()
#
```

```
plt.title('Image filtered using a Sobel gradient over rows')
plt.figure()
#
plt.title('Image filtered using a Sobel gradient over columns')
plt.figure()
#
plt.title("Sobel's Gradient magnitude")
```



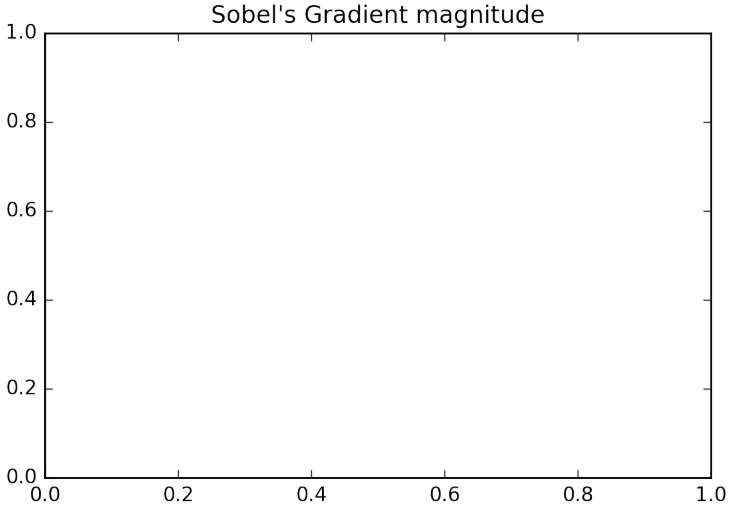



Sobel's transfer function

```
# Sobel's transfer function
#
#
```

idem with Barbara

```
B=plt.imread('barbara.png') # ou cellules.png
#
#
#
#
#
plt.title("Sobel's Gradient magnitude")
```



# 11

# Lab on basics in image processing

## 11.1 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.

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.

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 below) or using external windows; for that call `%matplotlib`. To return to the inline mode, use `%matplotlib inline`.

```
#This was for Python v. 2.7
#from __future__ import division
#from __future__ import print_function

import matplotlib.pyplot as plt
import numpy as np
from numpy import abs, where, sin, cos, ones, zeros, log, pi
from scipy import ndimage as ndi
from scipy import signal as sig
from numpy.fft import fft, ifft, fft2, ifft2, fftshift
from scipy.ndimage import convolve as convolvend
# For 3D representations
```

```
from mpl_toolkits.mplot3d.axes3d import Axes3D

# change the dpi for a better vizualisation of images
import matplotlib
savedpi=matplotlib.rcParams['savefig.dpi']
%matplotlib inline
```

Listing of offered utility functions

```
# from defs_tp_img import *
#
## Definitions
# ===========

def isodd(num):
        return num & 1 and True or False
# or num%2==1

##
def rect2(L,N):
    """ Returns a (2L+1)x(2L+1) rectangle centered over a
    NxN matrix"""
    x=np.zeros((N,N))
    center=(round((N+1)/2),round((N+1)/2)) if isodd(N) else (round(N/2+1),
        round(N/2+1))
    x[center[0]-L:center[0]+L+1,center[1]-L:center[1]+L+1]=1
    return x

##
def bandpass2d(position,half_width,total_dim):
    """ Returns the frequency response of a bandpass filter, correctly
        centered on
    `position`, of half-width `half_width` and symmeterized in frequency
        domain
    (zero at the center of the resulting matrix)
    Parameters
    ----------
    position: array
        center-frequency of the band-pass response
        (in samples -- the corresponding frequencies are 2*position/
            total_dim)
    half_width: integer
         half frequency band
    total_dim: integer
          image dimension
    Author : jfb"""
    N=total_dim
    center=(round((N+1)/2),round((N+1)/2)) if isodd(N) else (round(N/2+1),
        round(N/2+1))
    #
    H=zeros((N,N))
    H[center[0]+position[0]-half_width:center[0]+position[0]+half_width+1,
        center[1]+position[1]-half_width:center[1]+position[1]+half_width
        +1]=1
    position=-np.array(position)  # symmetric frequencies
    H[center[0]+position[0]-half_width:center[0]+position[0]+half_width+1,
        center[1]+position[1]-half_width:center[1]+position[1]+half_width
        +1]=1
    return H
```

```
## 2D fequency representation
def showfft2(X,Zero='uncentered', Freq='normalized', num=None,cmap='hot'):
    """Display an image, in the 2D Fourier domain
    The representation is centered and in normalized frequencies.
    **/!\\   *showfft2* **does not compute the Fourier transform**.
    The user is responsible for supplying correct data.
    Parameters:
    ----------
        X : 2D data in the Fourier domain
        Num: Figure number(optional)
        Zero='uncentered' (default), the input Fourier transform are
            supposed uncenterd and a
        fftshift is applied
        Freq='normalized' (default); otherwise axis are in samples
    ``Author: jfb``
    """
    N,M=np.shape(X)
    plt.figure(num)
    if Zero!='centered':
        X=fftshift(X)
    if Freq=='normalized':
        extent=(-0.5, 0.5, -0.5, 0.5)
    else:
        extent=(-N/2, N/2, -N/2, N/2)
    im=plt.imshow(X,aspect='auto',origin='lower', extent=extent, cmap=cmap
        )
    if Freq=='normalized':
        ticks=np.arange(-0.5,0.6,0.1)
        plt.xticks(ticks)
        plt.yticks(ticks)
    plt.colorbar()



def mesh(obj3D,numfig=None,subplt=(1,1,1),cmap='hot'):
    """
    Emulates Matlab's mesh
    Author: jfb
    """
    fig = plt.figure(numfig)
    ax = fig.add_subplot(subplt[0],subplt[1],subplt[2], projection='3d')
    m,n=np.shape(obj3D)
    mm=np.arange(m)
    nn=np.arange(n)
    X,Y=np.meshgrid(nn,mm)
    ax.plot_surface(X, Y, obj3D, rstride=max([round(m/50), 1]),
                    cstride=max([round(n/50), 1]), cmap=cmap)

def saltpepper(percent=85,shape=None):
        s=np.random.random(size=shape)
        d=np.where(s>percent/100)
        out=np.ones(shape=shape)
        out[d]=0
        return out
```

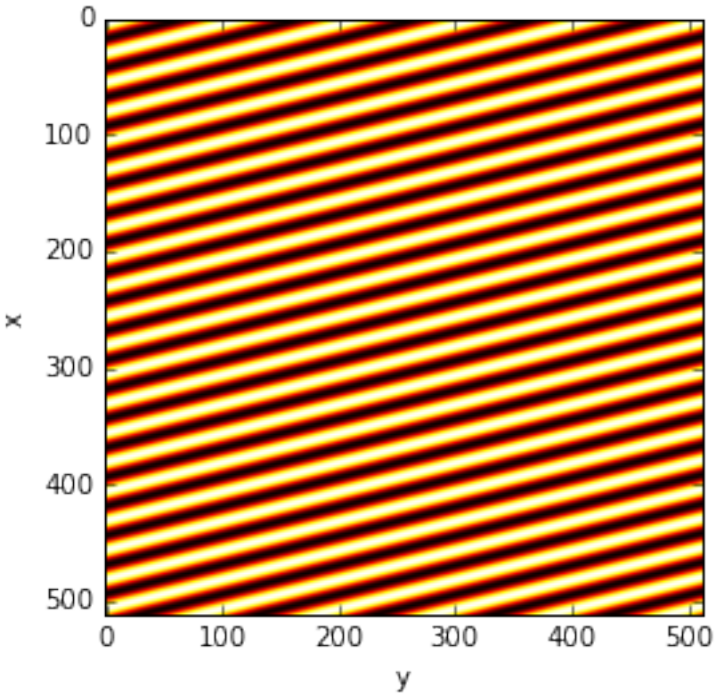
## 11.2 Frequency representation – Filtering in the frequency domain

### 11.2.1 A pretty sine wave

- Generate a sine wave $f(x,y) = \sin(2\pi(f_x x + f_y y))$ , with $f_x$ an $f_y$ between 0.02 à 0.2, on $N \times N$ points (e.g. $N = 512$). For the implementation, you may use a `for` loop, or a double comprehension list, or even the smart function `fromfunction()` and an anonymous `lambda` function. Display the result using `imshow`.

Note that for a matrix, usually the first index denotes the rows and the second the columns. Therefore modifying something with respect to x results in a variation along the rows of the matrix (usually the "y" axis)..

```
fx,fy= 0.04, 0.01

x=np.arange(512)
y=np.arange(512)
# using a double comprehension list
S=[[sin(2*pi*(fx*xx+fy*yy)) for xx in x] for yy in y]
# of the fromfunction
Z=np.fromfunction(lambda x,y: sin(2*pi*(fx*x+fy*y)),(512,512))
# Display
plt.imshow(Z,cmap='hot',origin='upper')
_=plt.xlabel('y')
_=plt.ylabel('x')
```



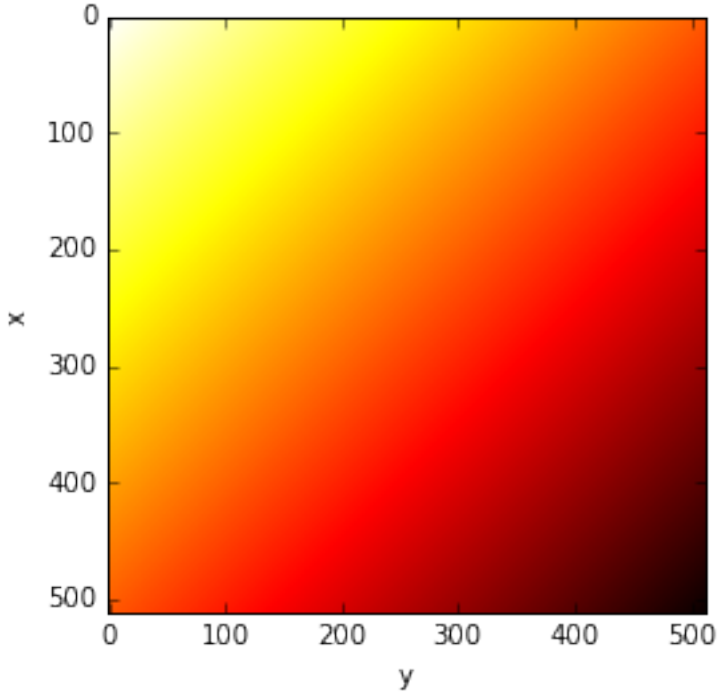

- Transform your program into a function `pltsin` which takes the two frequencies as inputs. Then implement a interactive version, using the lines

```
from IPython.html.widgets import interact, interactive, fixed
interact(pltsin,fx=[-0.1,0.1,0.01], fy=[-0.1,0.1,0.01])
```

Then play with the parameters, and look at what happens when you change the values of the frequencies.

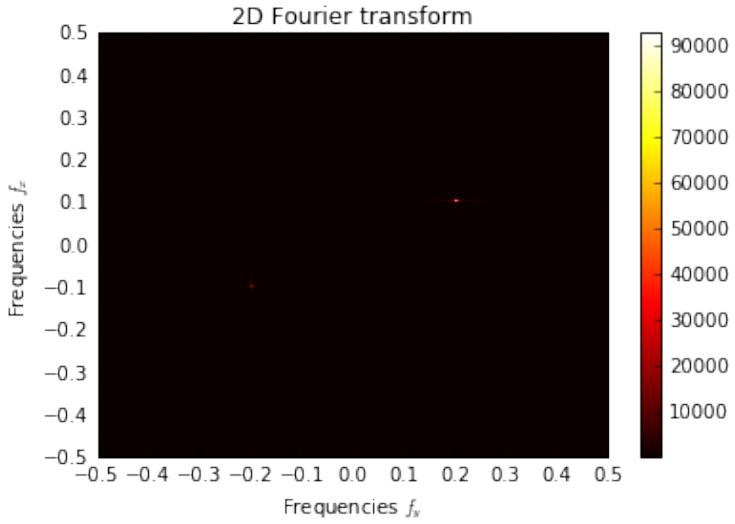
```
#from IPython.html.widgets import interact, interactive, fixed
from ipywidgets import interact, interactive, fixed
def pltsin(fx,fy):
    Z=np.fromfunction(lambda x,y: sin(2*pi*(fx*x+fy*y)),(512,512))
    plt.imshow(Z,cmap='hot',origin='upper')
    _=plt.xlabel('y'), plt.ylabel('x')
interact(pltsin,fx=[-0.1,0.1,0.01], fy=[-0.1,0.1,0.01])
```

```
<function __main__.pltsin>
```



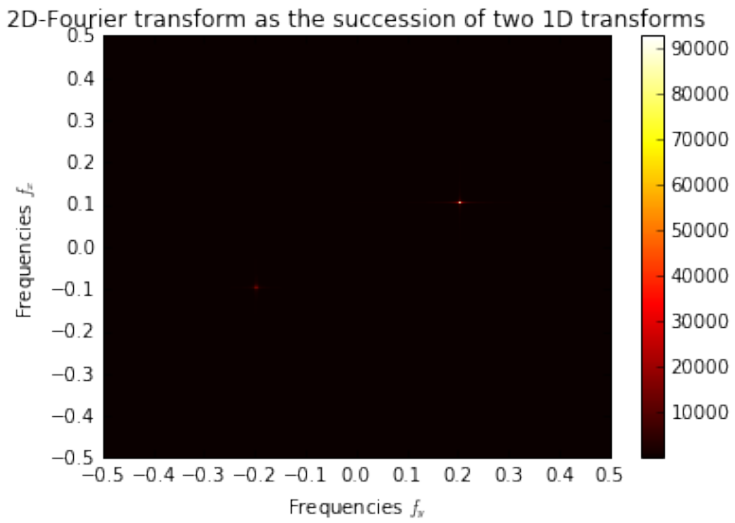

- Compute the 2D Fourier transform of $f(x,y)$ (using function fft2) and display the modulus (function abs) of the result, via `showfft2`. What are the significations of the axes? What are the spatial frequencies of the sine wave. Experiment with several values of $f_x, f_y$.

```
# We change the frequenies for ease of visualization in the frequency
    domain
fx,fy= 0.1, 0.2
S=np.fromfunction(lambda x,y: sin(2*pi*(fx*x+fy*y)),(512,512))

# Display
showfft2(abs(fft2(S)),num=4)
plt.title('2D Fourier transform')
plt.xlabel('Frequencies $f_y$')
_=plt.ylabel('Frequencies $f_x$')
```




- Show theoretically, and then check it numerically, that the 2D Fourier transform can be obtained as the succession of two 1-dimensional transforms, applied respectively to the rows and then to the columns (or vice versa). You will take advantage of the fact that the standard `fft` has a parameter axis which is the axis over which the `fft` is actually computed.

```
# Correction
showfft2(abs(fft(fft(S,axis=0),axis=1)),num=5)
plt.title('2D-Fourier transform as the succession of two 1D transforms')
plt.xlabel('Frequencies $f_y$')
_=plt.ylabel('Frequencies $f_x$')
```

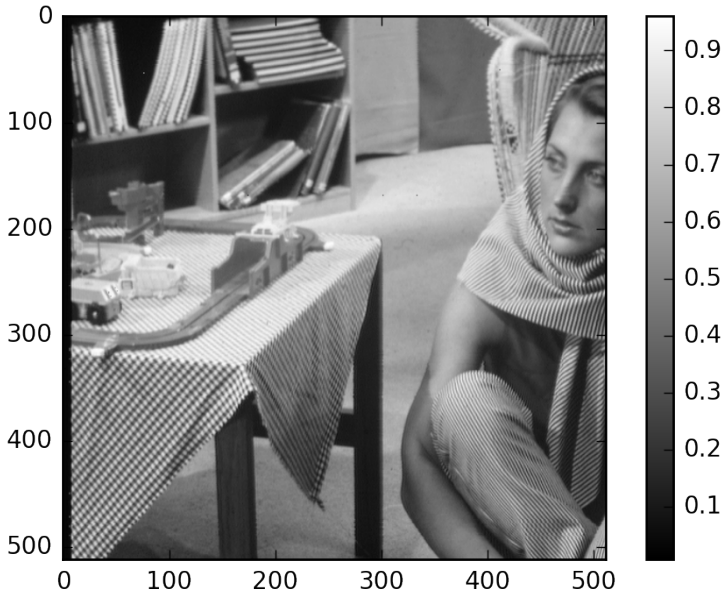



## 11.3 Playing with Barbara – Filtering in the frequency domain

*Rappelle-toi Barbara Il pleuvait sans cesse sur Brest ce jour-là Et tu marchais souriante Épanouie ravie ruisselante Sous la pluie* (Prévert, 1946)

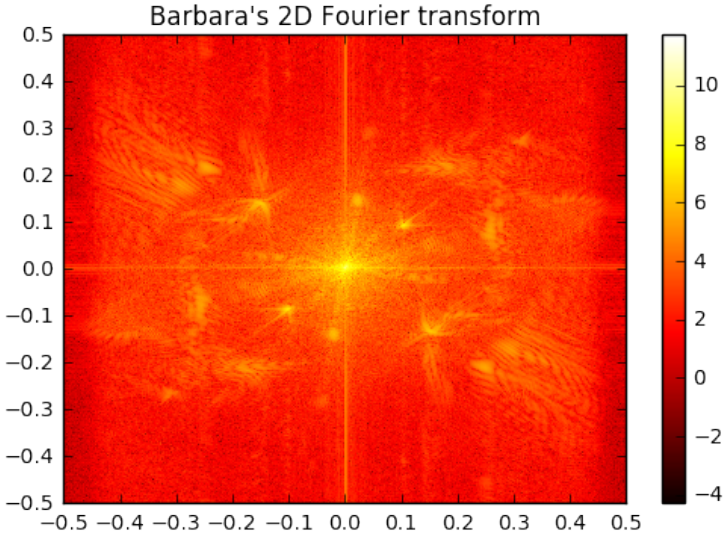
- Load the image Barbara, using `plt.imread(barbara.png)` and display it (in gray levels).

```
matplotlib.rcParams['savefig.dpi'] = 2*savedpi # double dpi

#%% reading and displaying the image
B=plt.imread('barbara.png')
plt.figure(1)
plt.imshow(B,cmap='gray',origin='upper')
_=plt.colorbar()
```




- Display the corresponding frequency representation `showfft2` , in log scale (simply take `log(abs())`!)

