A Journey in Signal Processing
with IPython

JEAN-FRANÇOIS BERCHER

ESIEE-PARIS

2

Contents

4

```
%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)
```

6

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

```python
# 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 (...)
```
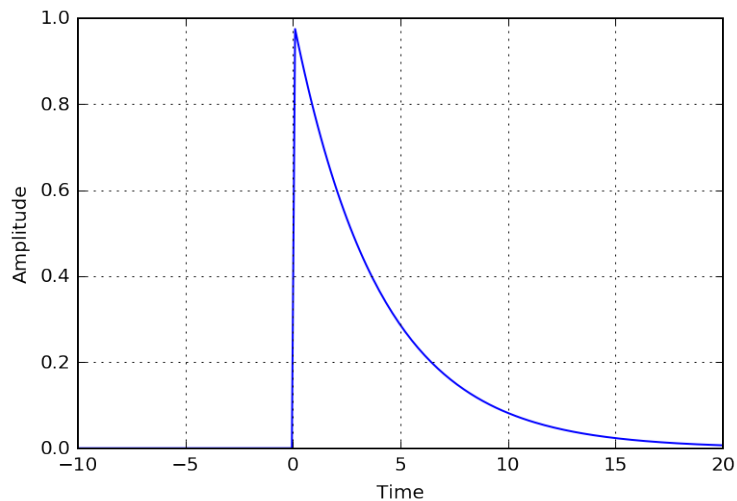
This to show that you do automatically several tests and plot the results all together.

```python
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()
```
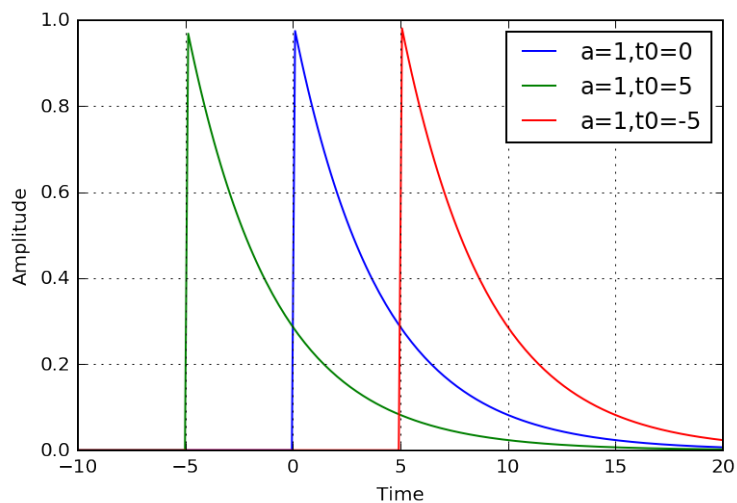

And finally an interactive version

```python
%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)
```
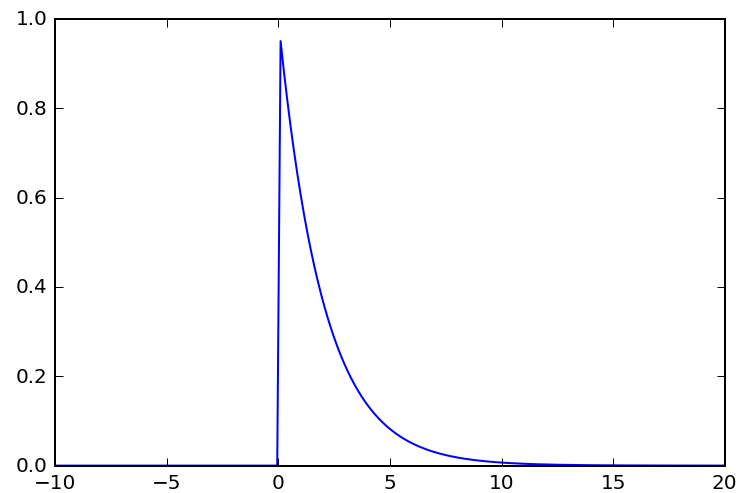


```
%run nbinit.ipy
```

```
... 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.

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

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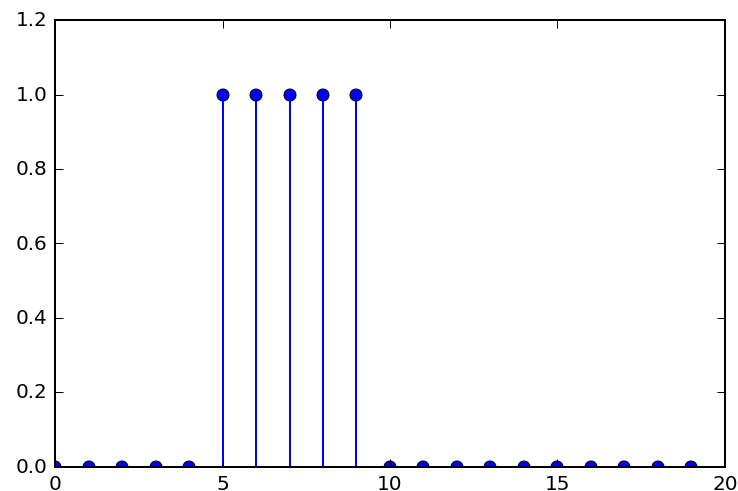


```python
def op1(signal):
    transformed_signal=np.zeros(np.size(signal))
    for  t  in  np.arange(np.size(signal)):
        transformed_signal[t]=signal[t]-signal[t-1]
    return  transformed_signal

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
```

```python
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")
```

Filtering of rectangular signal with op1

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()
```
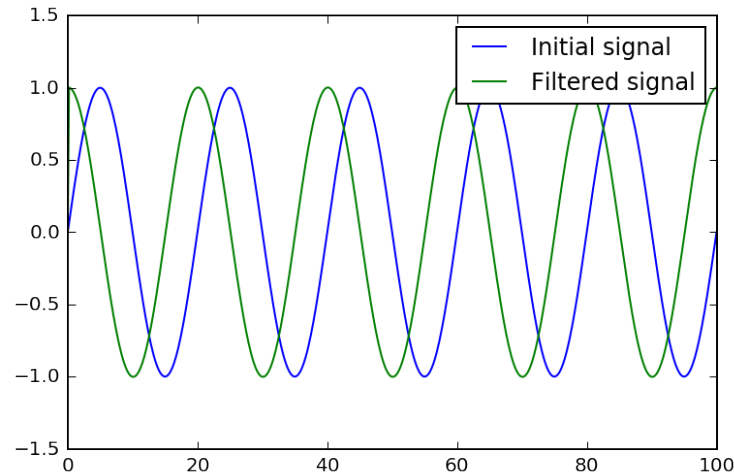
Composition of operations:

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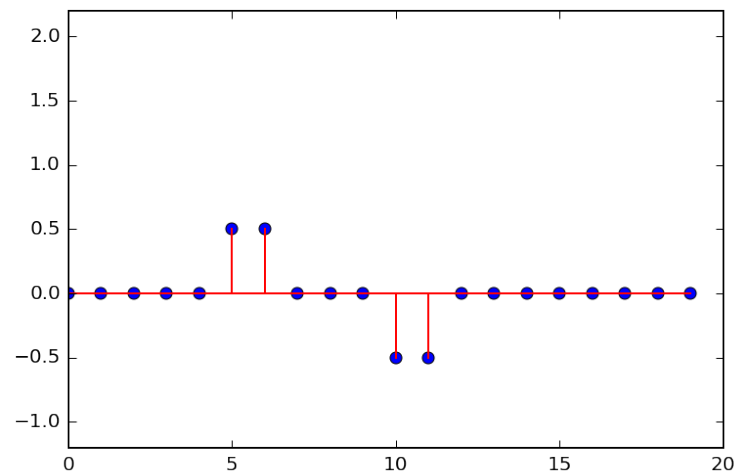


```
def op3(signal):
    transformed_signal=np.zeros(np.size(signal))
    for t in np.arange(np.size(signal)):
        transformed_signal[t]= 0.7*transformed_signal[t-1]+signal[t]
    return transformed_signal

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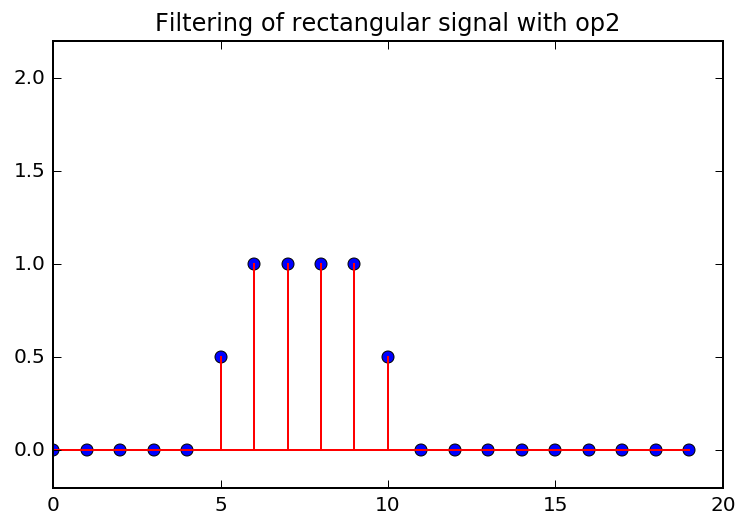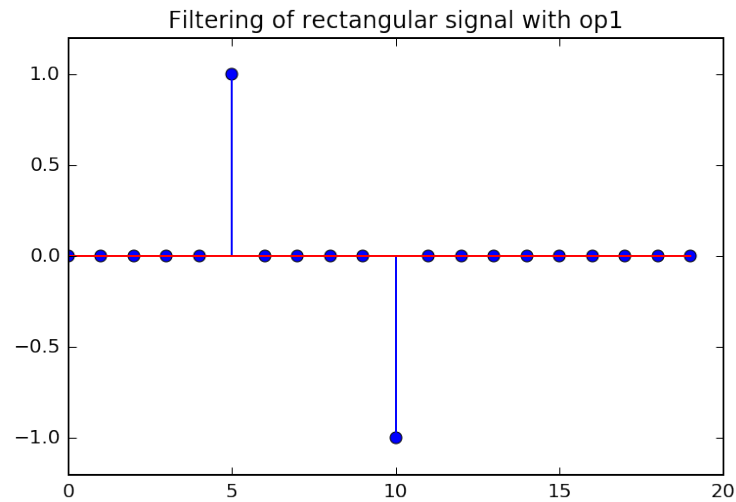
Filtering of rectangular signal with op3
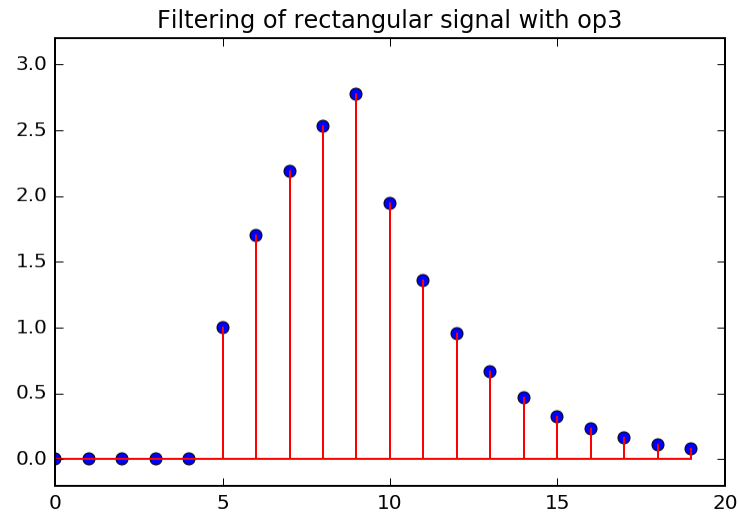
A curiosity