```
matplotlib.rcParams['savefig.dpi'] = savedpi #restore dpi
showfft2(log(abs(fft2(B))))
_=plt.title("Barbara's 2D Fourier transform")
```

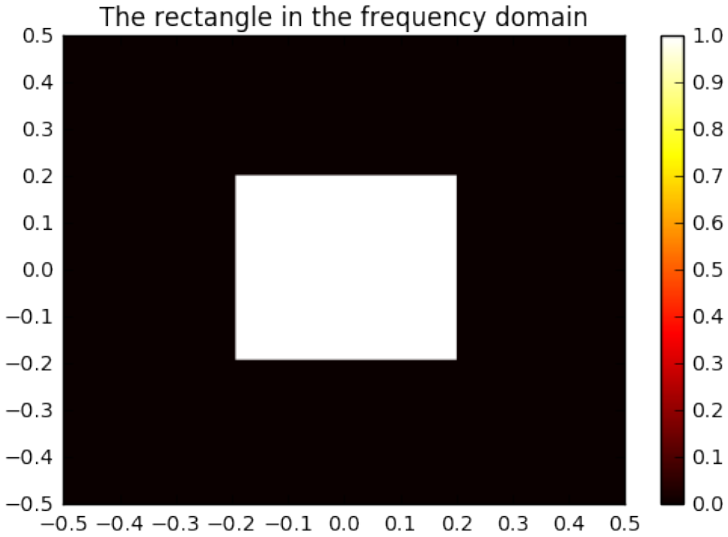


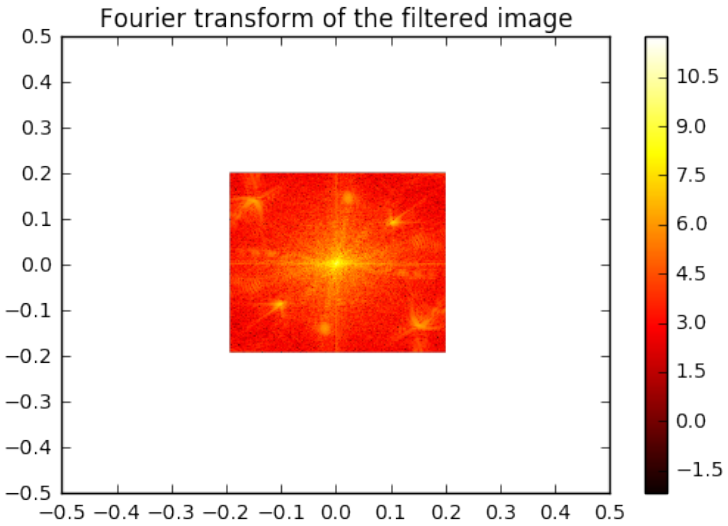
- Filter this image using a low-pass filter with a rectangular frequency response (use the function `rect2`), for rectangles of half-width 40, 80 ,100. Display the different resulting images, as well as the differences with the initial image. Observations, conclusions.

```
L=100
H=rect2(L,512)
```

```
L=100
H=rect2(L,512)
showfft2(H,Zero='centered') ## !
plt.title('The rectangle in the frequency domain')
# Filtering
Bf_filtered=fft2(B)*fftshift(H)
showfft2(log(abs(Bf_filtered)))
plt.title('Fourier transform of the filtered image')
```

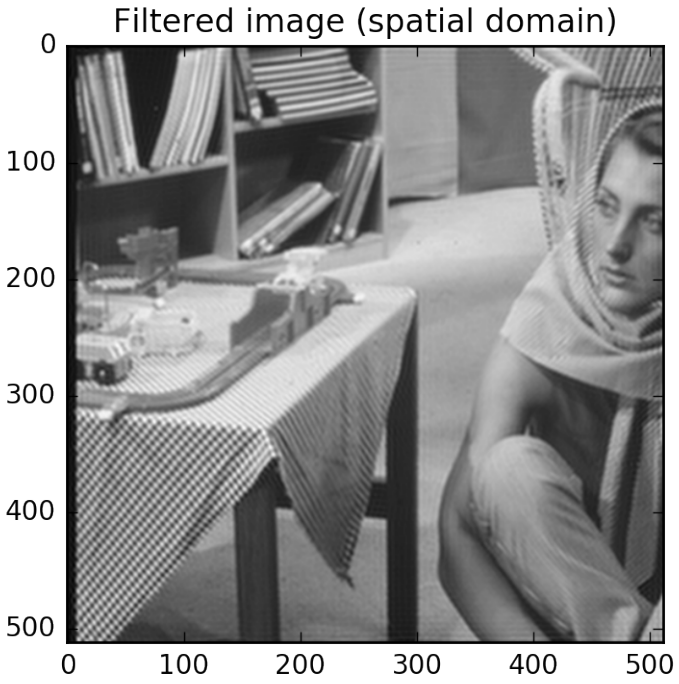
```
/usr/local/lib/python3.5/site-packages/ipykernel/__main__.py:7: RuntimeWarning: divide by zero e
```

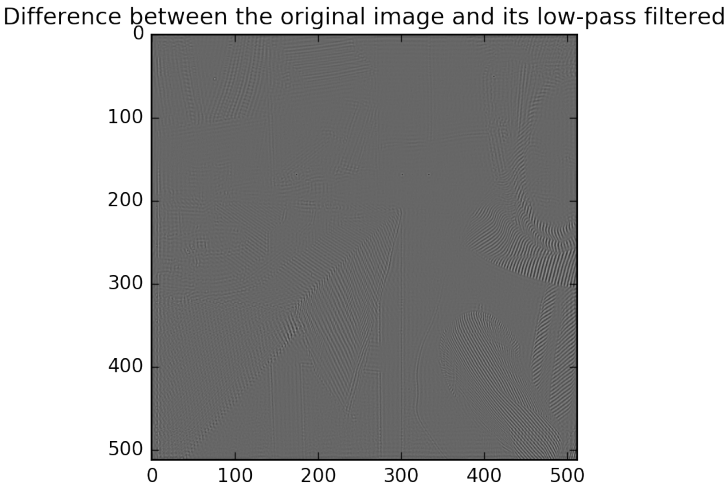


The corresponding results in the spatial domain are:

```
matplotlib.rcParams['savefig.dpi'] = 2*savedpi
#And in the spatial domain
plt.imshow(np.real(ifft2(Bf_filtered)),cmap='gray',origin='upper')
plt.title("Filtered image (spatial domain)")
# Also experiment with L=80, L=100
# Differences :
plt.figure()
plt.imshow(B-np.real(ifft2(Bf_filtered)),cmap='gray',origin='upper')
_=plt.title("Difference between the original image and its low-pass
    filtered")
```



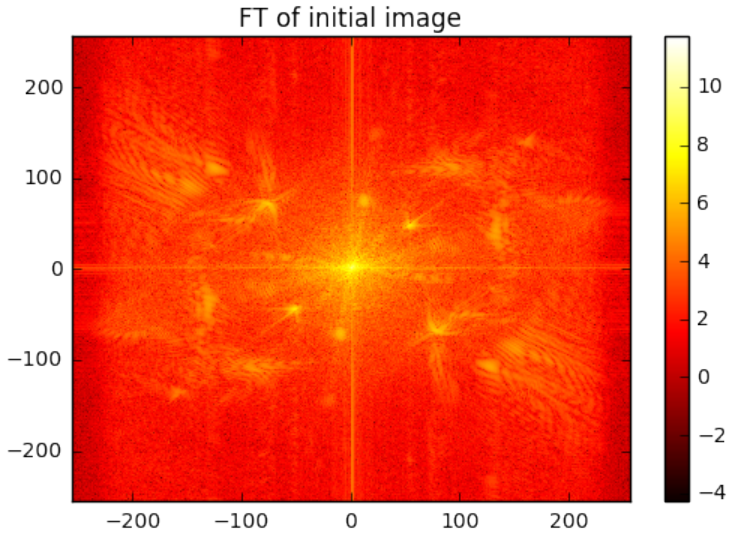



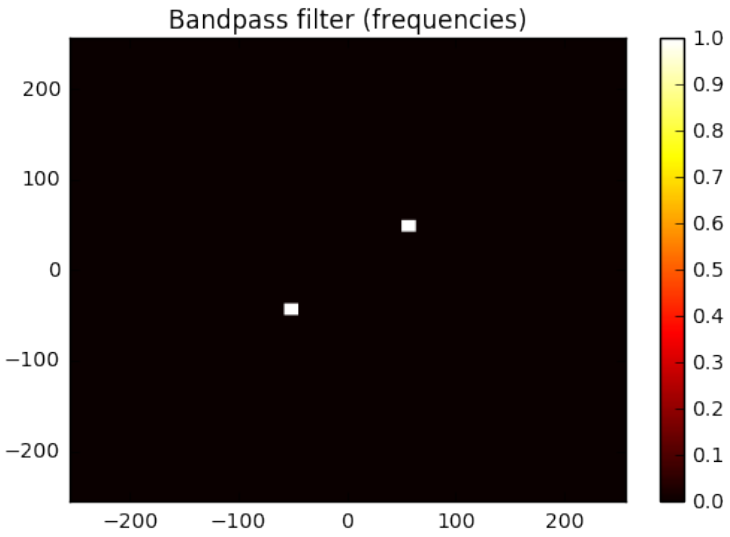

What appear in the difference image are indeed the “high frequencies”, which are the “details”

of the original image (Actually, by computing the difference with the low-pass filtered, we have implemented an high-pass filter...)

- design a frequency response that kills selectively the frequencies around points (46,54) and (-70,79), e.g.~on a neighborhood of ± 10 points. In order to do that, you may use the function `bandpass2d`, and you will create a frequency rejector by `1-bandpass2d`. Look again at the Fourier transform of Barbara, but wit the axes in points, so as to understand what you do. Apply the filter to the initial image and look at the result in the spatial domain. Try to understand the modifications. In particular, **look at the tablecloth**. Also look at the difference image.

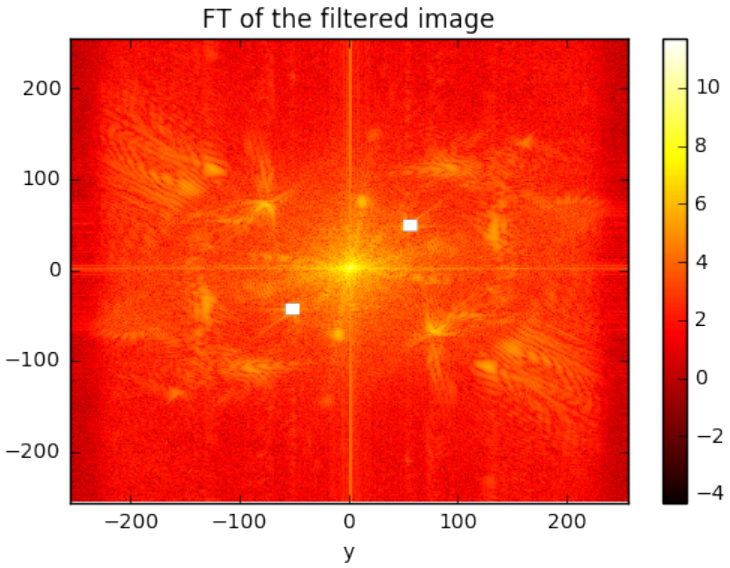
```
matplotlib.rcParams['savefig.dpi'] = savedpi  # restore initial dpi
# Rejector filter
B=plt.imread('barbara.png');
Bf=fft2(B)
showfft2(log(abs(Bf)),Zero='uncentered',Freq='unnorm') ## !
plt.title("FT of initial image")
## figure(3)
X=bandpass2d((46,54),6,512)
# beware ; when adressing an array A(x,y), the "x" are the rows
# (1st index) eand the  "y" the columns, therefore we have to exchange the
    two indexes.

showfft2(X,Zero='centered',Freq='unnorm')
plt.title('Bandpass filter (frequencies)')
```





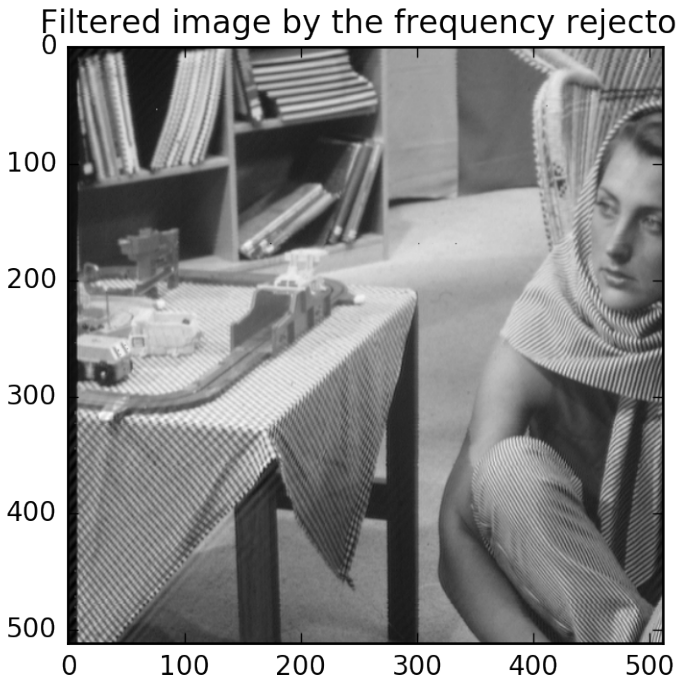

```
matplotlib.rcParams['savefig.dpi'] = savedpi
X=bandpass2d((46,54),6,512)
## figure(4)
X=1-X #rejector
Bf_filtered=X*fftshift(Bf)
showfft2(log(abs(Bf_filtered)),Zero='centered',Freq='unnorm') ## !
plt.title("FT of the filtered image")
plt.xlabel('y')
```

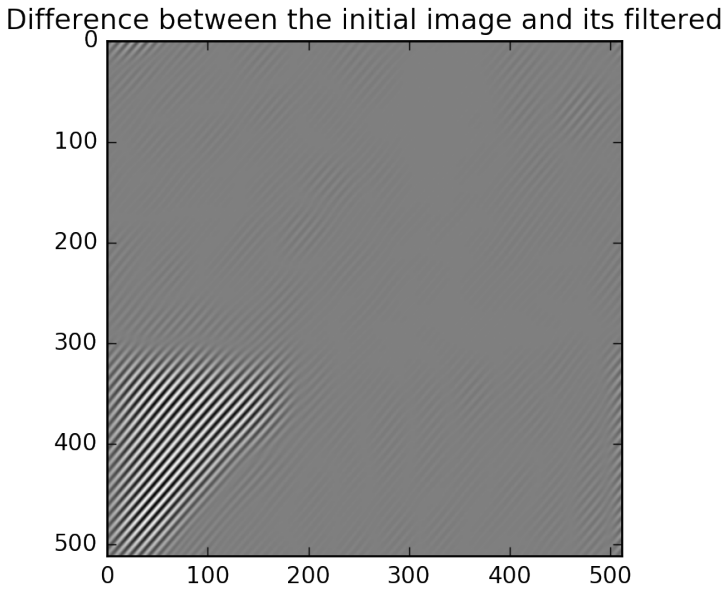
```
/usr/local/lib/python3.5/site-packages/ipykernel/__main__.py:6: RuntimeWarning: divide by zero e
```




```
# in the spatial domain we than have
matplotlib.rcParams['savefig.dpi'] = 2*savedpi
B_filtered=np.real(ifft2(fftshift(Bf_filtered)))
plt.figure()
plt.imshow(B_filtered,cmap='gray',origin='upper')
```

```
plt.title("Filtered image by the frequency rejector")
# Differences:
plt.figure()
plt.imshow(B-B_filtered,cmap='gray',origin='upper')
plt.title("Difference between the initial image and its filtered")
```




*Look at the tablecloth!*

## 11.4 Filtering by convolution

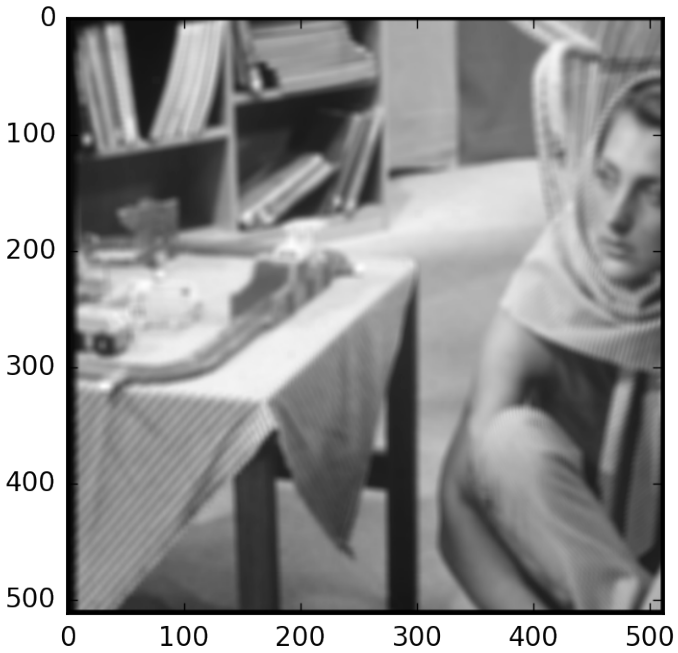
The function that will be useful in this part is the function `convolve` from the module scipy.ndimage (get it by `ndi.convolve` if ndimage has been imported under the name ndi). We will alo use a `np.random.normal(size=(...))` for adding gaussian noise (with typically a scale 0.1); or `saltpepper` for a salt and pepper noise.

- begin by implementing a convolution in two dimensions, by studying the following code:

```
h=ones((2*ll+1,2*ll+1)) # h the impulse response
for m in range(ll,M-ll):
    for n in range(ll,N-ll):
        B_filtered[m,n]=sum(sum(h*B[m-ll:m+ll+1,n-ll:n+ll+1]))
```

Compute a low-pass filtering of Barbara (h constant over an half width of 3 to 10), and examine the result.

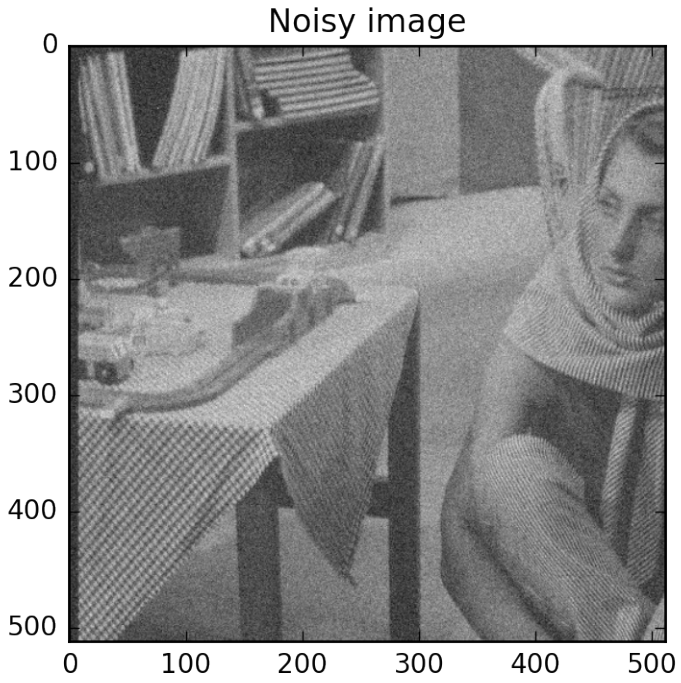
```
(N,M)=np.shape(B)
ll=3
h=ones((2*ll+1,2*ll+1)) # the impulse response
for m in range(ll,M-ll):
    for n in range(ll,N-ll):
        B_filtered[m,n]=sum(sum(h*B[m-ll:m+ll+1,n-ll:n+ll+1]))

plt.figure(9)
plt.imshow(B_filtered,cmap='gray',origin='upper')
```

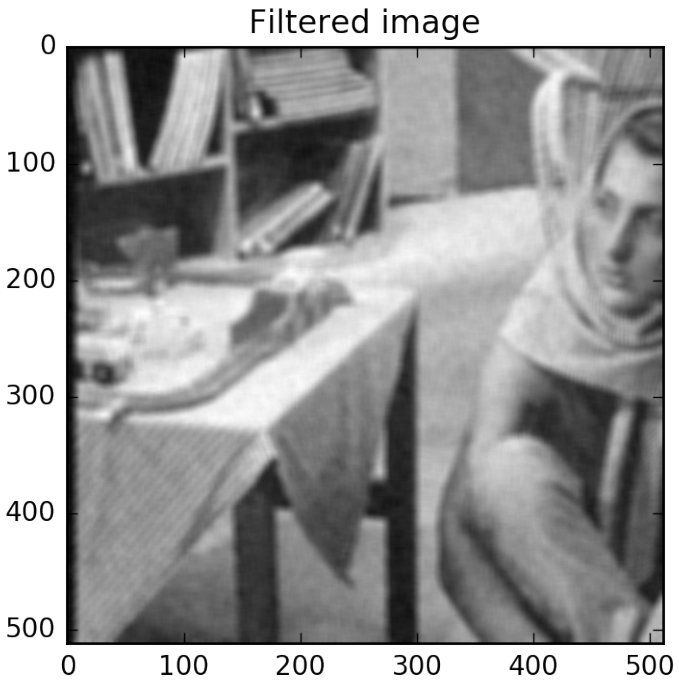



As we see, it is very simple. Nothing but a sum of products. A better implementation would take care of edge effects.

- Do this filtering again, but using the function `ndi.convolve`. Examine the effect of the filtering with an additive gaussian noise `np.random.normal(size=(512,512)` with scale 0.1 or a salt and pepper noise. Check that the filtering is indeed a low-pass filter by looking at its transfer function – use a zero-padding when computing the Fourier transform `fft2(h,s=(1000,1000))`. You can also use the `mesh` function for the representation.

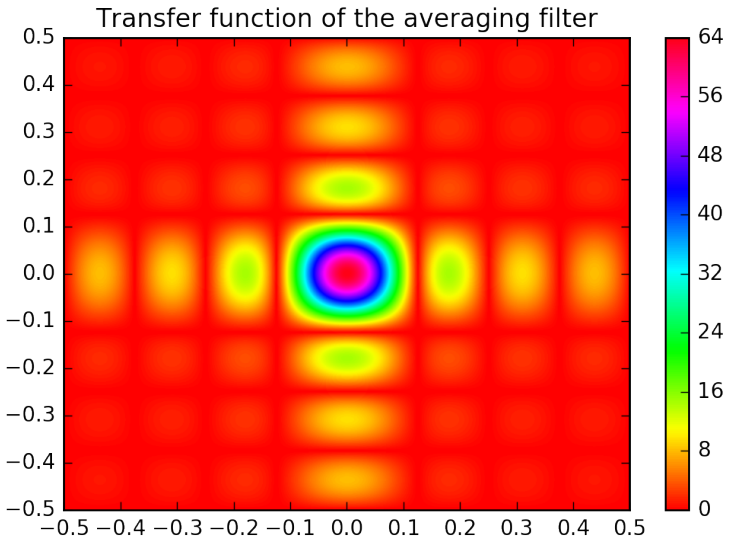
```
B=plt.imread('barbara.png')
L=8
h=ones((L,L)) # the IR
B=B+0.1*np.random.normal(size=(512,512))
#B=B+0.5*saltpepper(percent=98, shape=np.shape(B))
plt.figure(9)
plt.imshow(B,cmap='gray',origin='upper')
plt.title('Noisy image')
B_filtered=ndi.convolve(B,h)
plt.figure(10)
plt.imshow(B_filtered,cmap='gray',origin='upper')
plt.title('Filtered image')
```

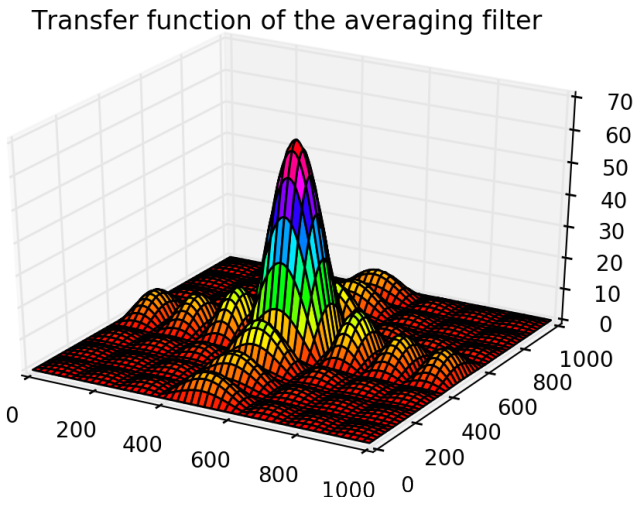





Transfer function

```
# Lets us look at the transfer function associated with this impulse
    response..
H=fft2(h,s=(1000,1000))
showfft2((abs(H)),Zero='uncentered',Freq='normalized', num=11, cmap='hsv')
     ## !
plt.title('Transfer function of the averaging filter')
#
mesh(abs(fftshift(H)),numfig=12, cmap='hsv')
plt.title('Transfer function of the averaging filter')
```






Conclusion: it is indeed a low-pass!

- On the Barbara image, or on the image of cells, or of bacterias, test a Prewit or a Sobel gradient, with impulse responses

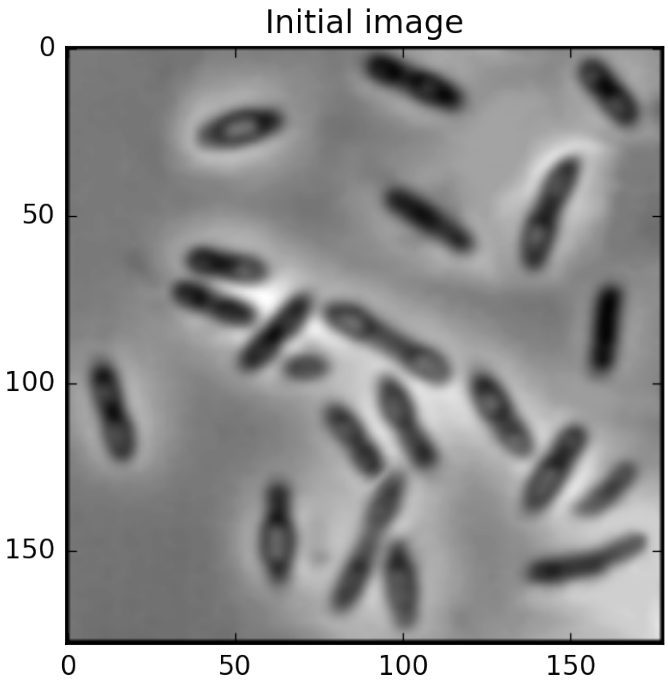
```
dx=np.array([[1.0, 0.0, -1.0],[1.0, 0.0, -1.0],[1.0, 0.0, -1.0],])
# dx=np.array([[1.0, 0.0, -1.0],[2.0, 0.0, -2.0],[1.0, 0.0, -1.0],]) #Sobel
dy=np.transpose(dx)
```

applied over the two directions (x,y), by `ndi.convolve` and build the gradient magnitude image.

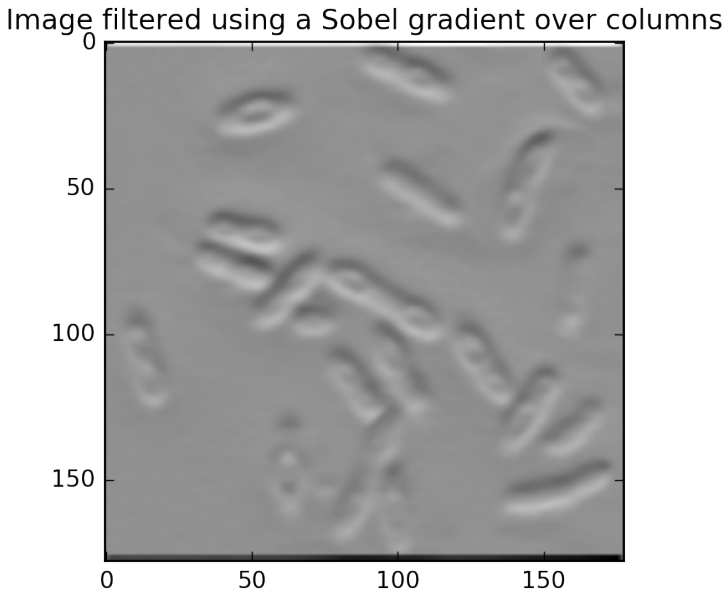
NB: if Dx and Dy are the gradient images obtained in directions x, y, then the gradient magnitude image is $\sqrt{Dx^2 + Dy^2}$.

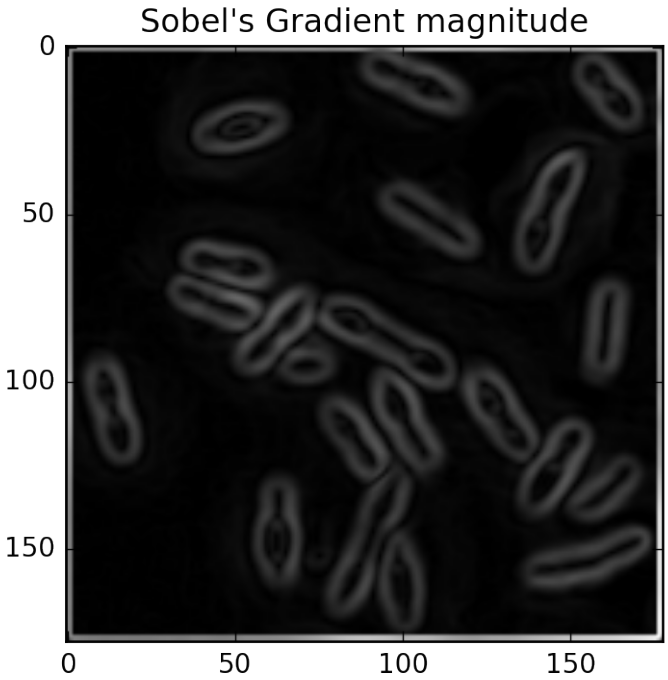
The ndimage module contains many predefined filters, such as `scipy.ndimage.filters.sobel`. However, we use here the direct convolution, for pedagogical purposes, instead of these functions.

```
# Computation of gradient images and magnitude image
B=plt.imread('bacteria.png') # ou cellules.png
plt.figure()
plt.imshow(B,cmap='gray',origin='upper')
plt.title('Initial image')
# Sobel Filter
dx=np.array([[1.0, 0.0, -1.0],[2.0, 0.0, -2.0],[1.0, 0.0, -1.0],])
dy=np.transpose(dx)
fo1=ndi.convolve(B,dx, output=np.float64, mode='nearest')
fo2=ndi.convolve(B,dy, output=np.float64, mode='nearest')
magnitude=np.sqrt(fo1**2+fo2**2)

### Prewitt Filter
#dx=np.array([[1.0, 0.0, -1.0],[1.0, 0.0, -1.0],[1.0, 0.0, -1.0],])
#dy=np.transpose(dx)
```



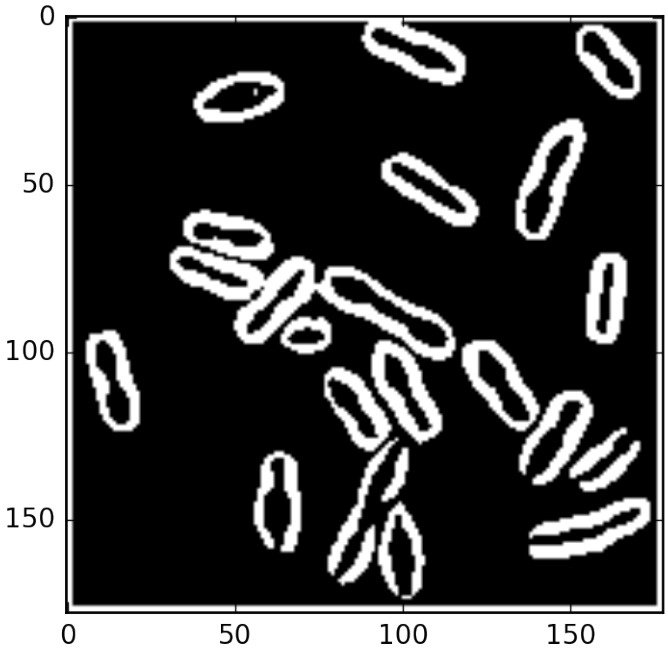

```
plt.figure()
plt.imshow(fo1,cmap='gray',origin='upper')
plt.title('Image filtered using a Sobel gradient over rows')
plt.figure()
plt.imshow(fo2,cmap='gray',origin='upper')
plt.title('Image filtered using a Sobel gradient over columns')
plt.figure()
plt.imshow(magnitude,cmap='gray',origin='upper')
plt.title("Sobel's Gradient magnitude")
```



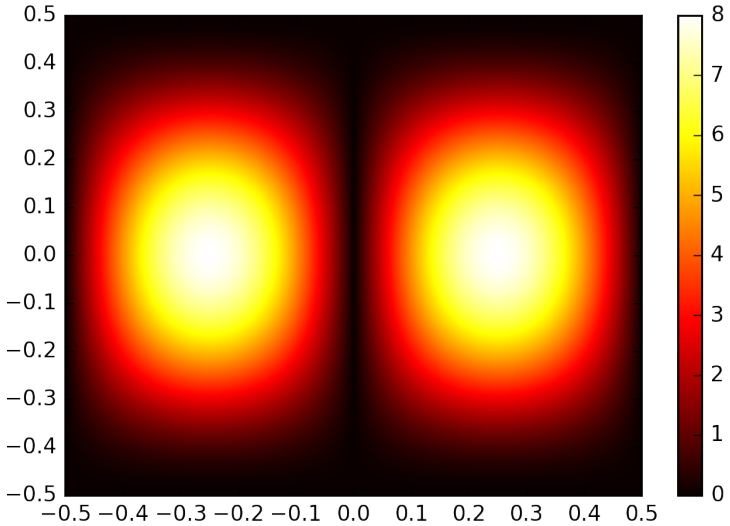

Let us transform this into a black and white image:

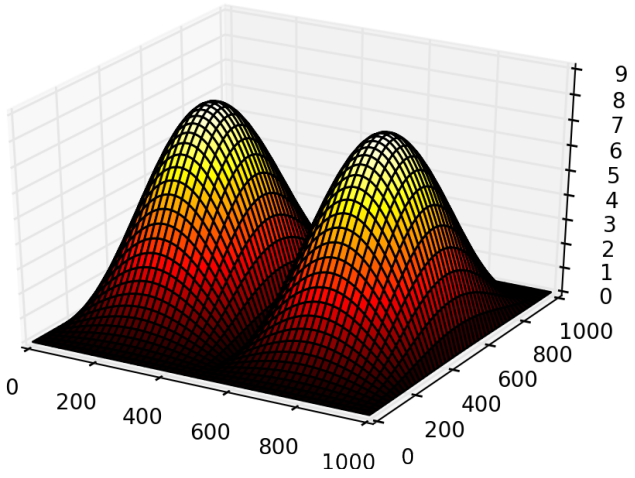
```
magnitude=magnitude/np.max(magnitude)
X=np.zeros(np.shape(magnitude))
X[magnitude>0.15]=100
plt.imshow(X, cmap='gray')
```




Sobel's transfer function

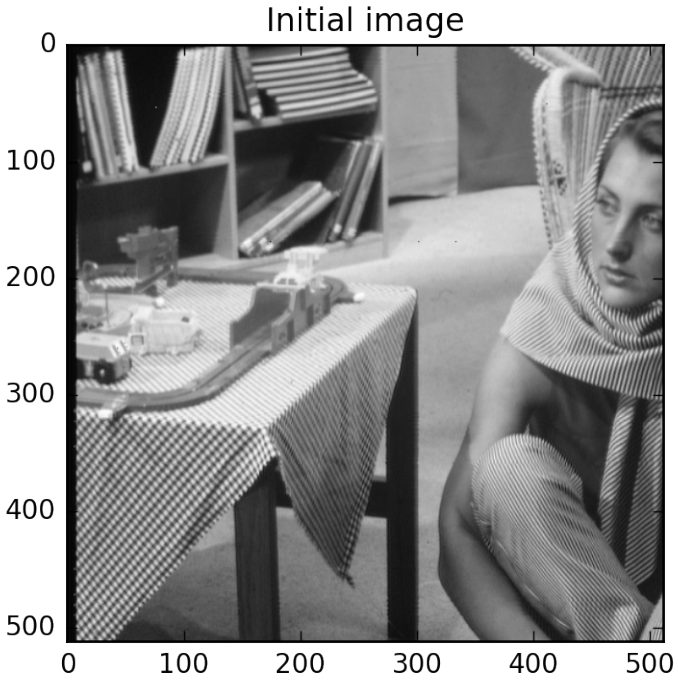
```
# Sobel's transfer function
Dx=fft2(dx,s=(1000,1000))
showfft2((abs(Dx)),Zero='uncentered',Freq='normalized') ## !
# or mesh(abs(fftshift(Dx)))
mesh(abs(fftshift(Dx)))
# It is indeed an high-pass filter!
```






Idem with Barbara