```
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])
```

Filtering of rectangular signal with op4

Filtering of rectangular signal with op1(op4)
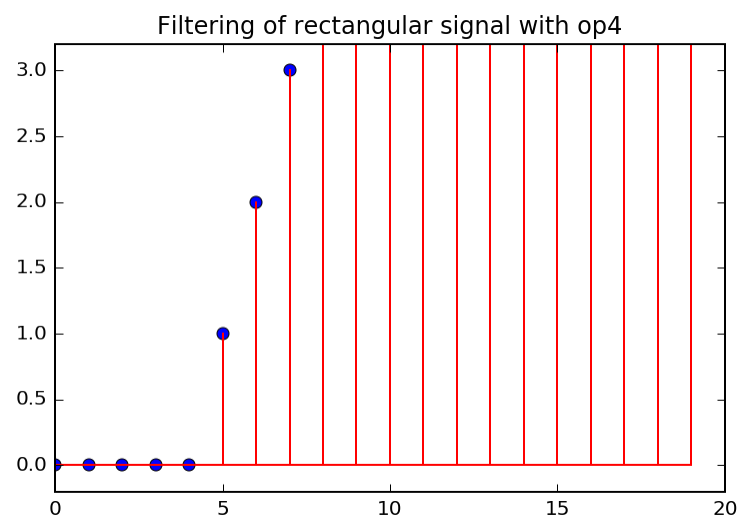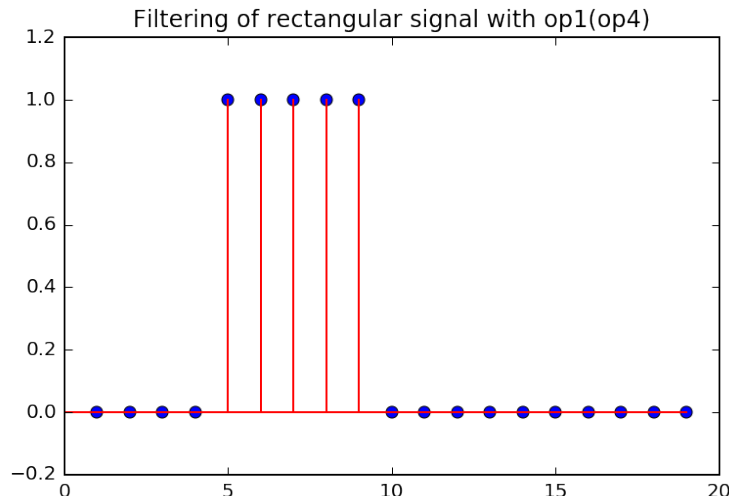
1.2.2   Filters

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")
```
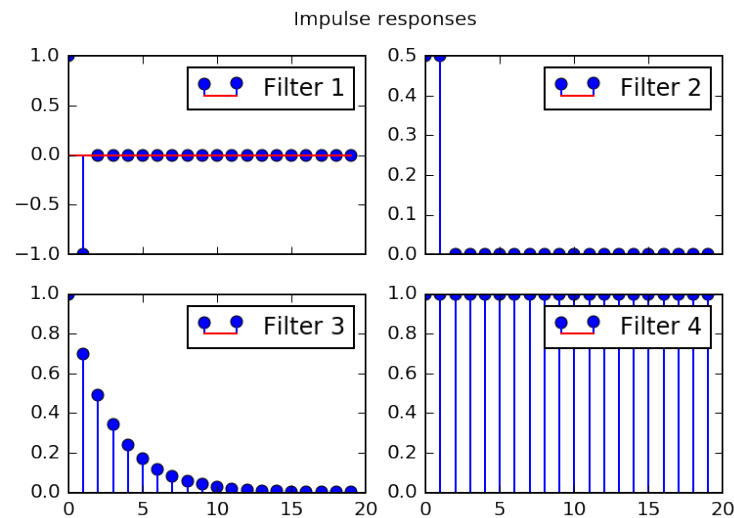

Curiosity (continued)

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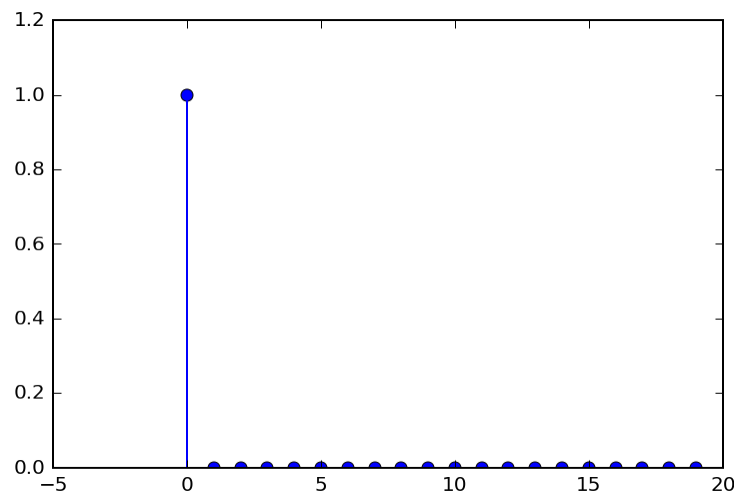
This is nothing but a Dirac impulse! We already observed that op4(op1(signal))=signal; that is the filter is an identity transformation. In other words, op4 acts as the "inverse" of op1. Finally, we note that the impuse response of the indentity filter is a Dirac impulse.

```
%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)
```

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.

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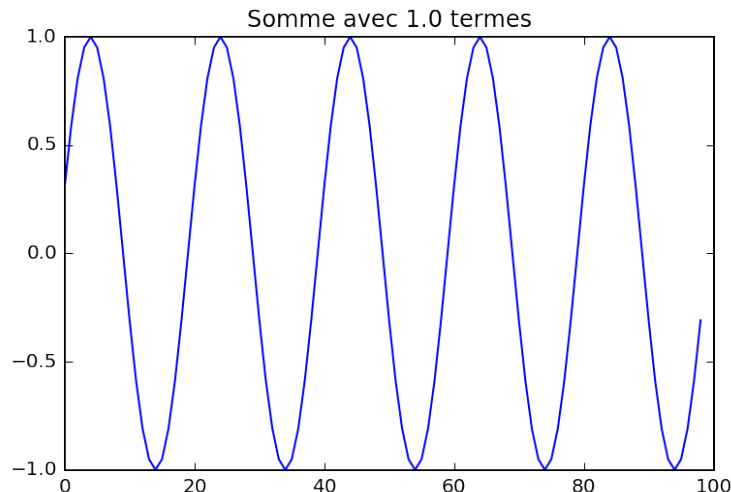
The next example is more involved in that it sums sin a cos of different frequencies and with different amplitudes. We also add widgets (sliders) which enable to interact more easily with the program.

```python
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')
```
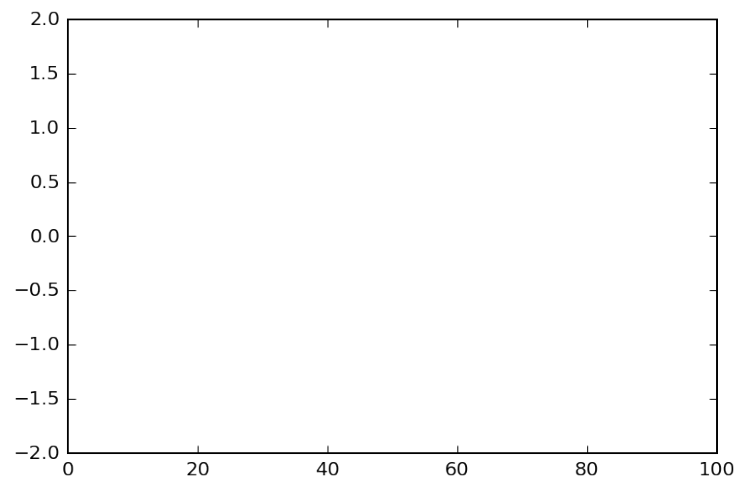

2.1.1 Decomposition on basis - scalar producs

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')
```
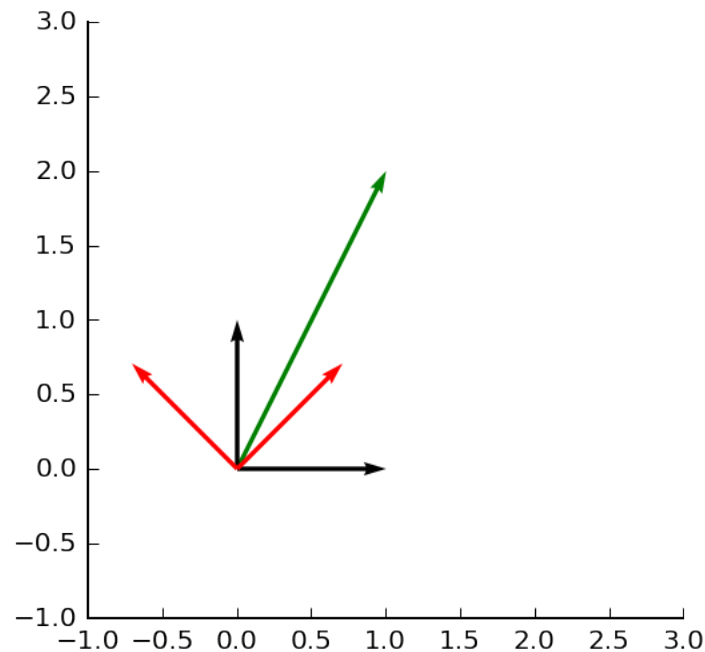

From a coordinate system to another: Take a vector (in green in the illustration). Its coordinates in the system (u, v) are [1,2]. In order to obtain the coordinates in the new system (O, u_1, v_1), we have to project the vector on u_1 and u_2. This is done by the scalar products:

```
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

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

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

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

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

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

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>]`
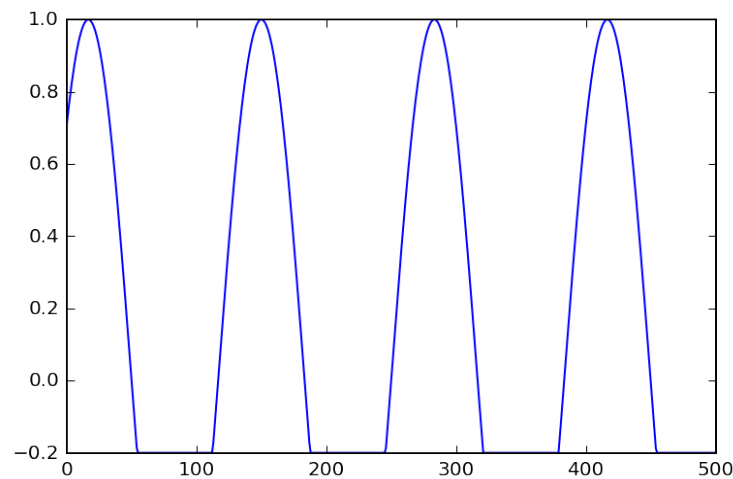
A function for computing the Fourier series coefficients

```
# 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.

reconstruction by Fourier series

Fourier series coefficients (modulus)
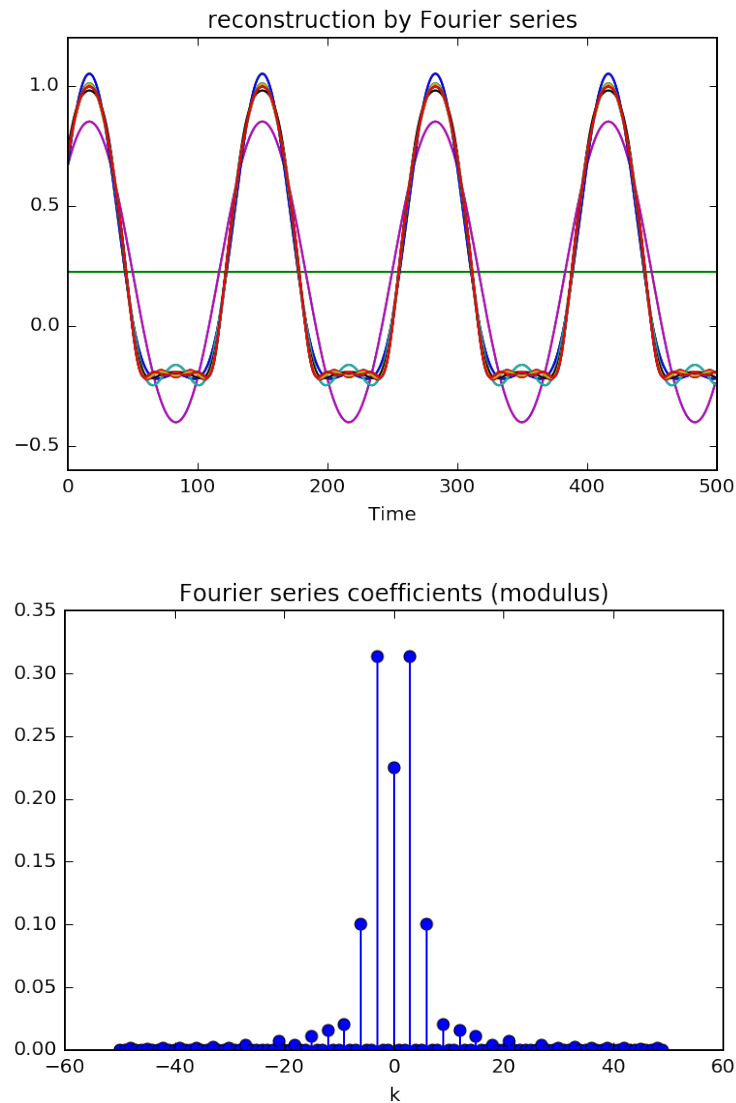
A pulse train corrupts our original signal

```python
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>]
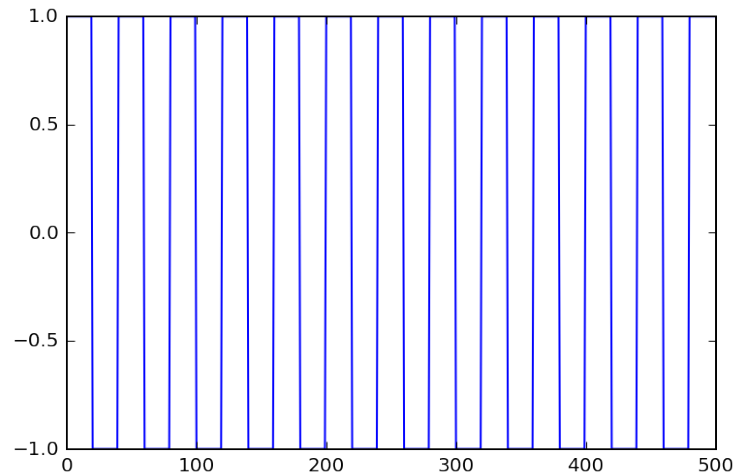
Compute and represent the Fourier coeffs of the pulse train

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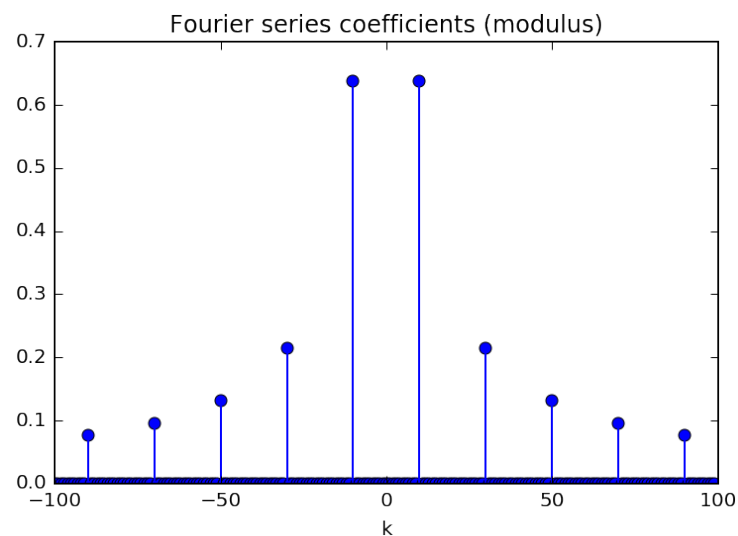
The fundamental frequency of the pulse train is 1 over the length of the pulse, that is 1/40 here. Since The Fourier series is computed on a length L=400, the harmonics appear every 10 samples (ie at indexes k multiples of 10).

```
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")
```