```
B=plt.imread('barbara.png') # ou cellules.png
plt.figure()
plt.imshow(B,cmap='gray',origin='upper')
plt.title('Initial image')
# Sobel Filter
dx=np.array([[1.0, 0.0, -1.0],[2.0, 0.0, -2.0],[1.0, 0.0, -1.0],])
dy=np.transpose(dx)
fo1=ndi.convolve(B,dx, output=np.float64, mode='nearest')
fo2=ndi.convolve(B,dy, output=np.float64, mode='nearest')
magnitude=np.sqrt(fo1**2+fo2**2)
#
plt.figure()
plt.imshow(fo1,cmap='gray',origin='upper')
plt.title('Image filtered using a Sobel gradient over rows')
plt.figure()
plt.imshow(fo2,cmap='gray',origin='upper')
plt.title('Image filtered using a Sobel gradient over cols')
plt.figure()
plt.imshow(magnitude,cmap='gray',origin='upper')
plt.title("Sobel's Gradient magnitude")
```


Initial image
0
100
200
300
400
500
0
100
200
300
400
500

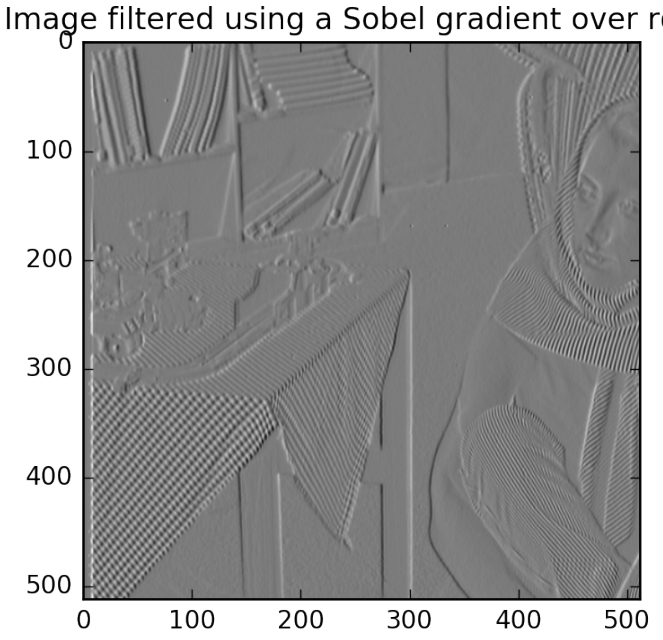


Image filtered using a Sobel gradient over rows
0
100
200
300
400
500
0
100
200
300
400
500

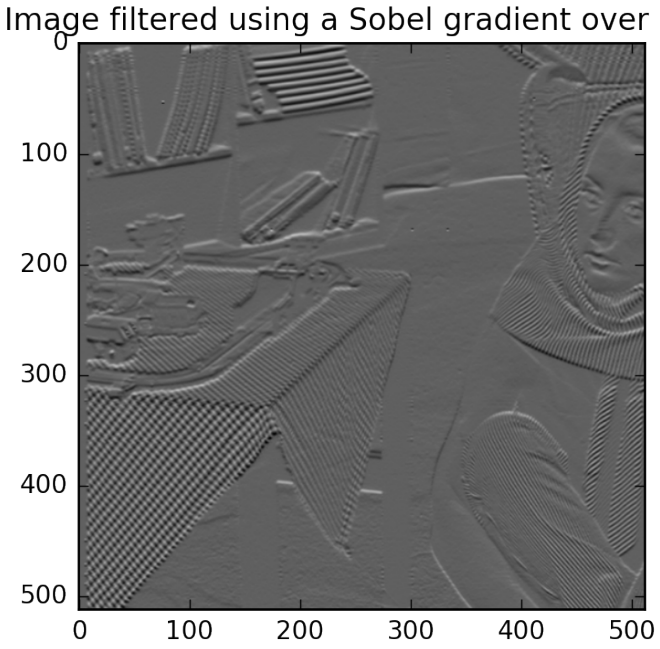
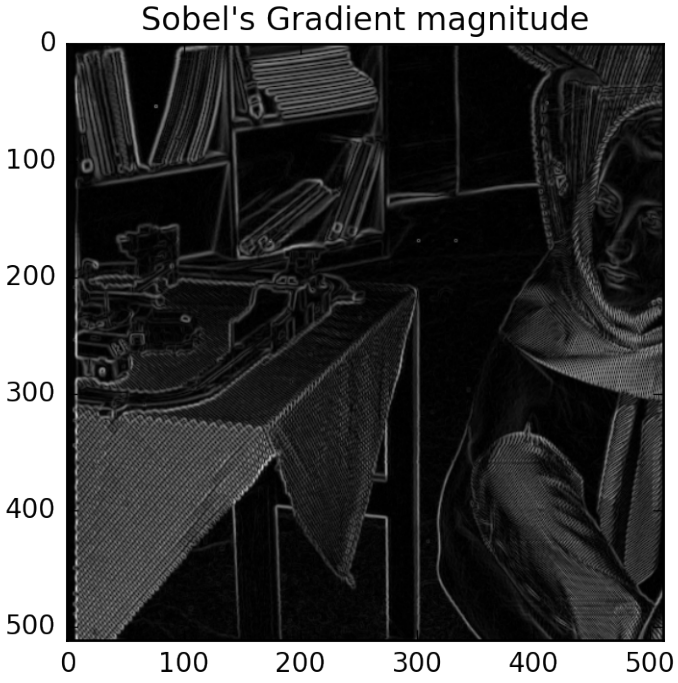


Image filtered using a Sobel gradient over cols
0
100
200
300
400
500
0
100
200
300
400
500


Sobel's Gradient magnitude
0
100
200
300
400
500
0
100
200
300
400
500

# 12 Digital filters

### 12.0.1 Introduction

To be continued

### 12.0.2 The z-transform

To be continued

## 12.1 Pole-zero locations and transfer functions behavior

```
from zerospolesdisplay import ZerosPolesDisplay
```

Let $H(z)$ be a rational fraction

$$H(z) = \frac{N(z)}{D(z)}, \tag{12.1}$$

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 1.** *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:

$$H(z = e^{j2\pi f}) = H(f). \tag{12.2}$$

**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=\rho e^{j\theta}\$ 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.

### 12.1.1 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

$$N(z) = (z - z_0)(z - z_0^*) = (z - \rho e^{j\theta})(z - \rho e^{-j\theta}) \tag{12.3}$$
$$= z^2 - 2\rho\cos(\theta)z + \rho^2 \tag{12.4}$$

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)
```

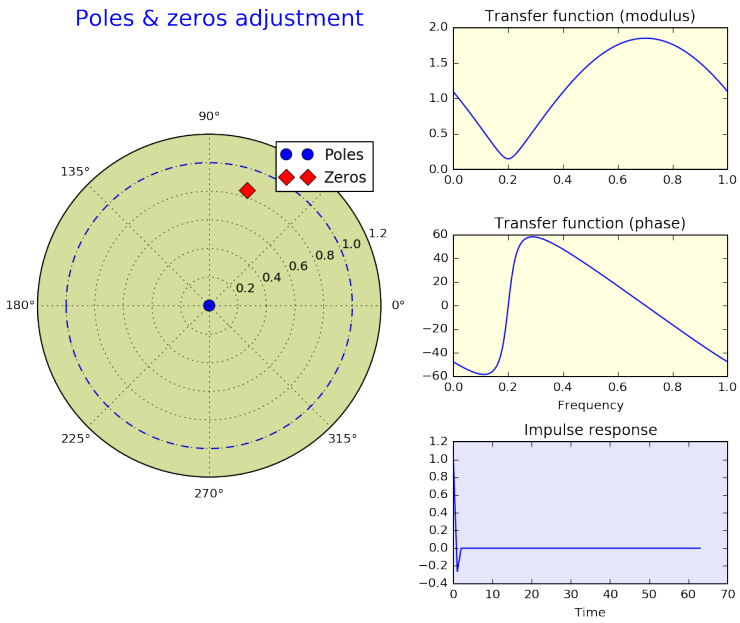



Figure 12.1: Poles-Zeros representation, Transfer function and Impulse response for a single zero

```
poles=np.array([0])
zeros=np.array([0.95*np.exp(1j*2*pi*0.4)])
A=ZerosPolesDisplay(poles,zeros)
```

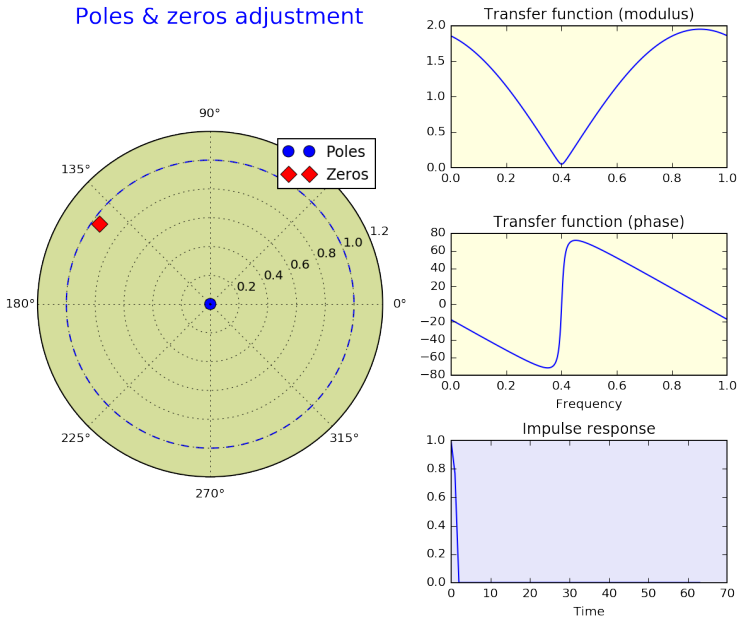



Figure 12.2: Poles-Zeros representation, Transfer function and Impulse response for a single zero

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).

```
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)
```

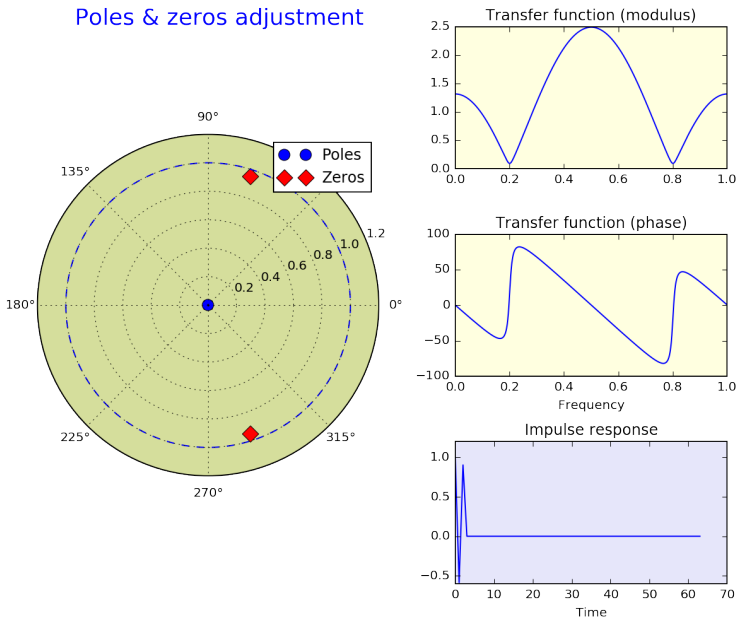



Figure 12.3: Poles-Zeros representation, Transfer function and Impulse response for a double zero

```
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), 0.97*
    np.exp(1j*2*pi*0.3), 0.97*np.exp(-1j*2*pi*0.3)])
A=ZerosPolesDisplay(poles,zeros)
```

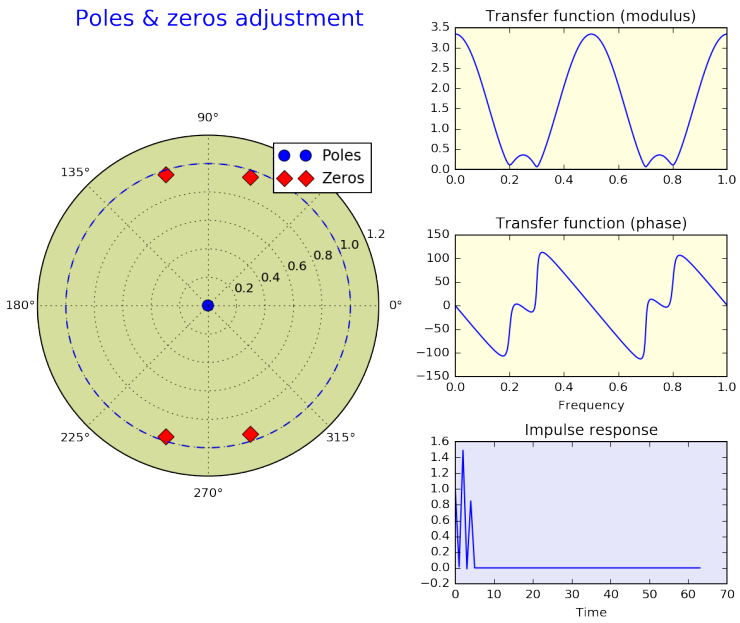



Figure 12.4: Poles-Zeros representation, Transfer function and Impulse response for a 4 zeros

### 12.1.2 Analysis of all-poles transfer functions

For an all-pole transfer function,

$$H(z) = \frac{1}{D(z)} \tag{12.5}$$

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)
```

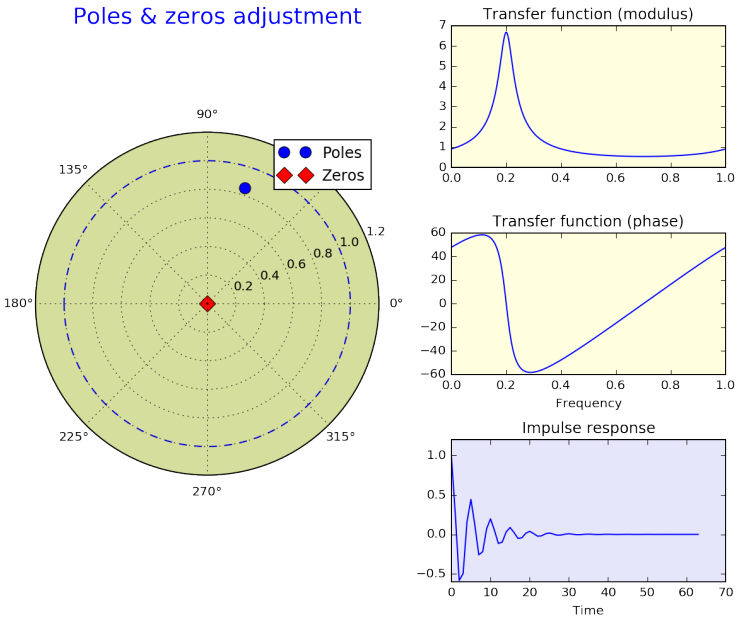



Figure 12.5: Poles-Zeros representation, Transfer function and Impulse response for a single pole

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)
```

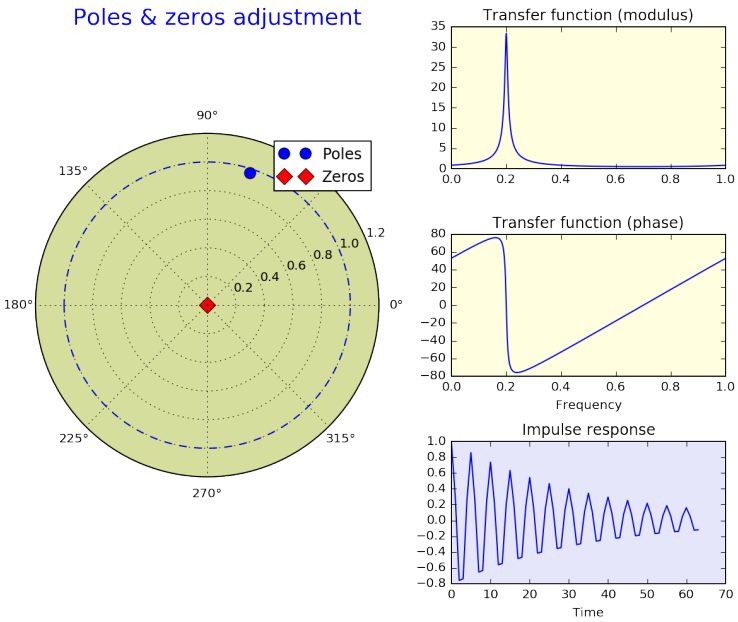



Figure 12.6: Poles-Zeros representation, Transfer function and Impulse response for a single pole

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.