Corrupted signal

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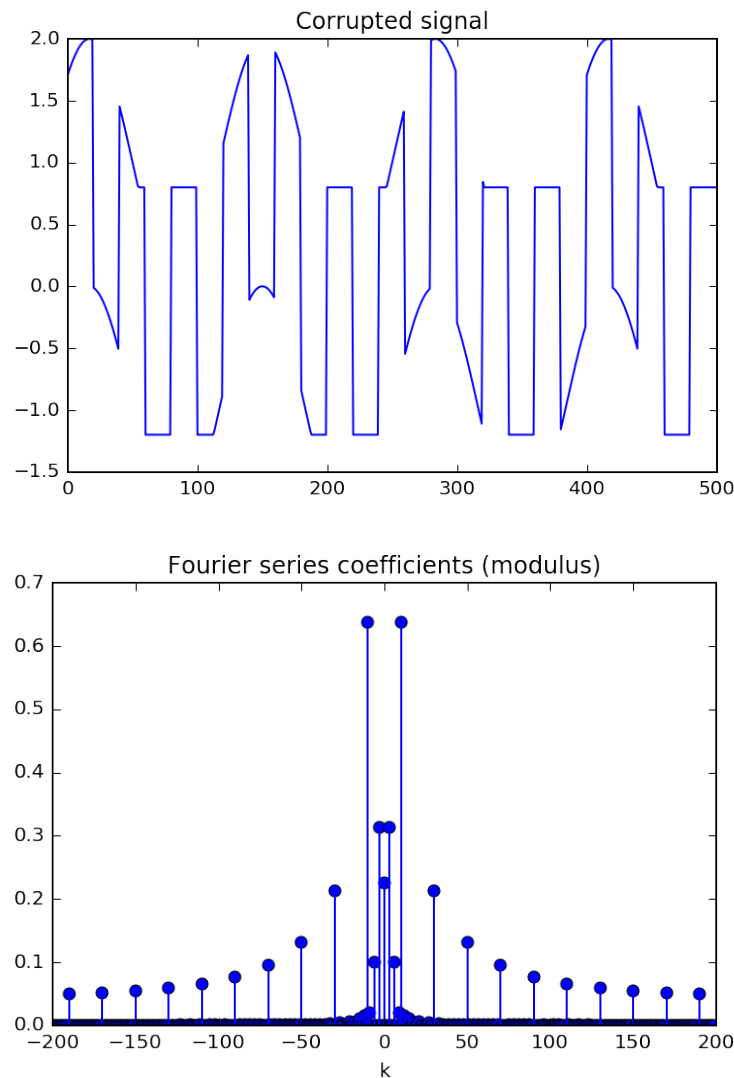
Now, we try to kill all the frequencies harmonics of 10 (the fundamental frequency of the pulse train), and reconstruct the resulting signal...

```
# kill frequencies harmonics of 10 (the fundamental frequency of the pulse
    train)
```

```python
# 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")
```

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

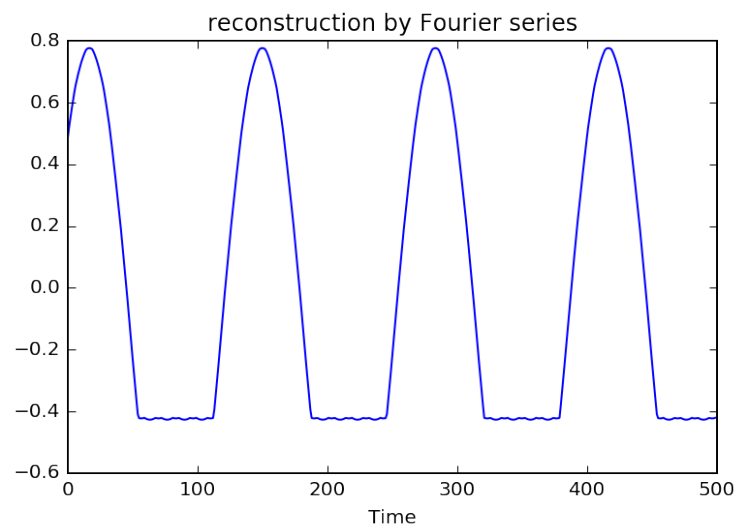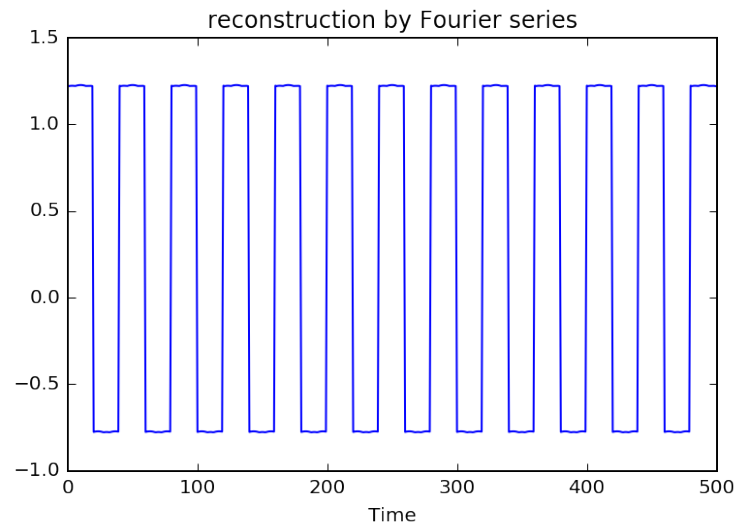Suppose that we only have an observation of length N. So no periodic signal, but a signal of size N. We do not know if there were data before the first sample, and we do not know if there were data after sample N. What to do? Facing to such situation, we can still - imagine that the data are periodic 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

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

- 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 Δf=1/period, we see that $\Delta f \to 0$. Then the Fourier representation becomes continuous. This is illustrated below.

```python
# 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)
```

```python
    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
    ])
```

```python
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")
```
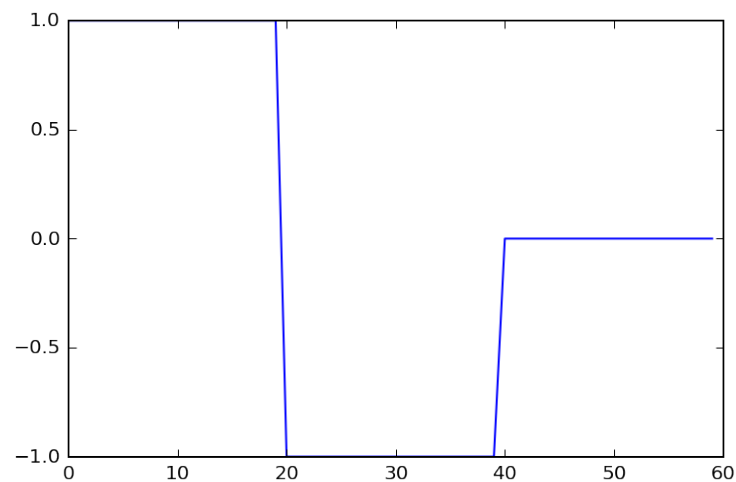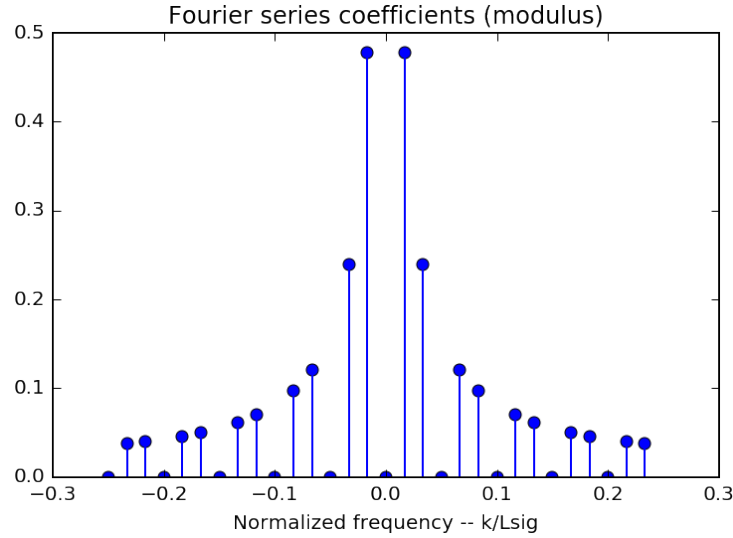
Hence we obtain a formula where the discrete sum for reconstructing the time-series $x(n)$ becomes a continuous sum, since f is now continuous:

$$x(n) = \sum_{k=0}^{N-1} c_k e^{j2\pi\frac{kn}{N}} = \sum_{k/N=0}^{1-1/N} NX(k)e^{j2\pi\frac{kn}{N}}\frac{1}{N}$$

$$\rightarrow x(n) = \int_0^1 X(f)e^{j2\pi fn}\mathrm{d}f$$

(3.2)

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

$$x(n) = \int_0^1 X(f)e^{j2\pi fn}\mathrm{d}f$$

$$\text{with } X(f) = \sum_{n=-\infty}^{\infty} x(n)e^{-j2\pi fn}$$

(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>]
```
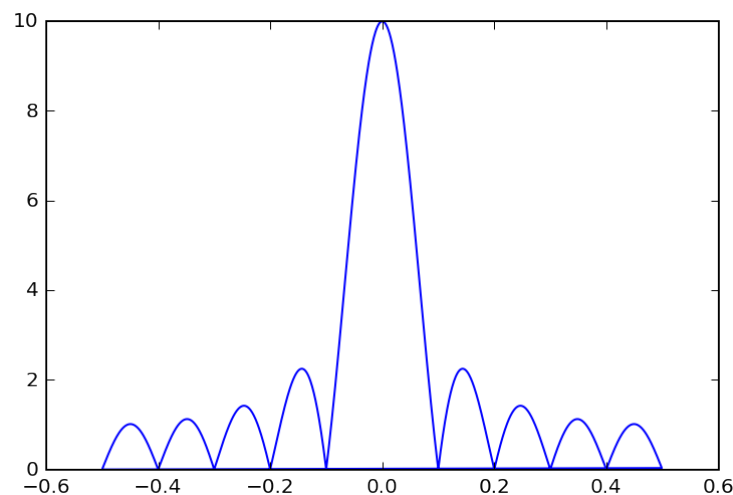

It remain to compare this to a discrete cardinal sinus. First we define a function and then compare the results.

```
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')
```
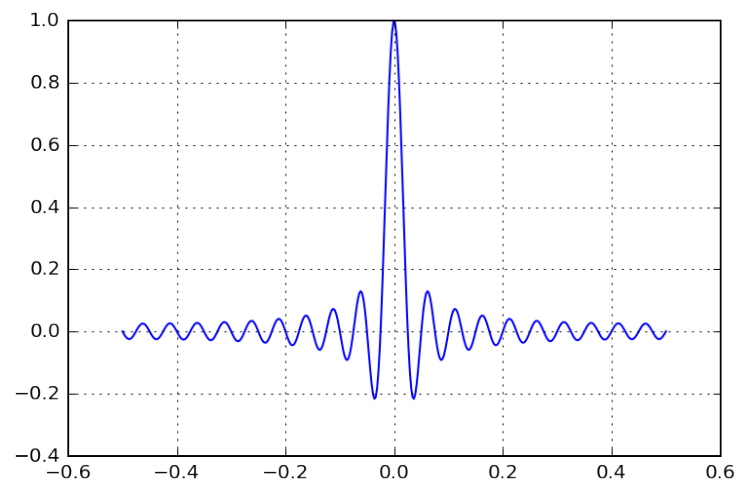
Comparison of the Fourier transform of a rectangle and a cardinal sine:

```
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...

```python
# 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')
```
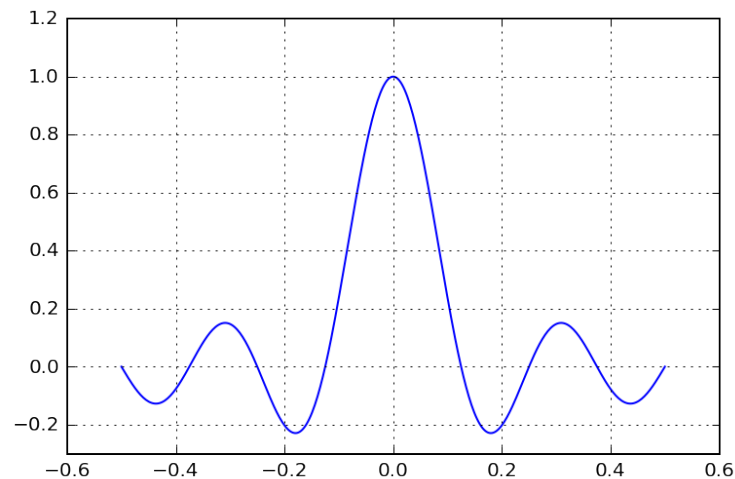

This is an example with matplotlib widgets interactivity, (instead of html widgets). The docs can be found at

```python
%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
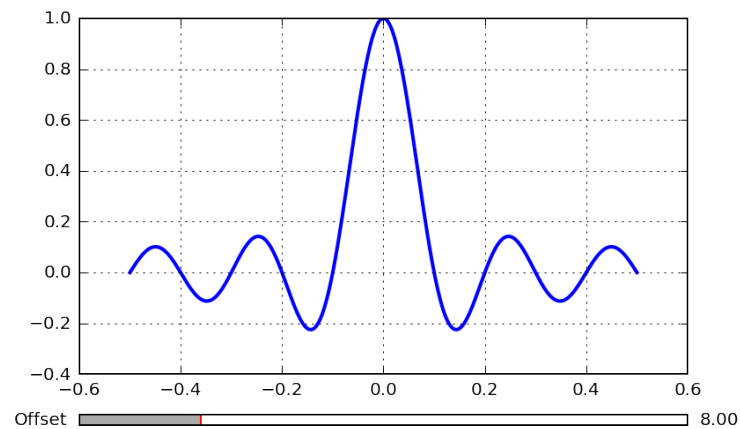
3.2.2 Fourier transform of a sine wave

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)
```
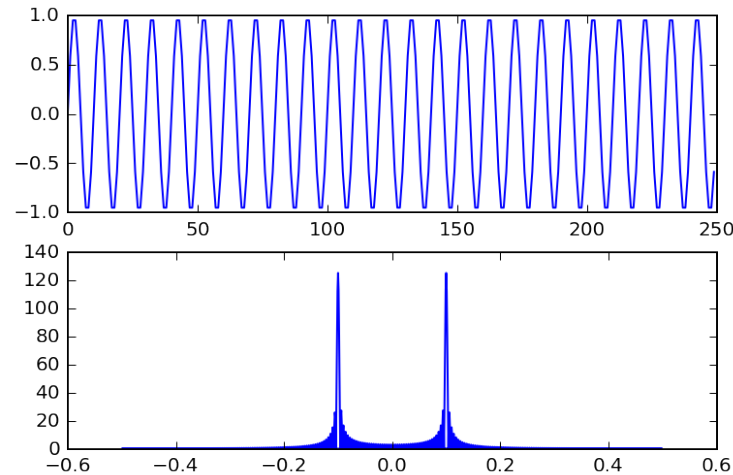
Interactive versions

```
# 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)
```
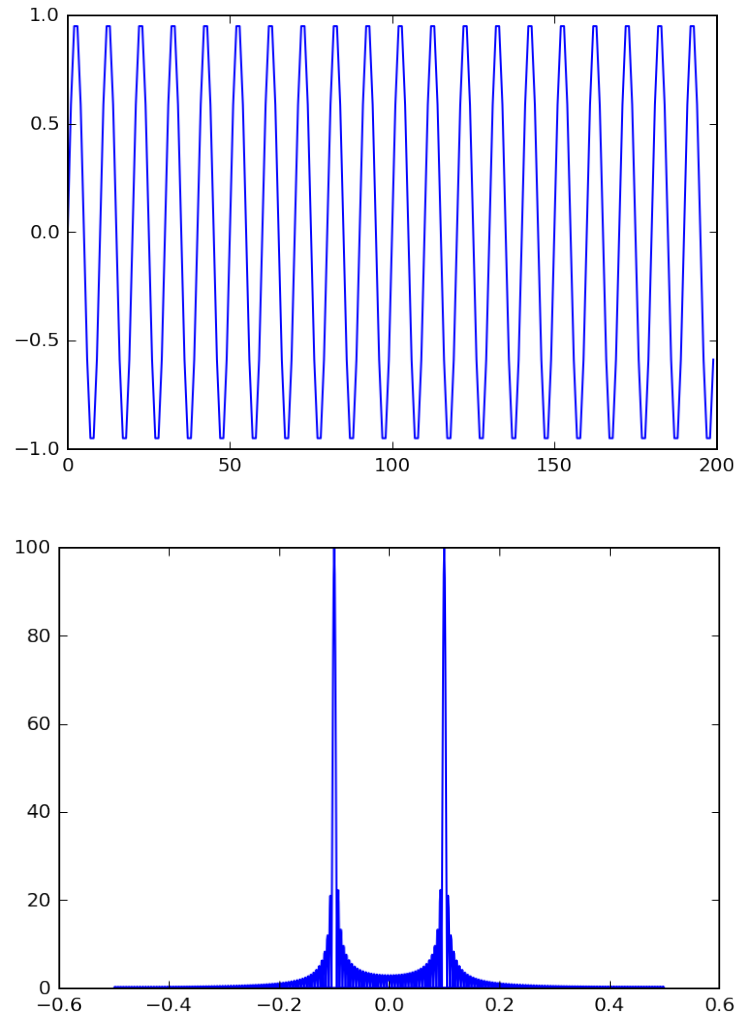
```
%matplotlib
from matplotlib.widgets import Slider

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
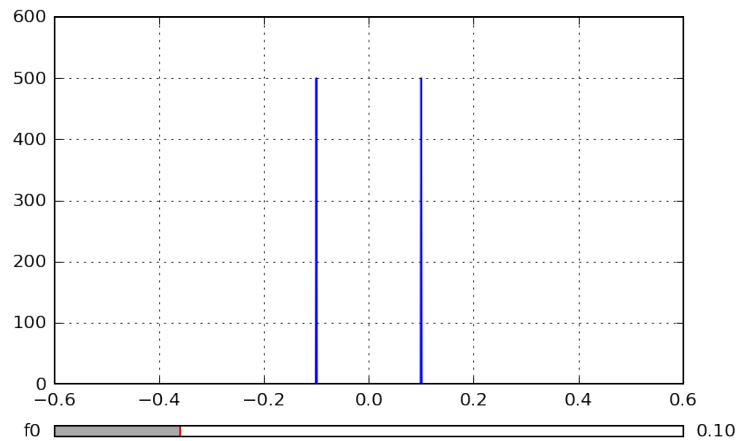
Some definitions

$$ \tag{3.4} $$