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)
```

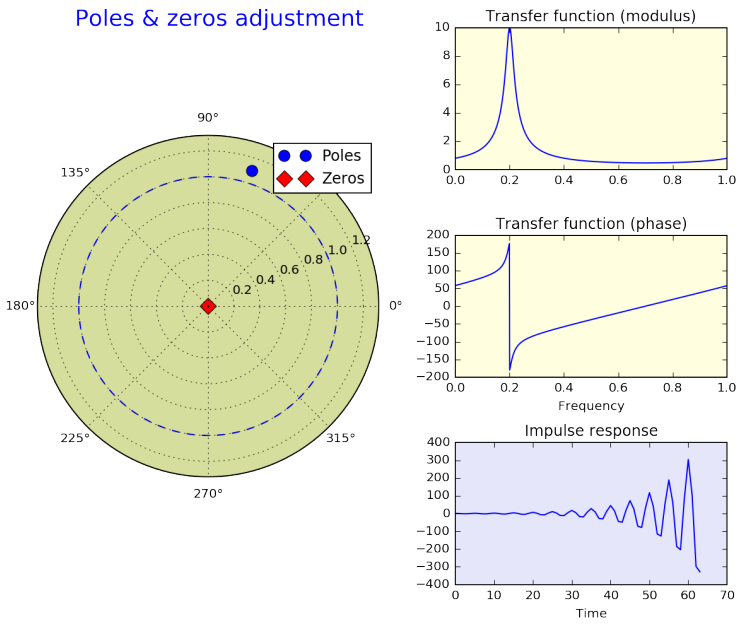



Figure 12.7: Poles-Zeros representation, Transfer function and Impulse response for a single pole

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)
```

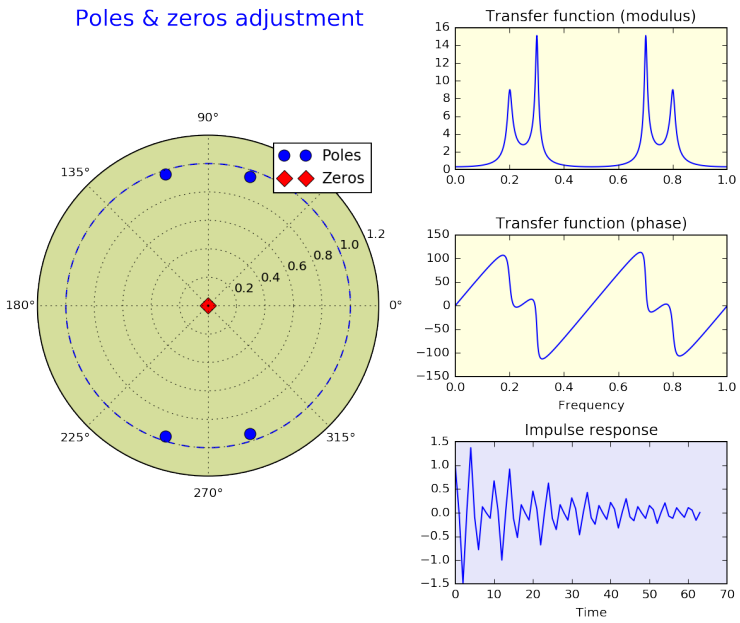



Figure 12.8: Poles-Zeros representation, Transfer function and Impulse response for a 4 poles

### 12.1.3 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)
```

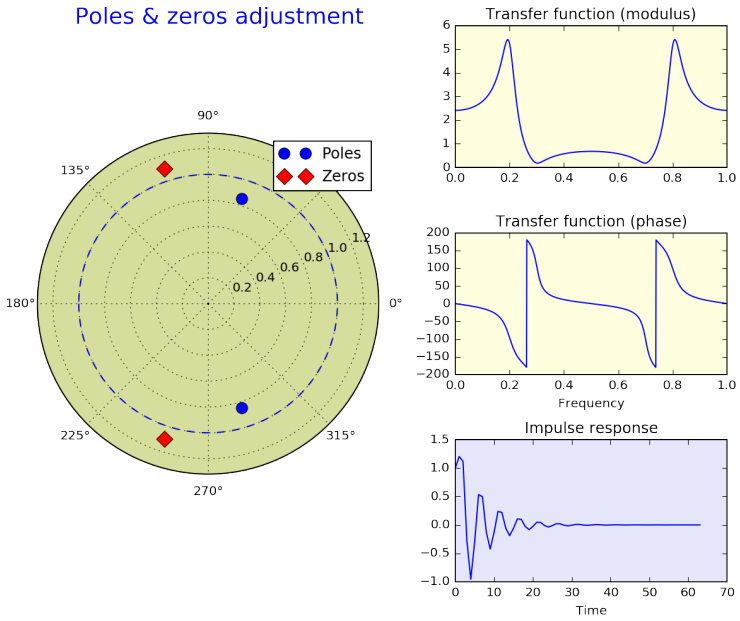



Figure 12.9: Poles-Zeros representation, Transfer function and Impulse response for a 2 poles and 2 zeros

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:

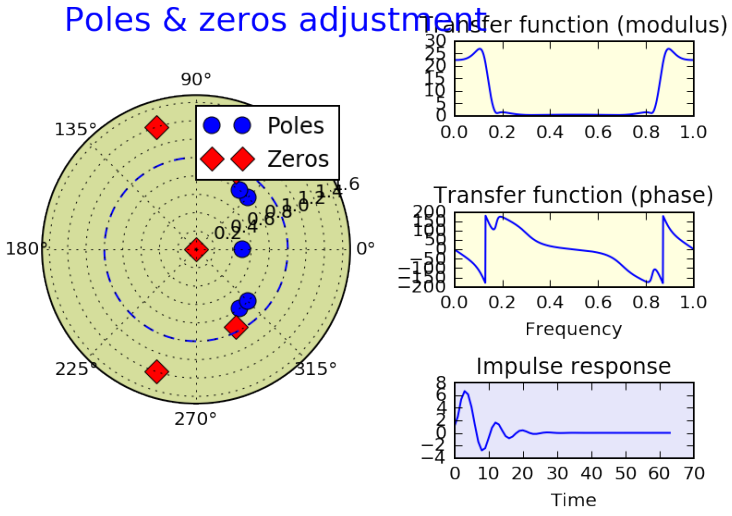
```
print("poles",A.poles)
print("zeros",A.zeros)

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
```




### 12.1.4 Appendix – listing of the class ZerosPolesPlay

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

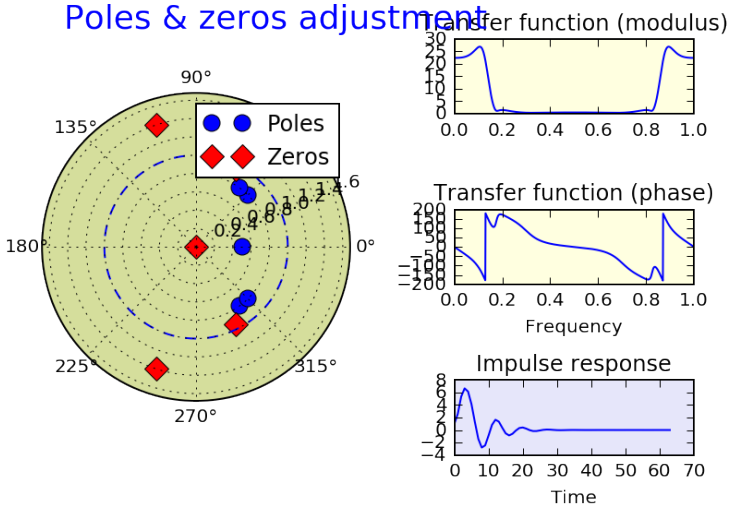
```
"""
Transfer function adjustment using zeros and poles drag and drop!
jfb 2015 – last update november 22, 2015
"""
import numpy as np
import matplotlib.pyplot as plt
from numpy import pi


#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,
        axisbg='#d5de9c')
    #self.ax=self.fig.add_subplot(221,polar=True, axisbg='#d5de9c')
    self.ax = plt.subplot(gs[0:,0:6],polar=True,axisbg='#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,axisbg='LightYellow')
    self.axTF0= plt.subplot(gs[0,6:11],axisbg='LightYellow')
    #self.axTF[0].set_axis_bgcolor('LightYellow')
    self.axTF0.set_title('Transfer function (modulus)')
    #self.axTF1=self.fig.add_subplot(224,axisbg='LightYellow')
    self.axTF1=plt.subplot(gs[1,6:11],axisbg='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,axisbg='Lavender')
    self.axIR =  plt.subplot(gs[2,6:11],axisbg='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
```

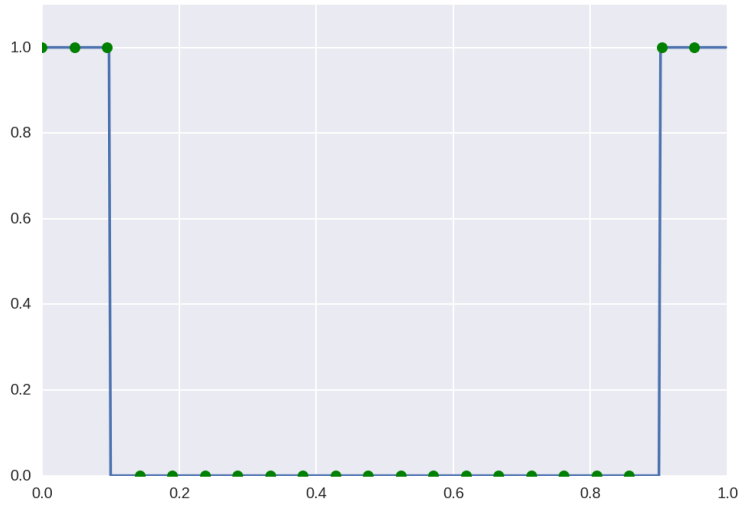


## 12.2 Synthesis of FIR filters

### 12.2.1 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.

```
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])
```




The associated impulse response is given by the inverse DFT. It is represented on figure 12.10.

```
%precision 3
# The impulse response:
h=real(ifft(rs))
print("Impulse response h:",h)
plt.plot(h)
plt.title("Impulse response")
```

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

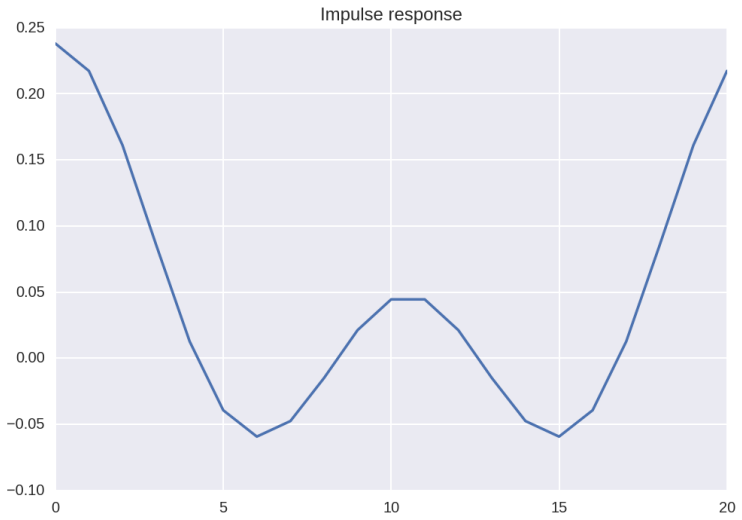
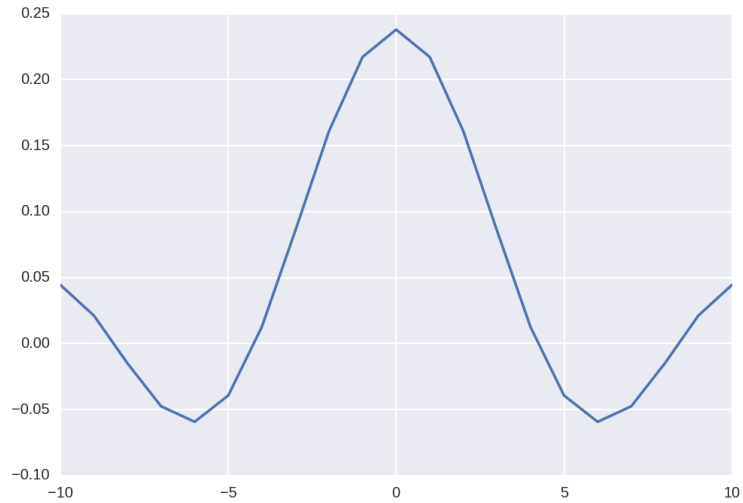



Figure 12.10: Impulse response obtained by frequency sampling

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))
```



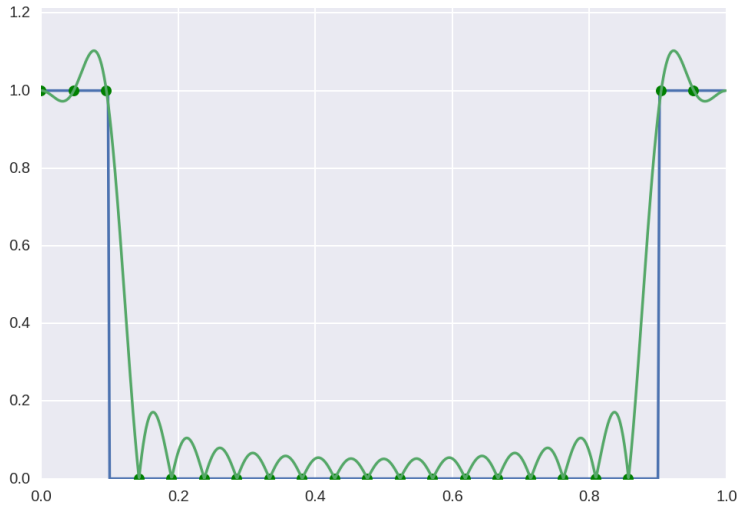

It is very instructive to look at the frequency response which is effectively realized. In other words we must look at what happens between the points. For that, we approximate the discrete time Fourier transform by zero-padding. At this point, it is really important to shift the impulse response because 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.

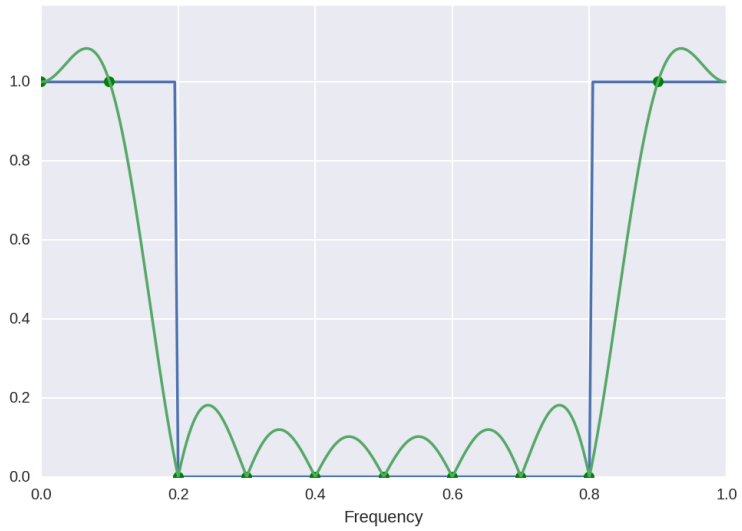
```
#ideal filter
plt.plot(np.arange(0,N)/N, (r))
#sampling the ideal filter
plt.plot(np.arange(0,N,step)/N, (rs), 'og')
_=plt.ylim([0, 1.1])
_=plt.xlim([0, 1])
#realized filter
_=plt.plot(np.arange(0,NN)/NN, np.abs(H))
_=plt.ylim([0, 1.1*np.max(np.abs(H))])
```




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.

```
def wLP_synth_fsampling():
    fc=fcw.value;
    L=Lw.value;
    plot_impresp=imprespw.value;
    LP_synth_fsampling(fc, L,plot_impresp)


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,impraspw])
d=widgets.VBox(children=[fcw,Lw,impraspw])
d.align="center"
d.box_style="info"
d.width="70%"
d.border_radius=20
display(d)
```

### 12.2.2 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:

$$H(f) \to h(n). \tag{12.6}$$

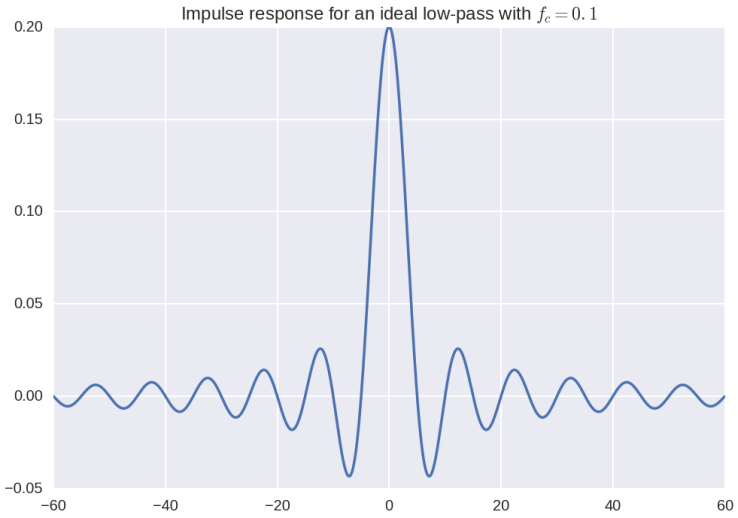
- Of course, this step would require by hand calculations, or a symbolic computation system. This leads to many exercises for students in traditional signal processing.
- In practice, one often begins with a precise numerical representation of the ideal filter and obtain the impulse response by IDFT. 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

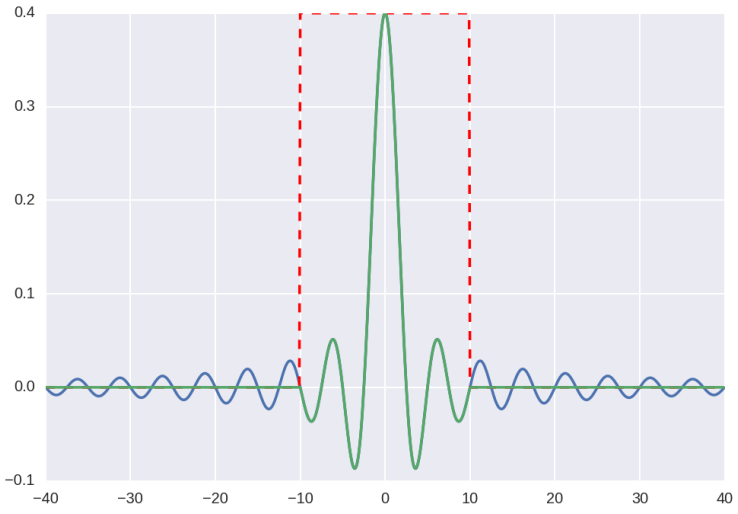
$$h(n) = 2f_c \mathrm{sinc}\,(2\pi f_c n). \tag{12.7}$$

```
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))
```



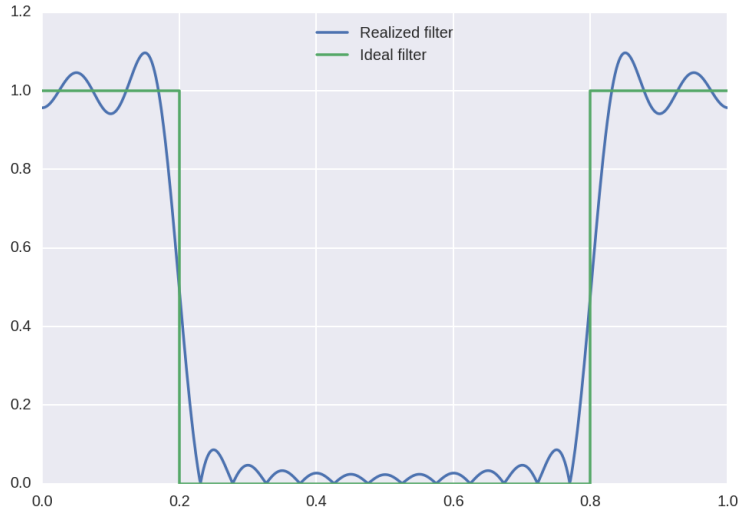

In order to get a finite number of points for our filter, we have no other solution but truncate the impulse response. Beware that one (you) need to keep both the positive and negative indexes. To get a causal system, it then suffices to shift the impulse response by the length of the non causal part. 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)
```



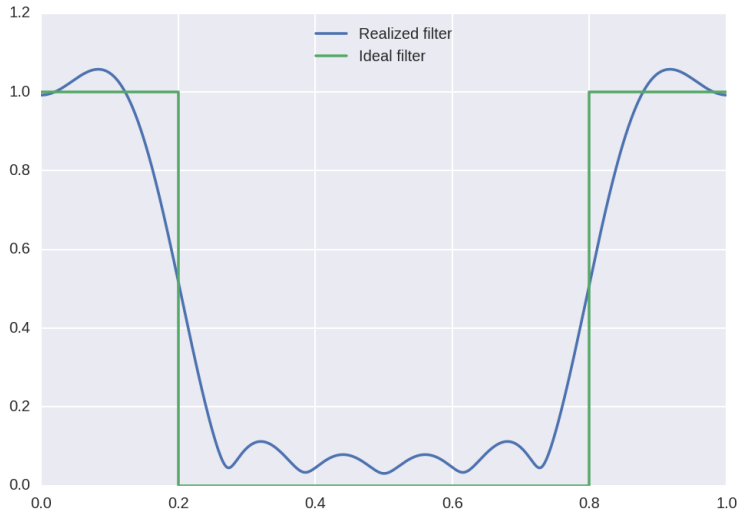

Then the realized transfer function can be computed and compared with the ideal filter.