```
%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

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

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

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

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

$$\tag{3.13}$$

Finally, we have

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

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	$\sum_{n=-\infty}^{\infty} |x(n)|^2$	$\int_{[1]} |X(f)|^2 df$

$$\tag{3.15}$$

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:

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

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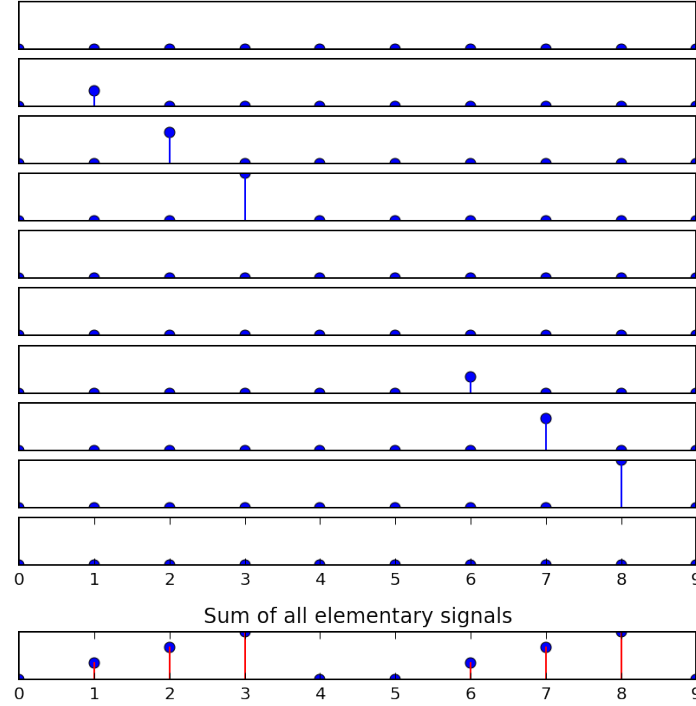
```
ax[L+1].get_yaxis().set_ticks([])
ax[L+1].stem(x,'r')
ax[L+1].set_title("Sum of all elementary signals")
#f.tight_layout()
f.suptitle("Decomposition of a signal into a sum of Dirac",fontsize=14)
```

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

Decomposition of a signal into a sum of Dirac

Sum of all elementary signals

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

```python
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])
```
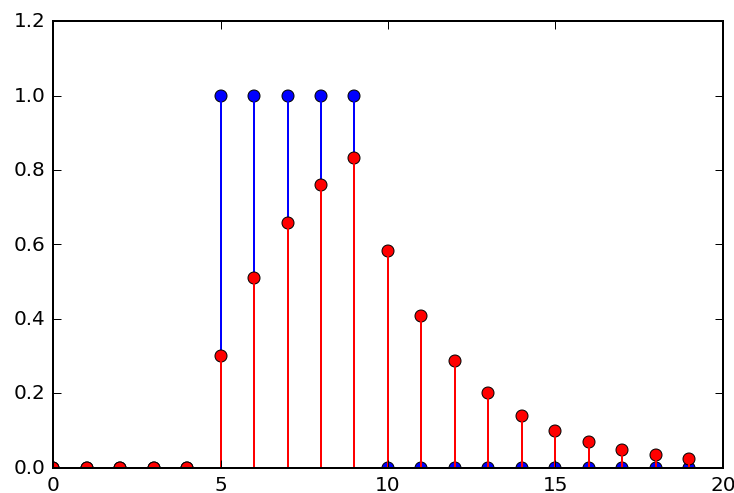

Response by the sum of delayed impulse responses

```python
s=np.zeros((N,N))
for k in range(N):
```

```python
            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)
```

Convolution as the sum of all delayed impulse responses

Sum of all responses to elementary signals
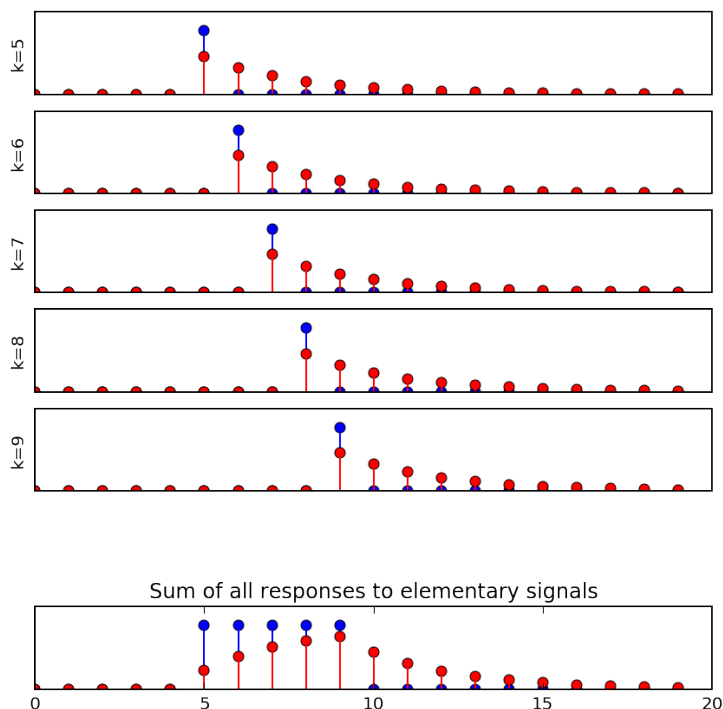
4.2.3 Exercises

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*

```python
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.]
```

```python
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:

```python
#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
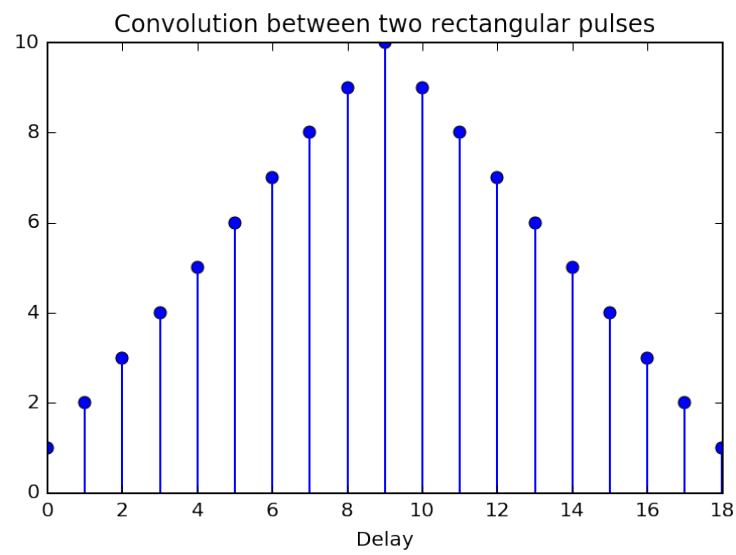
Transfer function

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*.

49

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:

$$\mathrm{FT}[x(n)y(n)] = \sum_n x(n)y(n)e^{-j2\pi fn} \tag{5.9}$$

$$= \sum_n \int X(f_1)e^{j\pi f_1 n}\mathrm{d}f_1 \int Y(f_2)e^{j\pi f_2 n}\mathrm{d}f_2 e^{-j2\pi fn} \tag{5.10}$$

$$= \iint X(f_1)Y(f_2)\sum_n e^{j\pi f_1 n}e^{j\pi f_2 n}e^{-j2\pi fn}\mathrm{d}f_1\mathrm{d}f_2 \tag{5.11}$$

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:

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

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**.

$$\tag{5.19}$$

$$\tag{5.20}$$

$$\tag{5.21}$$

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 impulse 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.
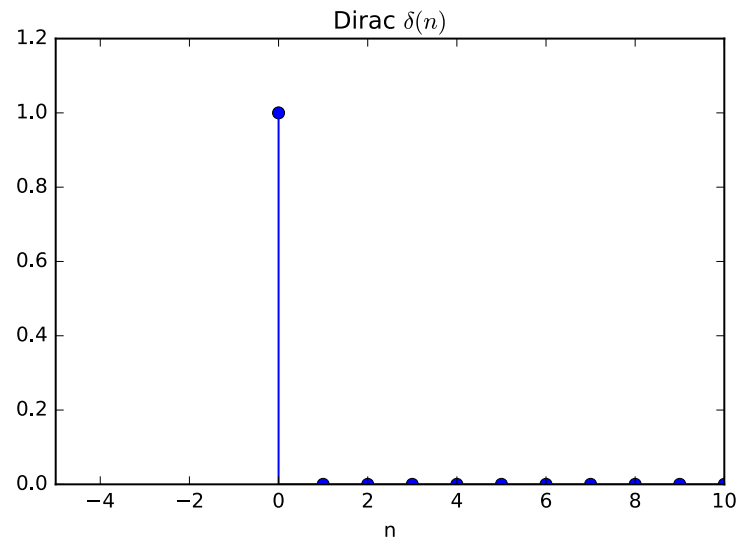
```
from pylab import *
```

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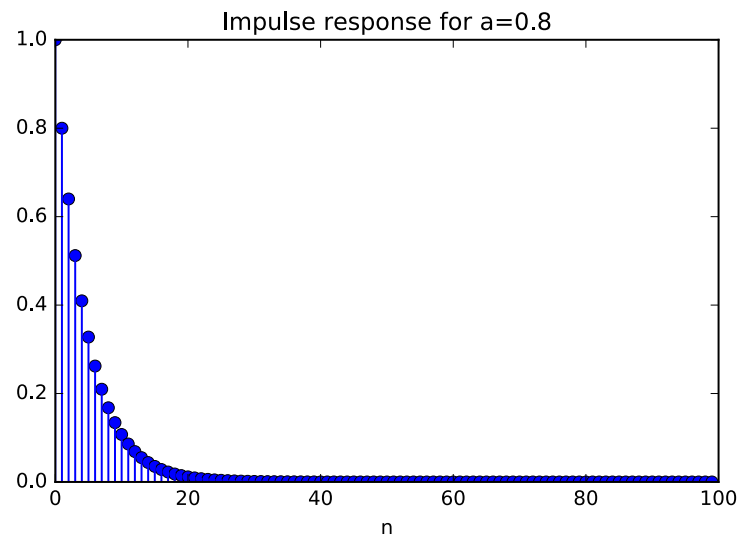
```
                          -1                    -nb
              b[0] + b[1]z   + ... + b[nb] z
     Y(z) = ----------------------------------- X(z)
                          -1                    -na
              a[0] + a[1]z   + ... + a[na] z
```

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!**

```python
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:

```python
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")
```

Impulse Response for a=0.99

Impulse Response for a=1.01
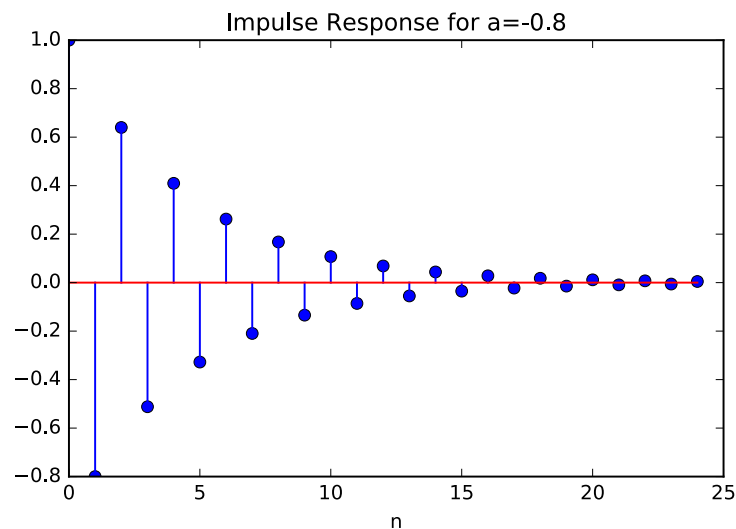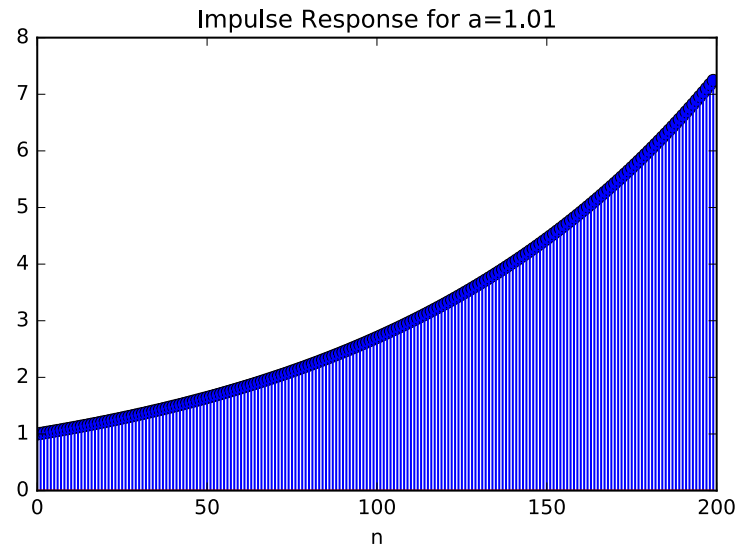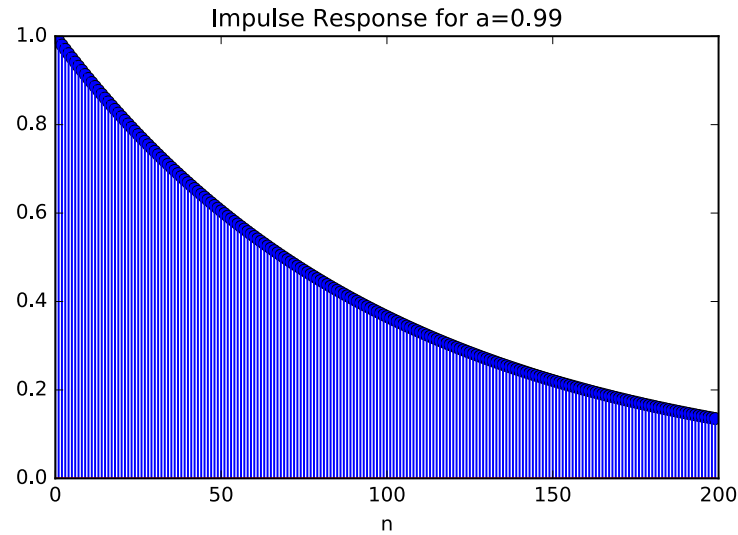
Conclusions:

- 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.

```python
# 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
```
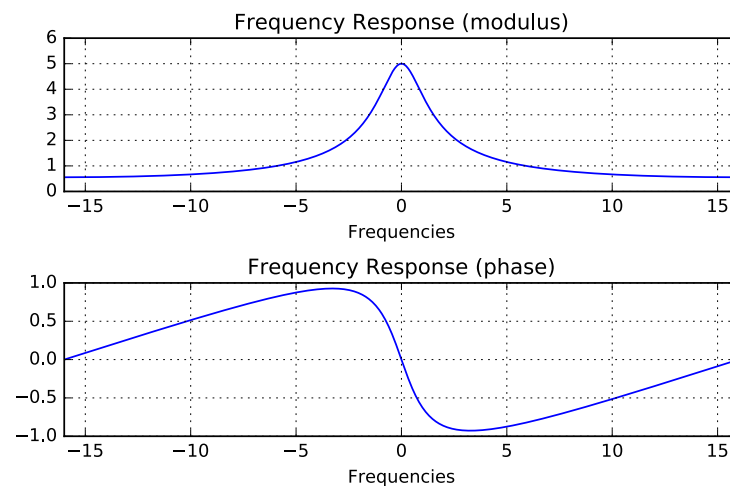


```python
# Value at f=x: we look for it by find(f==x)
print ("Value at f=0 : ".rjust(20),H[find(f==0)].real)
print ("Value at f=Fe/2 : ",H[find(f==-Fe/2)].real)
print ("To compare with theoretical values")
```

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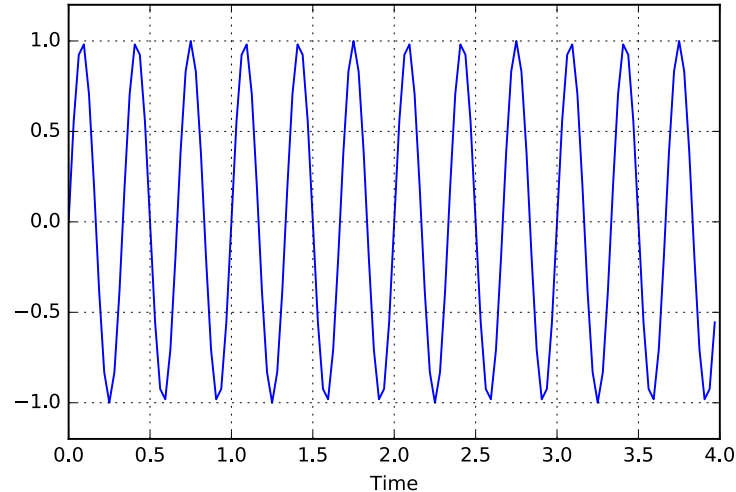
8.4   Filtering

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.
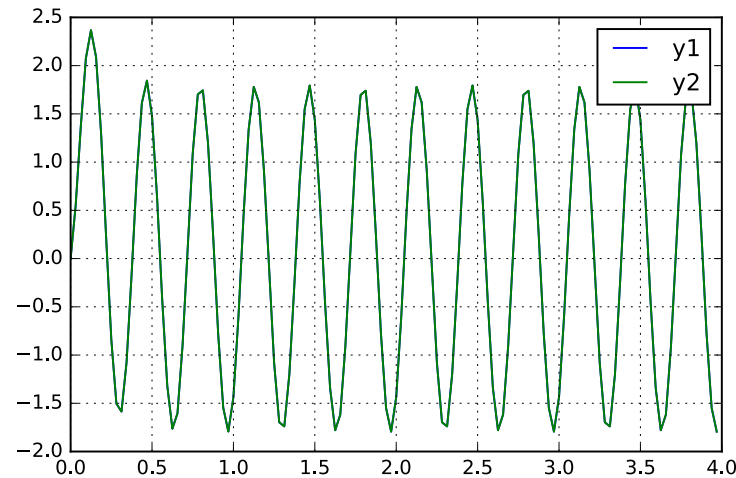
8.4.1   Analysis in the time domain