```
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:

$$h_w(n) = h(n)w(n). \tag{12.8}$$

By the Plancherel theorem, we immediately get that

$$H_w(h) = [H * W](f). \tag{12.9}$$

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)

$$W(f) = \frac{\sin(\pi f(2L+1))}{(2L+1)\sin(\pi f)}. \tag{12.10}$$

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 | $\mathbf{w(n)}$ | $\mathbf{\Delta f}$ | 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. $\Delta f$ 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

$$w(n) = \sum_{k=0}^{3} a_k \cos\left(\frac{2\pi k n}{M-1}\right) \tag{12.11}$$

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

$$w(n) = I_0\left(\beta\sqrt{1 - \frac{4n^2}{(M-1)^2}}\right)/I_0(\beta), \tag{12.12}$$

where $I_0$ is the modified zeroth-order Bessel function. The shape parameter $\beta$ determines a trade-off between main-lobe width and side lobe level. As $\beta$ 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
```

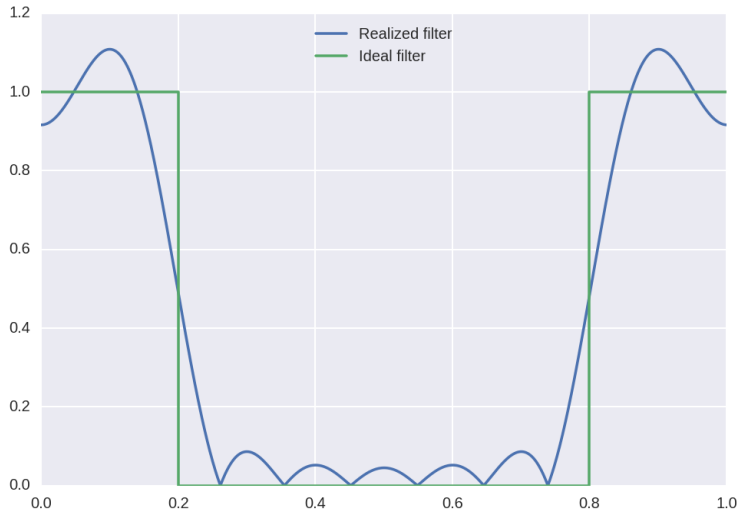
The main observation is that with $N$ fixed, we have a trade-off to find between the width of the main lobe, thus of the transition width, and the amplitude of the side-lobes. The choice is usually

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.

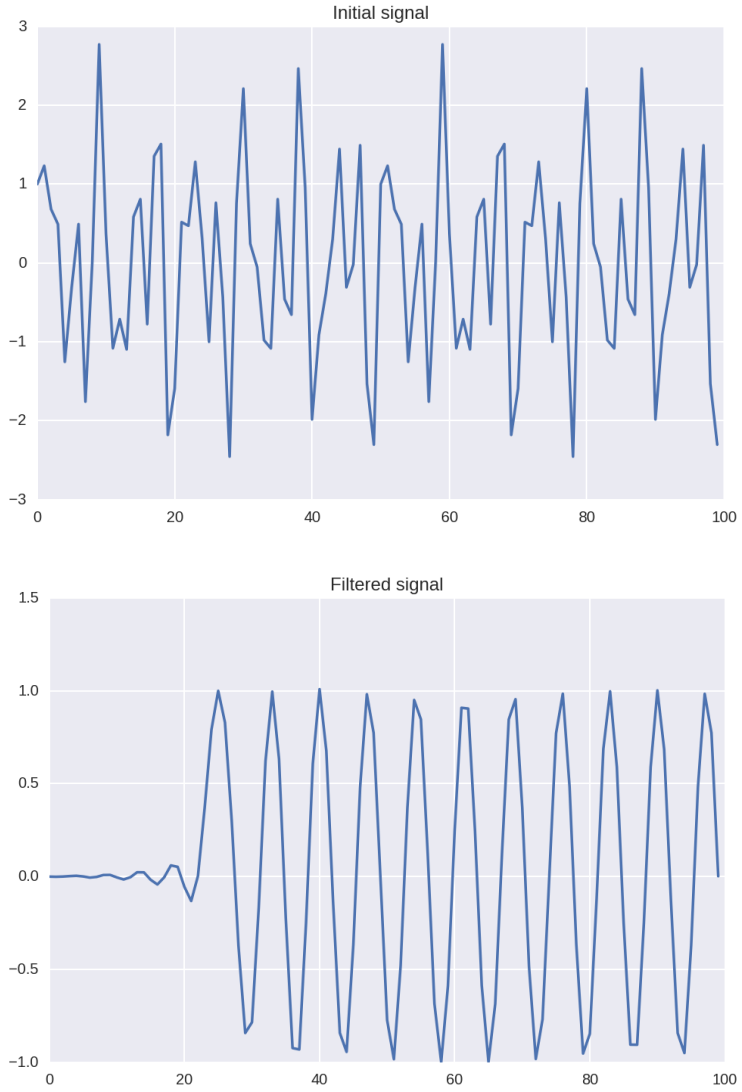
It is quite simple to adapt the previous script with the rectangular window to accept more general windows. This is done by adding a parameter `window`.

```
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=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")
```




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
xtest=sin(2*pi*f0*n)+sin(2*pi*f1*n)+sin(2*pi*f2*n)
plt.plot(xtest)
plt.title("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")

# then filter the signal
y1=sig.lfilter(h1,[1],xtest)
#and display it
plt.figure()
plt.plot(y1)
plt.title("Filtered signal")
```

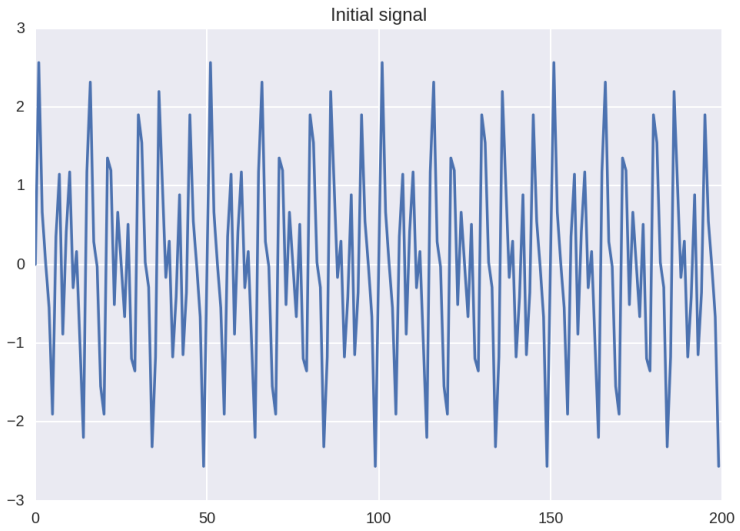



Figure 12.11: Initial signal

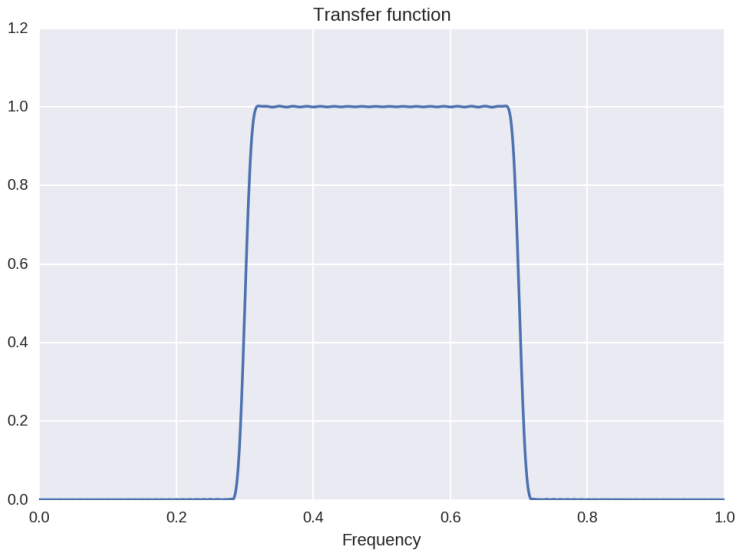



Figure 12.12: Transfer function

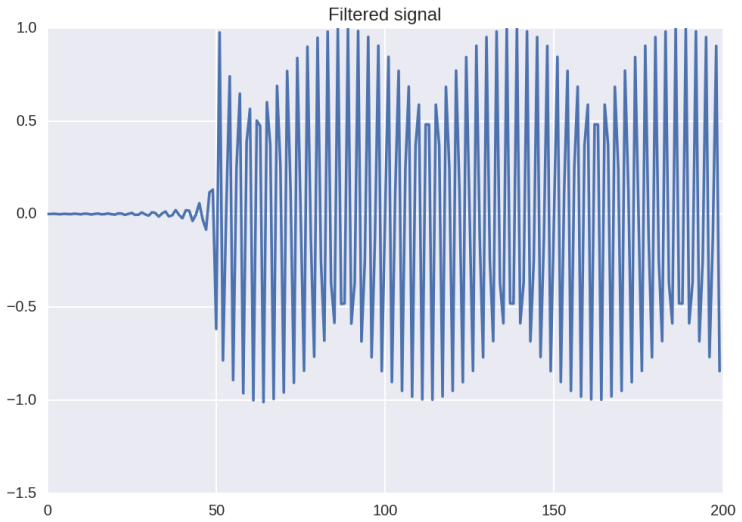



Figure 12.13: Filtered signal

## 12.3 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 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.

### 12.3.1 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:

$$p = k\frac{1 - z^{-1}}{1 + z^{-1}}, \tag{12.13}$$

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 (12.13), we readily obtain

$$\boxed{\omega_a = k \tan\left(\frac{\omega_d}{2}\right)}, \tag{12.14}$$

or

$$\boxed{\omega_d = 2 \arctan\left(\frac{\omega_a}{k}\right)}. \tag{12.15}$$

The transformation (12.14) corresponds to the initial transform of the specifications in the digital domain into analog domain specifications. It is often called a *pre-warping* . Figure 12.14 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")
```

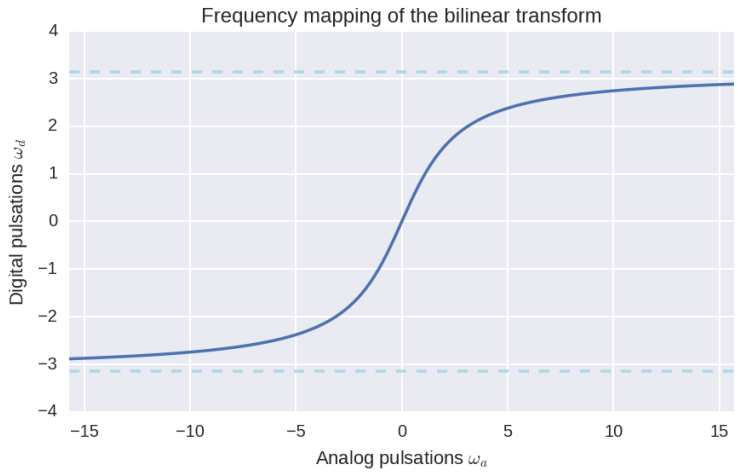



Figure 12.14: Frequency mapping of the bilinear transform

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).

### 12.3.2 Synthesis of low-pass filters – procedure

Let $\omega_p$ and $\omega_s$ 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($\omega\_p/2$)]$which ichensures that \Omega_p = 1$. 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 (12.13), that is compute [ $\boxed{H(z) = H_a(p)|_{p=k\frac{1-z^{-1}}{1+z^{-1}}}}$ ]

**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) = $1+2z^{-1} + z^{-2}\frac{}{(4+\sqrt{6})-4z^{-1}+(4-\sqrt{6})z^{-2}]}$

```
# 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")
```

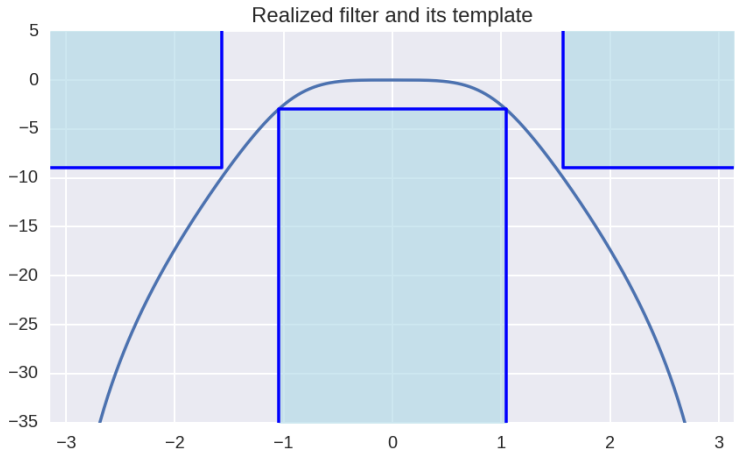



Figure 12.15: Realized filter and its template

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

$$z = \frac{1 + p/k}{1 - p/k}. \tag{12.16}$$

It remains a global gain that can be fixed using a value at $H(1)$ ($\omega = 0$).This dramatically simplifies the synthesis in practice.

### 12.3.3 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 ($\omega_p$) to another type

Let $\omega_1$ and $\omega_2$ denote the low and high cut-off frequencies (only $\omega_1$ 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 $\omega_p$– low-pass $\omega_1$**

$$z^{-1} \to \frac{z^{-1} - \alpha}{1 - \alpha z^{-1}} \tag{12.17}$$

with

$$\alpha = \frac{\sin\left(\frac{\omega_p - \omega_1}{2}\right)}{\sin\left(\frac{\omega_p + \omega_1}{2}\right)} \tag{12.18}$$

3. **low-pass $\omega_p$– band-pass $\omega_1, \omega_2$** $z^{-1} \to -\frac{z^{-2} - \frac{2\alpha\beta}{\beta+1} z^{-1} + \frac{\beta-1}{\beta+1}}{\frac{\beta-1}{\beta+1} z^{-2} - \frac{2\alpha\beta}{\beta+1} z^{-1} + 1}$ with $\alpha = \frac{\cos\left(\frac{\omega_p +$

**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.*

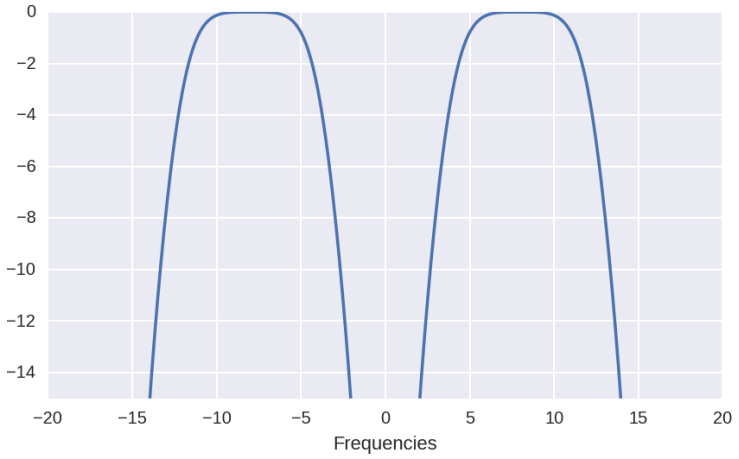
** Sketch of solution** - normalized pulsations: $\omega_0 = \frac{\pi}{8}, \omega_1 = \frac{\pi}{4}, \omega_2 = \frac{3\pi}{4}, \omega_3 = \frac{7\pi}{8}$ - transposition into a digital low-pass $\alpha = 0, \beta = \tan(\omega_p/2)$. Choosing $\beta = 1$, we get $\omega_p = \pi/2$ - the transform is 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

$$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{(2 + \sqrt{2}) + (2 - \sqrt{2})z^{-2}} \tag{12.19}$$

for the digital low-pass, which after the transform $z^{-1} \to -z^{-2}$ gives

$$H(z) = \frac{1 - 2z^{-2} + z^{-4}}{(2 + \sqrt{2}) + (2 - \sqrt{2})z^{-4}} \tag{12.20}$$

```
# 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])
```




### 12.3.4 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")
```

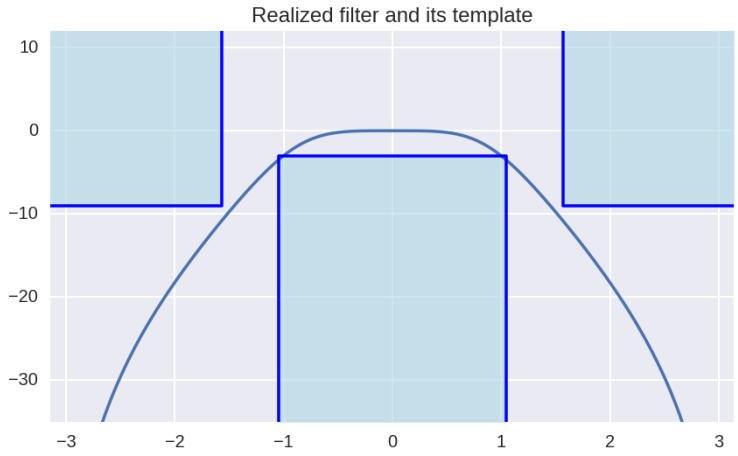
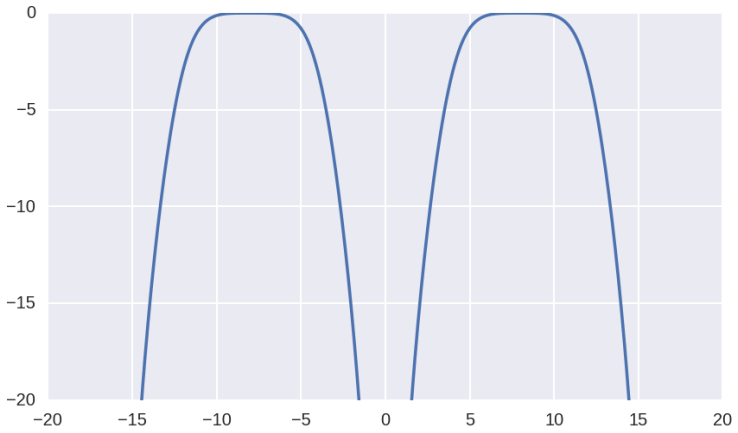



Figure 12.16: Realized filter and its template

```
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])
```




## 12.4 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 . It is possible to load it via the instruction `f=np.load(sig1.npz)`

```
Fs=8000
Ts=1/Fs
```

First load all useful modules:

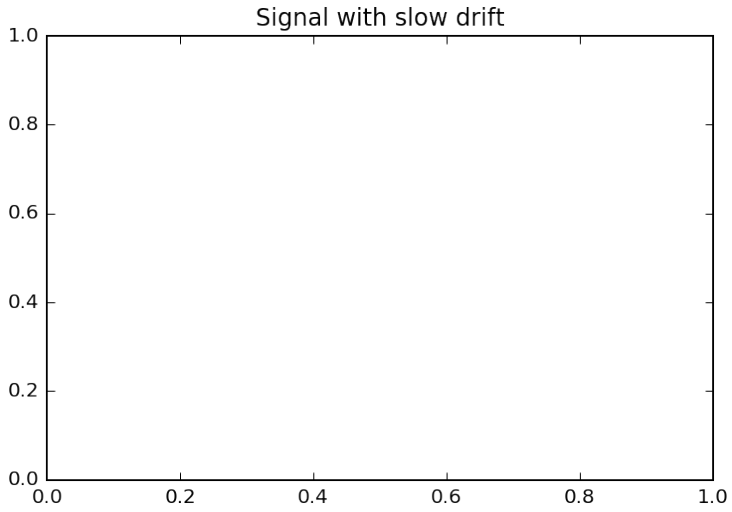
```
import numpy as np
from numpy import ones, zeros, abs, exp, pi, sin, real, imag
import matplotlib.pyplot as plt
import scipy.io
from scipy.signal import lfilter
from numpy.fft import fft, ifft, fftshift

%matplotlib inline
```

### 12.4.1 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.title('Signal with slow drift')
```




```
# Frequency representation
plt.figure(2)
N=len(x)
f=freq(N)
#plt.plot(f,...)
#plt.title('Fourier transform of the signal (modulus)')
```

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

AttributeError                            Traceback (most recent call last)

/usr/local/lib/python3.5/site-packages/IPython/core/formatters.py in __call__(self, obj)
335                 pass
336             else:
--> 337                 return printer(obj)
338             # Finally look for special method names
339             method = _safe_get_formatter_method(obj, self.print_method)


/usr/local/lib/python3.5/site-packages/IPython/core/pylabtools.py in <lambda>(fig)
207         png_formatter.for_type(Figure, lambda fig: print_figure(fig, 'png', **kwargs
208     if 'retina' in formats or 'png2x' in formats:
```

```
--> 209         png_formatter.for_type(Figure, lambda fig: retina_figure(fig, **kwargs))
    210     if 'jpg' in formats or 'jpeg' in formats:
    211         jpg_formatter.for_type(Figure, lambda fig: print_figure(fig, 'jpg', **kwargs


    /usr/local/lib/python3.5/site-packages/IPython/core/pylabtools.py in retina_figure(fig,
    124     """format a figure as a pixel-doubled (retina) PNG"""
    125     pngdata = print_figure(fig, fmt='retina', **kwargs)
--> 126     w, h = _pngxy(pngdata)
    127     metadata = dict(width=w//2, height=h//2)
    128     return pngdata, metadata


    /usr/local/lib/python3.5/site-packages/IPython/core/display.py in _pngxy(data)
    607 def _pngxy(data):
    608     """read the (width, height) from a PNG header"""
--> 609     ihdr = data.index(b'IHDR')
    610     # next 8 bytes are width/height
    611     w4h4 = data[ihdr+4:ihdr+12]


    AttributeError: 'NoneType' object has no attribute 'index'
```

### 12.4.2 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 $H(\mathrm{e}^{j2\pi f/Fs})$ in modulus and phase, for a filter described by the two vectors $B$ and $A$: `freqz(B,A)`

### 12.4.3 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
#...
# ...
#plt.plot(t,x,t,m_estimated)
#plt.title('Signal and estimated drift')
#
# We check G(f)
#G=...
#plt.xlabel('Normalized fequencies')
#plt.title('Transfer Function of the Averaging Filter')
```

### 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
#...
#plt.title('Signal with removed drift')
#plt.show()

#
#plt.xlabel('Frequencies')
#plt.xlim([-0.5, 0.5])
#plt.title('Fourier transform of the signal with removed drift')


#We check H(f)
#...
#plt.xlabel('Normalized fequencies')
#plt.title('Transfer Function of the Subtracting Filter')
```

### 12.4.4 Second part: Boost of a frequency band

We wish now to boost a range of freqencies aound 1000 Hz on the initial signal.

## 12.5 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')
```

### 12.5.1 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-15-2696f2851cc3> 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.

# 13

# Lab – Basic Filtering

Author: J.-F. Bercher date: november 19, 2013 Updates: february 25, 2014, december 08, 2014 Last update: november 21, 2015

## 13.1 Introduction

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 . 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
```

## 13.2 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
```

For the record, the following lines enable to read a Matlab `.mat` file and save the data in the NumPy format

```
mat = scipy.io.loadmat('sig1.mat') # Actually mat is a dict
mat.keys()   # whose keys are given by  .keys()
x=mat['x'].flatten()
m=mat['m'].flatten()
## We save data using the compressed numpy format
numpy.savez('sig1',x=x, m=m)
```

```
# 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']
```

We can plot the signal, in time and frequency, by the following lines

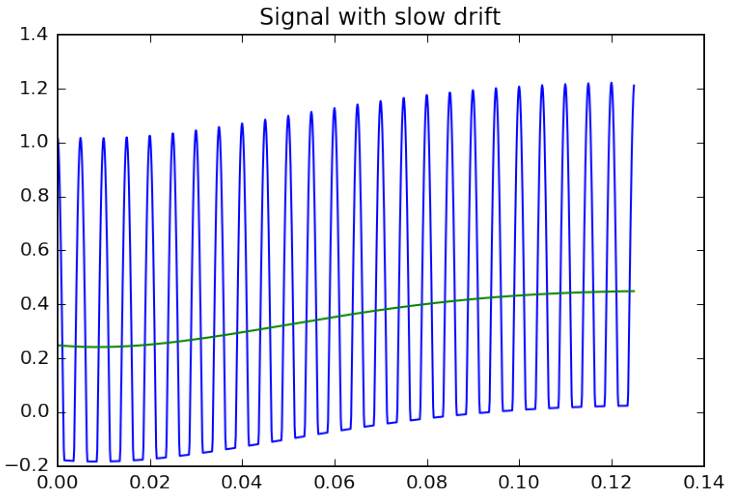
```
 # Loading

sig1=np.load('sig1.npz')
#sig1 is a dictionnary. One can look at the keys by: sig1.keys()
m=sig1['m']
x=sig1['x']

Fs=8000
Ts=1/Fs

# Time representation
plt.figure(1)
t=np.np.arange(len(x))*Ts
plt.plot(t,x,t,m)
plt.title('Signal with slow drift')
plt.show()
```

and

```
# Frequency representation
plt.figure(2)
N=len(x)
f=freq(N)
plt.plot(f,abs(fftshift(fft(x))))
plt.xlim([-0.5, 0.5])
plt.title('Fourier transform of the signal (modulus)')
```

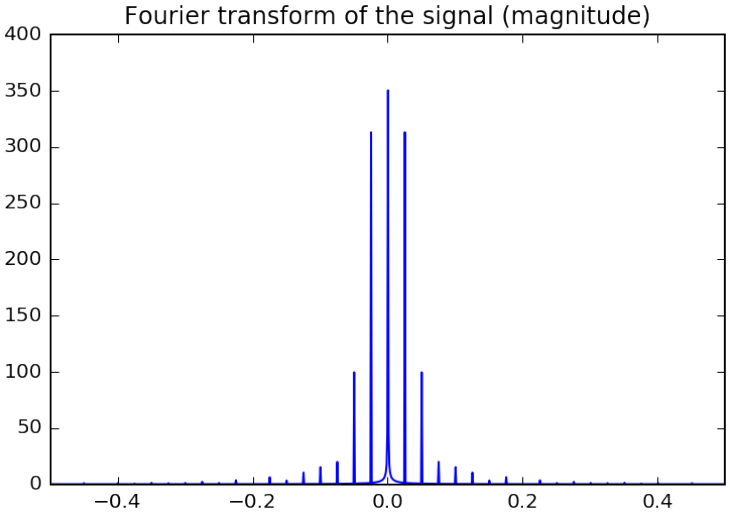
Analyze and comment these figures.

```
# Time
plt.figure(1)
t=np.arange(len(x))*Ts
plt.plot(t,x,t,m)
plt.title('Signal with slow drift')
plt.show()
```




We observe an approximately periodic time series, with period ~ 40, and a slow drift of the mean value

```
# Frequency representation
plt.figure(2)
N=len(x)
f=freq(N) #f=freq(N,Fs) if we want the representation in real frequencies
plt.plot(f,np.abs(fftshift(fft(x))))
plt.xlim([-0.5, 0.5])
plt.title('Fourier transform of the signal (magnitude)')
plt.show()
```




We see that the Fourier transform consists of a series of regularly spaced spectral lines, which is the indication that the underlying signal is periodic. The peaks are spaced-out by 0.025, in normalized frequencies. This effectively corresponds to a period of 40 observed on the time representation.

### 13.2.1 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 $H(\mathrm{e}^{j2\pi f/Fs})$ in modulus and phase, for a filter described by the two vectors $B$ and $A$: `freqz(B,A)`

## 13.3 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.

### 13.3.1 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 $h(n)$, which 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 $H(z)$.

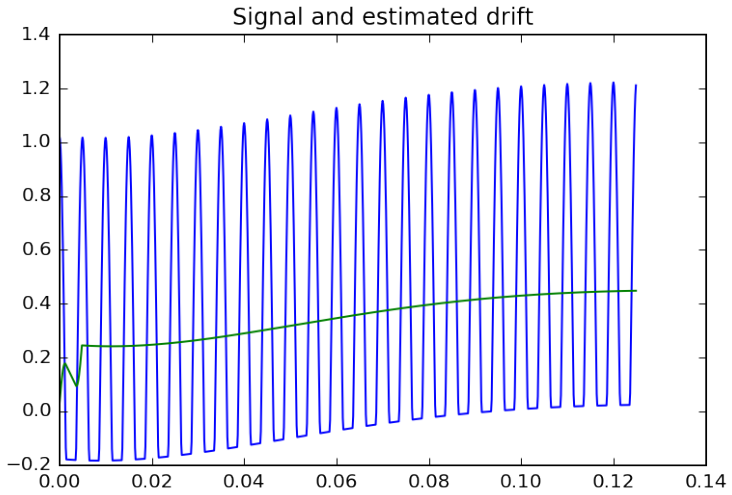
### 13.3.2 Practical part

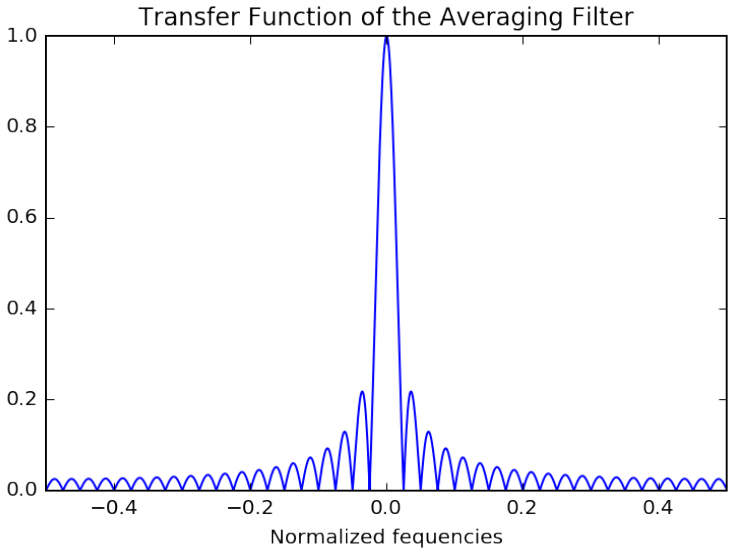
For the averaging filter and then for the subtracting filter:

- 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
g=np.ones(L)/L
m_estimated=lfilter(g,1,x)
```

```
plt.figure(3)
plt.plot(t,x,t,m_estimated)
plt.title('Signal and estimated drift')
plt.show()
#
# We check G(f)
G=fftshift(fft(g,1000))
plt.figure(4)
plt.plot(freq(1000),abs(G))
plt.xlim([-0.5, 0.5])
plt.xlabel('Normalized fequencies')
plt.title('Transfer Function of the Averaging Filter')
```

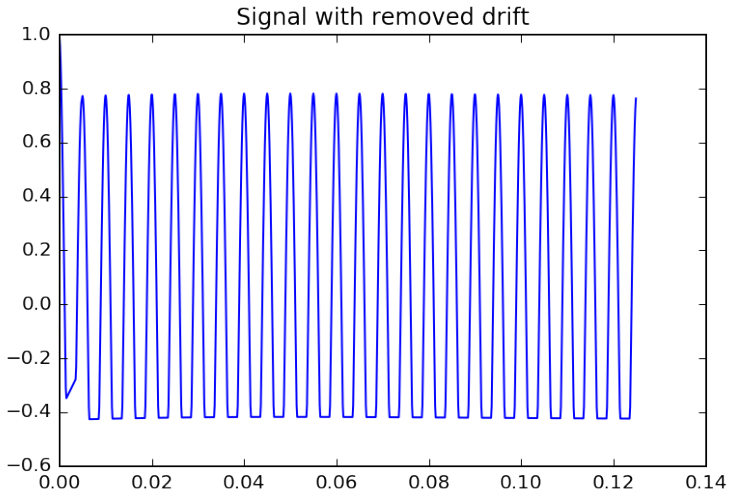


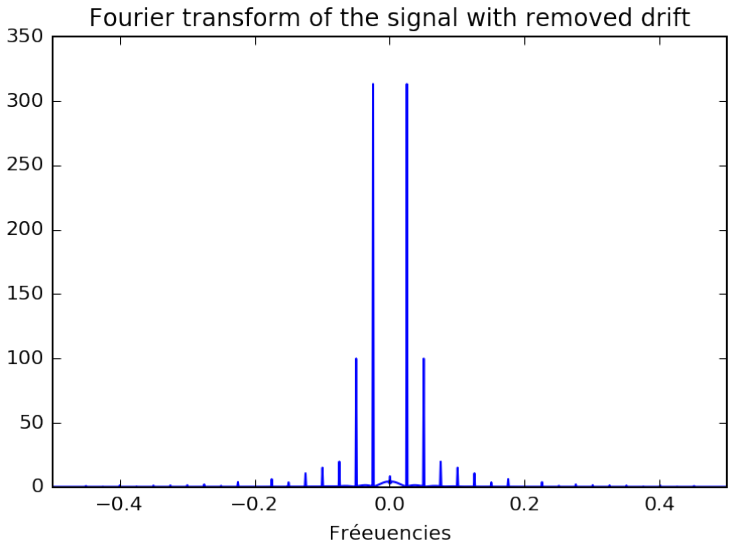



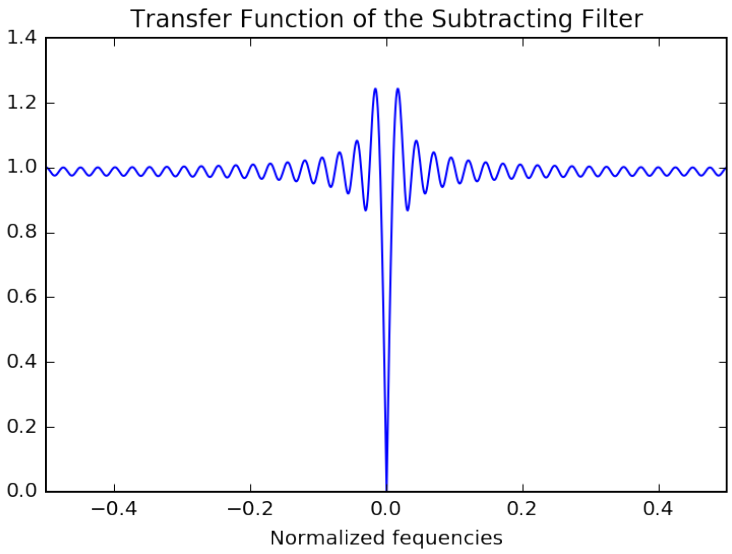

We shall note, and this is very nice, that this Fourier transform has the the good idea to cancel out every $1/L$: Thus the peaks in the frequency representation of the periodic signal are perfectely removed. It only remains the frequency part around $f = 0$.