```
# 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])
```
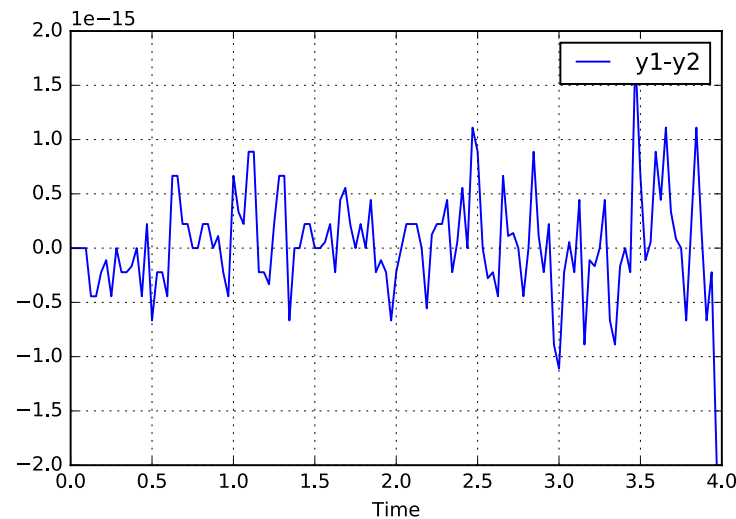
(-1.2, 1.2)


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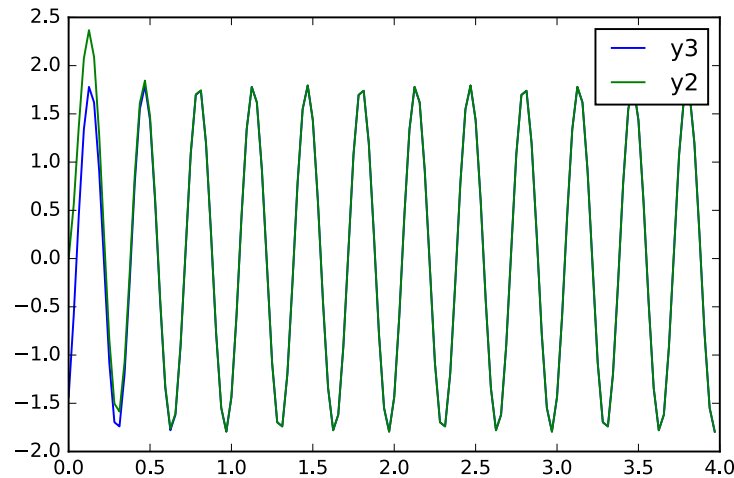
The difference observed at the beginning of the two plots comes from a different assumption on the values of the signals at negative (non observed) times. Actually, function`lfilter` assumes that the signal is zero where non observed, which implies a transient response at the output of the filter. The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

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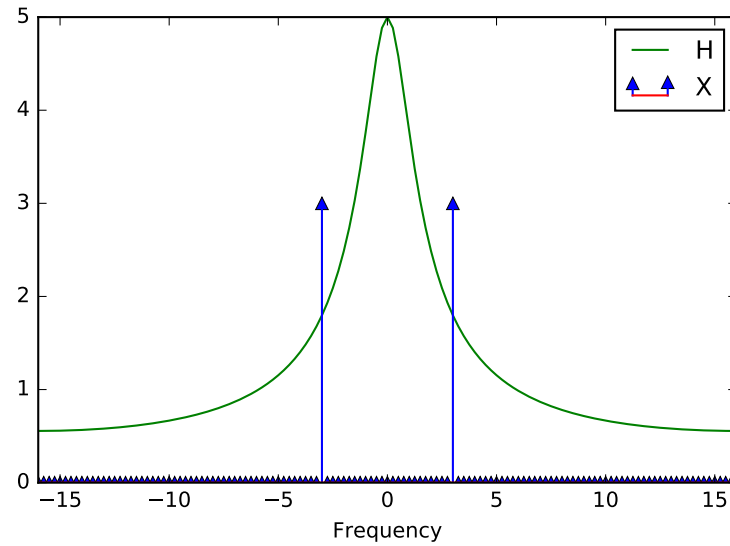
The sine wave has frequency $fo = 3$. let us measure the values of gain and phase at this frequency:

```
H3=H[find(f==3)]
print("Value of the complex gain:", H3)
print("Modulus :", abs(H3))
print("Phase (degrees):", angle(H3)/pi*180)
```
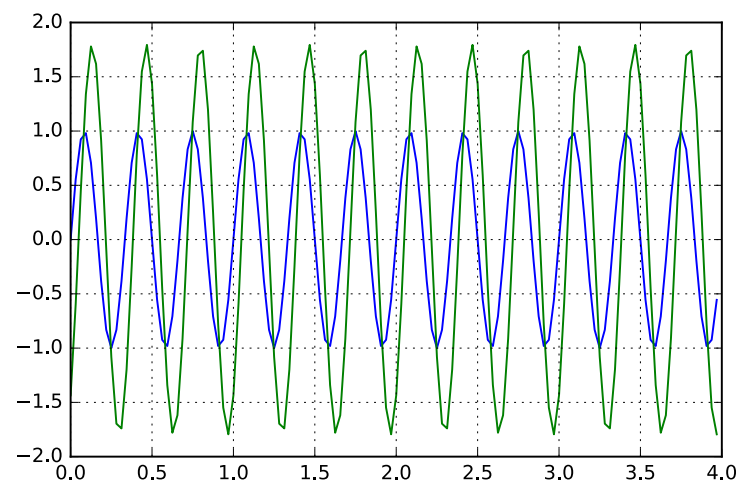
```
Value of the complex gain: [ 1.08130406-1.43535659j]
Modulus : [ 1.79707178]
Phase (degrees): [-53.00801337]
```

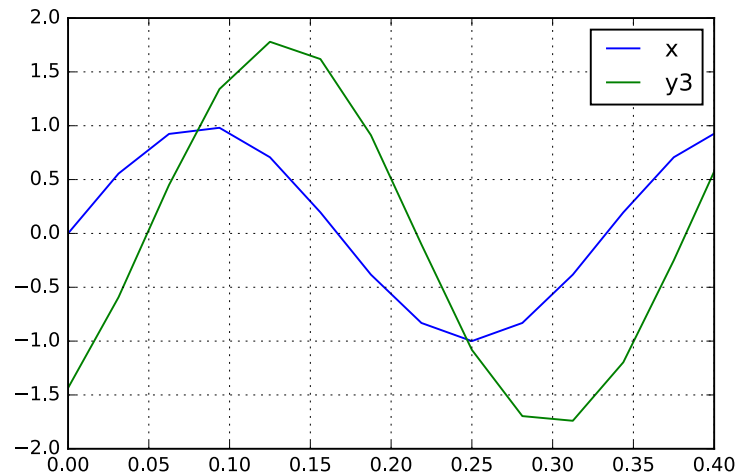Now, let us look at this on the time representation.

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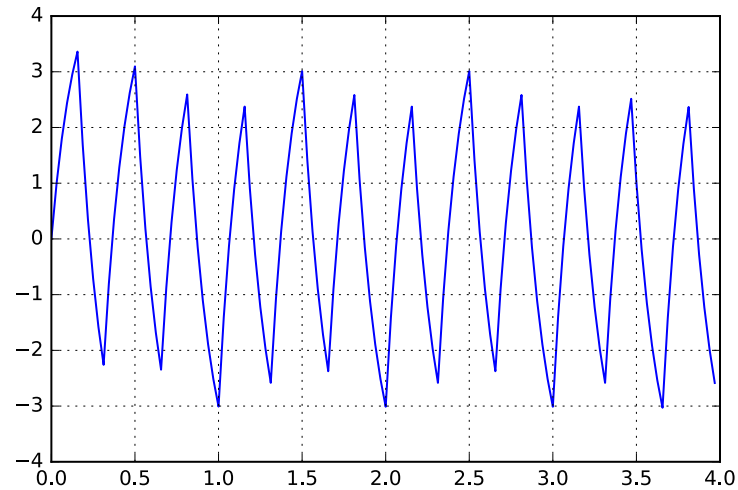
Finally, and for the *fun*, we examin what happens with a larger rectangular pulse. We see in particular the rise and fall time to attain a stable level, like in electrical circuits.

```python
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')
```