## 13.4 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
h=-np.ones(L)/L
h[0]=1-1/L
# or via the operation
# $xc(n)=x(n)-(g*x)(n)$
xc=lfilter(h,1,x)
plt.figure(5)
plt.plot(t,xc)
plt.title('Signal with removed drift')
plt.show()

plt.figure(6)
plt.plot(freq(len(xc)),abs(fftshift(fft(xc))))
plt.xlabel('Fréeuencies')
plt.xlim([-0.5, 0.5])
plt.title('Fourier transform of the signal with removed drift')
plt.show()

#We check H(f)
H=fftshift(fft(h,1000))
plt.figure(7)
plt.plot(freq(1000),abs(H))
plt.xlabel('Normalized fequencies')
plt.xlim([-0.5, 0.5])
plt.title('Transfer Function of the Subtracting Filter')
```






## 13.5 Second part: Boost of a frequency band

We wish now to boost a range of frequencies aound 1000 Hz on the initial signal.

## 13.6 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 $h(n)$.

**Answer**: Derivation of the impulse response by selecting a pair of complex conjugated poles for the frequency of interest. The theoretical impulse response can be computed as follows: Consider a transfer function with two poles $p_0$ and $p_1$:

$H(z) = \frac{1}{(1-p_0 z^{-1})(1-p_1 z^{-1})}$
By Heaviside partial-fraction expansion, we have

$$H(z) = \frac{A}{1-p_0 z^{-1}} + \frac{B}{1-p_1 z^{-1}} \tag{13.1}$$

with $A = \frac{p_0}{p_0-p_1}$ and $B = -\frac{p_1}{p_0-p_1}$. We thus have

$$H(z) = \frac{1}{p_0-p_1}\left(\frac{p_0}{1-p_0 z^{-1}} - \frac{p_1}{1-p_1 z^{-1}}\right) \tag{13.2}$$

The inverse z-transform of $1/(1-az^{-1})$ is $a^n u(n)$, where $u(n)$ is the Heaviside step function. Therefore, we get that

$$h(n) = \frac{1}{p_0-p_1}\left(p_0^{n+1} - p_1^{n+1}\right) u(n). \tag{13.3}$$

Finally, if we take two complex conjugated poles $p_0 = p_1^* = \rho e^{j2\pi f_0}$, it comes

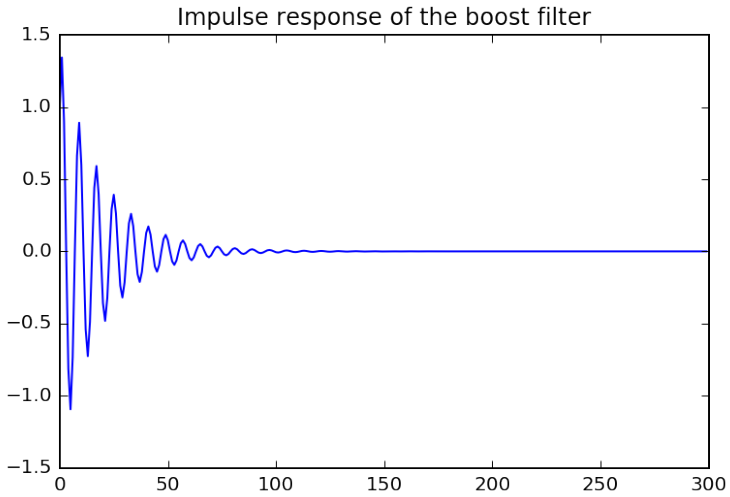
$$h(n) = \rho^n \frac{\sin(2\pi(n+1)f_0)}{\sin(2\pi f_0)} u(n). \tag{13.4}$$

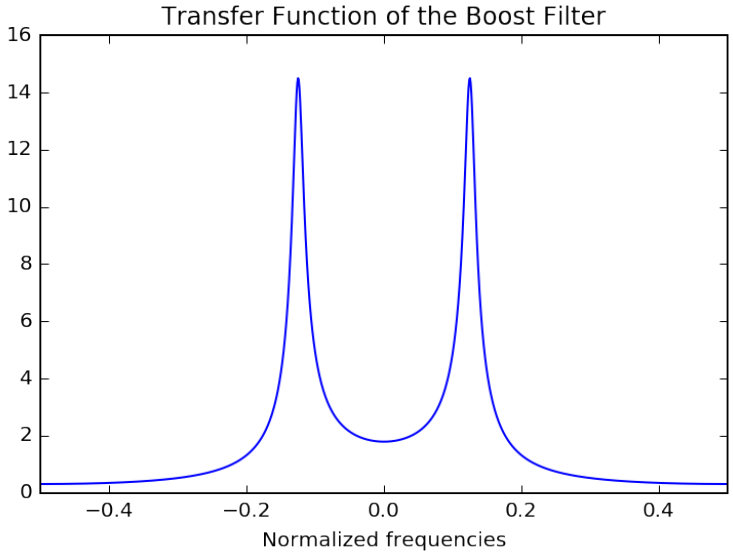
### 13.6.1 Pratical part

- 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[0]=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.

```
rho=0.95
fo, Fs=1000, 8000
a=np.poly([rho*exp(1j*2*pi*fo/Fs), rho*exp(-1.j*2*pi*fo/Fs)])


# Compute the IR
d=zeros(300)
d[0]=1
h_accentued=lfilter([1],a,d) # calcul de la RI
plt.figure(8)
plt.plot(h_accentued)
plt.title('Impulse response of the boost filter')

# en fréquence
H_accentued=fftshift(fft(h_accentued,1000))
```

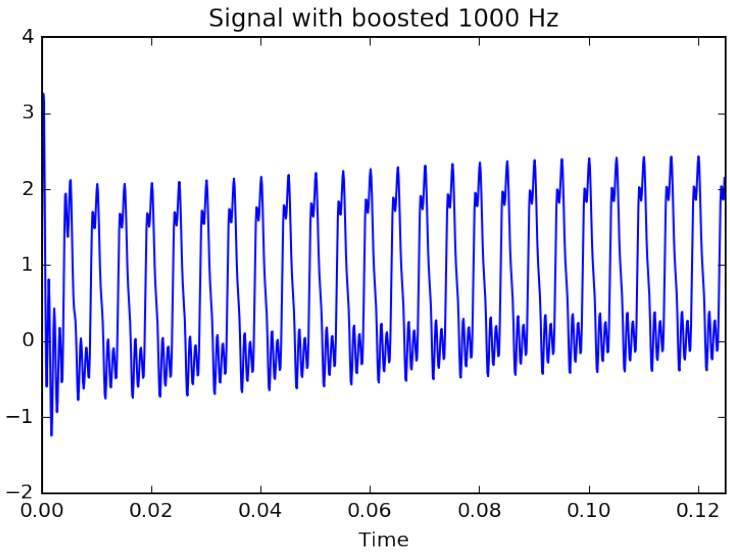
```
plt.figure(9)
plt.plot(freq(1000),abs(H_accentued))
plt.xlabel('Normalized frequencies')
plt.xlim([-0.5, 0.5])
plt.title('Transfer Function of the Boost Filter')
```

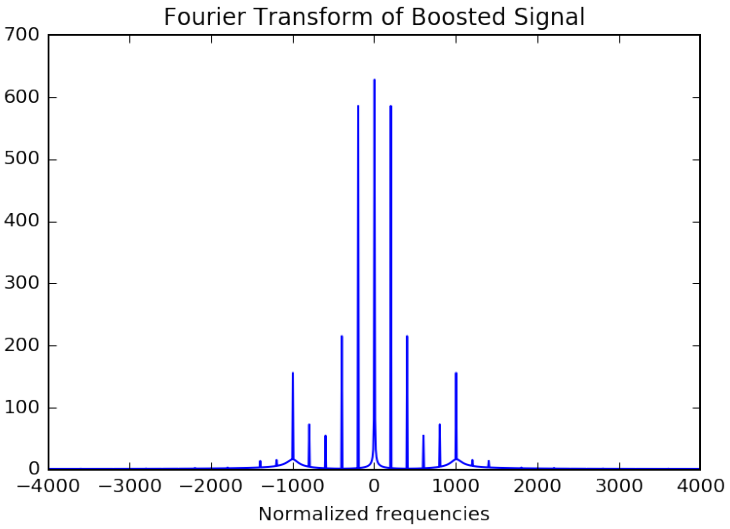



```
# Filtering
sig_accentuated=lfilter([1],a,x) #
plt.figure(10)
plt.plot(t,sig_accentuated)
plt.xlabel('Time')
plt.xlim([0, len(x)*Ts])
plt.title('Signal with boosted 1000 Hz')

# In the frequency domain
S_accentuated=fftshift(fft(sig_accentuated,1000))
plt.figure(11)
plt.plot(freq(1000,Fs),abs(S_accentuated))
```

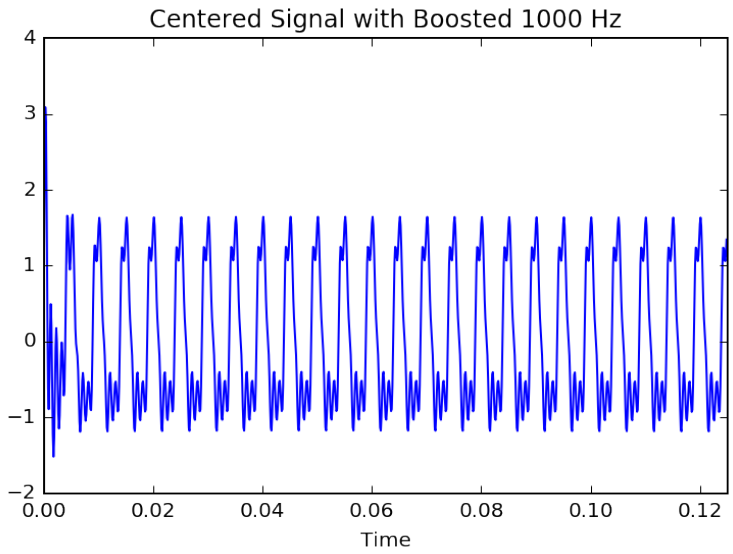
```
plt.xlabel('Normalized frequencies')
plt.xlim([-Fs/2, Fs/2])
plt.title('Fourier Transform of Boosted Signal')
```




- How can we simultaneously boost around 1000 Hz and remove the drift? Propose a filter that performs the two operations.

```
# both filterings:
sig_accentuated_centered=lfilter(h,a,x) #
plt.figure(12)
plt.plot(t,sig_accentuated_centered)
plt.xlabel('Time')
plt.xlim([0, len(x)*Ts])
plt.title('Centered Signal with Boosted 1000 Hz')
```




## 13.7 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.

### 13.7.1 Theoretical Part

**Answers** For a rectangular window with width $2B$, the inverse Fourier transform is

$$h(n) = \int_{[1]} \text{rect}_{2B}(f) e^{j2\pi fn} \mathrm{d}f \tag{13.5}$$

$$= \int_{-B}^{B} e^{j2\pi fn} \mathrm{d}f \tag{13.6}$$

$$= 2B \frac{\sin(2\pi Bn)}{2\pi Bn} \tag{13.7}$$

The problems that appear are that

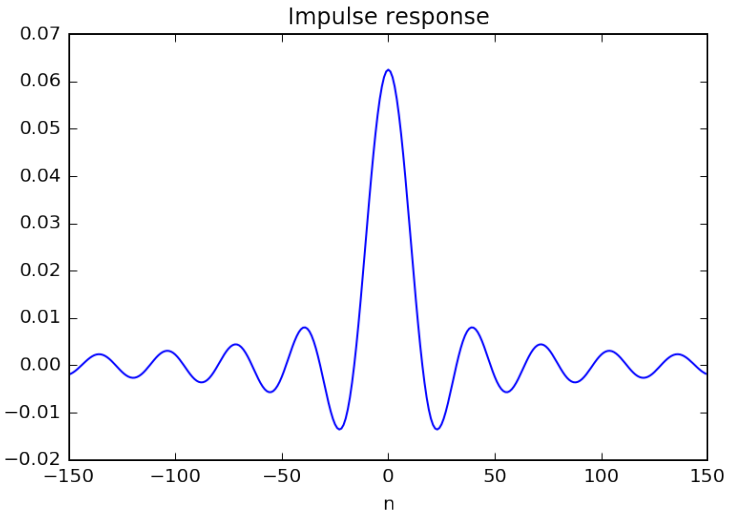
- the impulse response $h(n)$ has infinite suppport
- is non-causal (defined for $t < 0$)

So as to obtain an impulse response on $L$ points, it is thus necessary to

- troncate the impulse response, and then
- to delay it so as to obtain a causal response.

### 13.7.2 Practical part

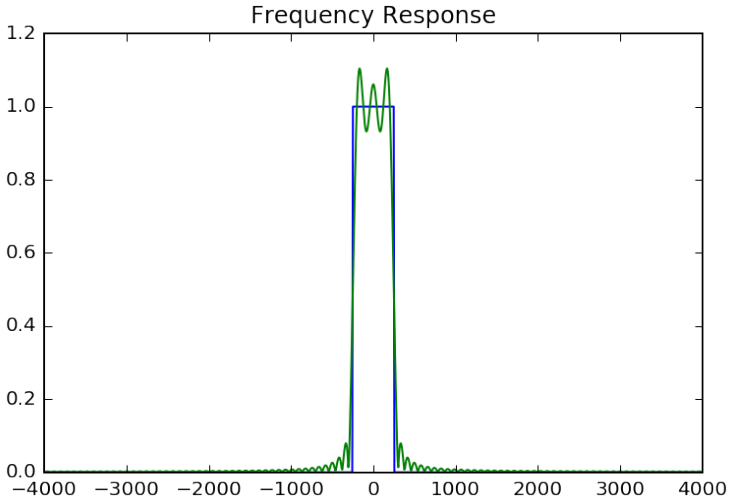
```
B=250
Fs=8000
B=B/Fs # Band in normalized fequencies
n=np.arange(-150,150)

def sinc(x):
    x=np.array(x)
    z=[sin(n)/n if n!=0 else 1 for n in x]
    return np.array(z)

#h=B*sin(pi*B*n)/(pi*B*n) #
#h[0]=B
h=2*B*sinc(pi*2*B*n)
plt.plot(n,h)
plt.xlabel('n')
plt.title('Impulse response')
```

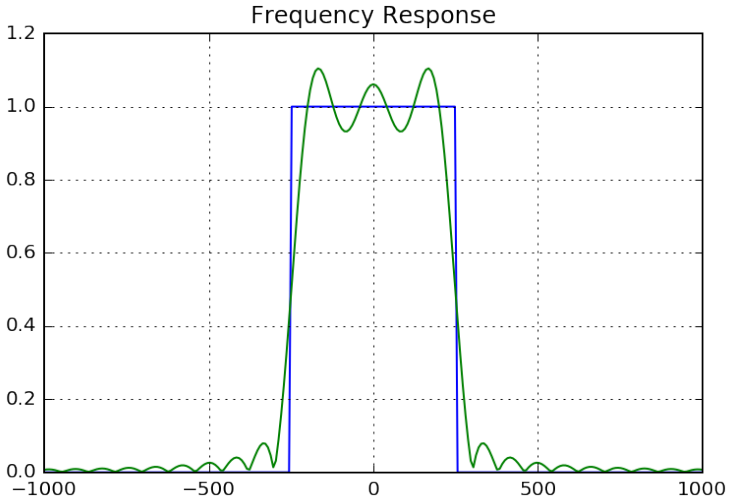



Trancation consists in keeping only the more interesting part; here, the samples between -50 and 50 (for a total of 101 points)

```
L=50
h_tronq=h=2*B*sinc(pi*2*B*np.arange(-L,L))
H_tronq=fft(h_tronq,1000)
f=(np.arange(1000)/1000-1/2)*8000
R=[1 if abs(ff)<250 else 0 for ff in f]
```

```
#
plt.figure()
plt.plot(f,R, f,abs(fftshift(H_tronq)))
plt.title("Frequency Response")
plt.figure()
plt.plot(f,R,f,abs(fftshift(H_tronq)))
plt.title("Frequency Response")
plt.xlim([-1000, 1000])
plt.grid(True)
```

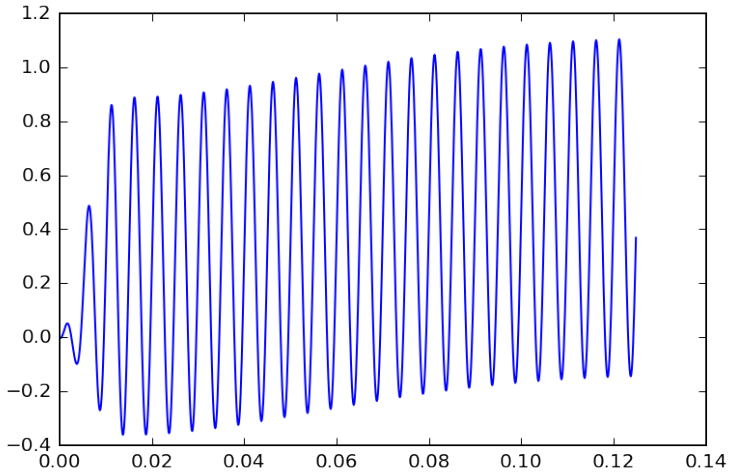



### 13.7.3 Output of the lowpass filter

```
sig_lowpass=lfilter(h,[1],x) #
plt.plot(t,sig_lowpass)
```

```
[<matplotlib.lines.Line2D at 0x7f77778e7400>]
```

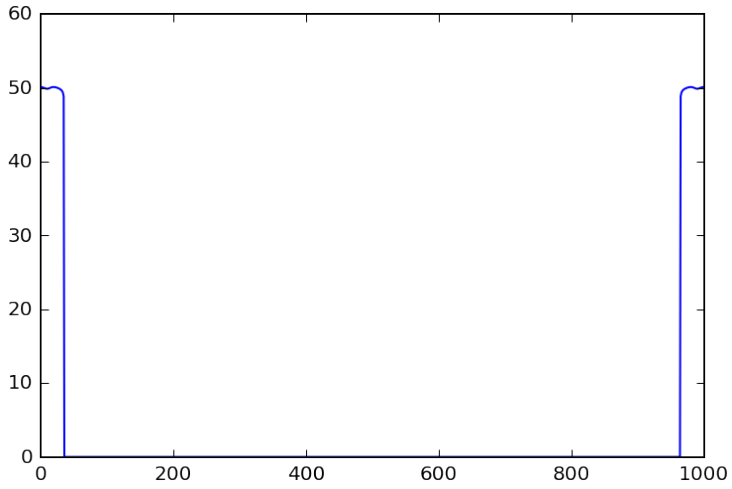



### 13.7.4 Group Delay

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)
```

```
[<matplotlib.lines.Line2D at 0x7f77777aff28>]
```




Thus we see that we have a group delay of about 50 in the band of the filter.
END.