# ECED 4502 - Digital Signal Processing

## Fundamental of Fourier transform, FFT and its practical aspect

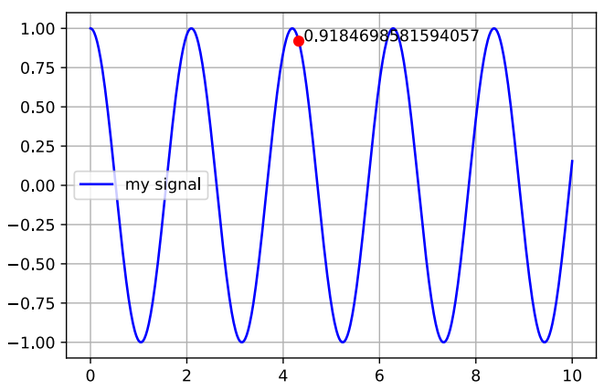
In [1]:
```
# Author : Jay Patel

# loading libraries and utilities
from __future__ import print_function, division # this insures python 2 / python 3 compatibility
# numpy provides a fast computation of large numerical arrays
import numpy as np # (here we just give numpy the (standard) nick name np for easing the source code)
# matplotlib is the graphic library
# matplotlib.pylab is an easy to use utility, a bit reminiscent to matlab graphics
import matplotlib as mpl
import matplotlib.pylab as plt
# matplotlib is a "magic" command to insert directly the graphics in the web page
%matplotlib inline
# %pylab inline
# would be a short cut for the lines above

#populate SVG images for better clarity
from IPython.display import Image, SVG
```

In [2]:
```
plt.rcParams['savefig.dpi'] = 100 # default 72
%config InlineBackend.figure_formats=['svg'] # SVG inline viewing
```

In [3]:
```
# let's use this to generate a pseudo-signal
x = np.linspace(0,10,1000) # a vector of 1000 points equi distant from 0.0 to 10.0
freq = 3.0
y = np.cos(freq*x) # takes the cos() values of all points in x - this will be the signal
print('length of vectors, x:',len(x), 'y:',len(y))
print('433th value:', x[432], y[432]) # !!! array indices are from 0 to 999 !!
```

```
length of vectors, x: 1000 y: 1000
433th value: 4.324324324324325 0.9184698581594057
```

In [4]:
```
# and plot it - like in Excel !
plt.plot(x,y,'b', label='my signal')
# add a point for the 433th element
plt.plot(x[432],y[432], 'ro') # r is for red o is for round points
plt.text(x[432]+0.1, y[432], str(y[432]))
plt.legend();plt.grid()
```




**basic FT**

In [5]:
```
# using FFT is as simple as
from numpy import fft # as fft is just giving it a short nick-name
# this is just about what you have to know !
```

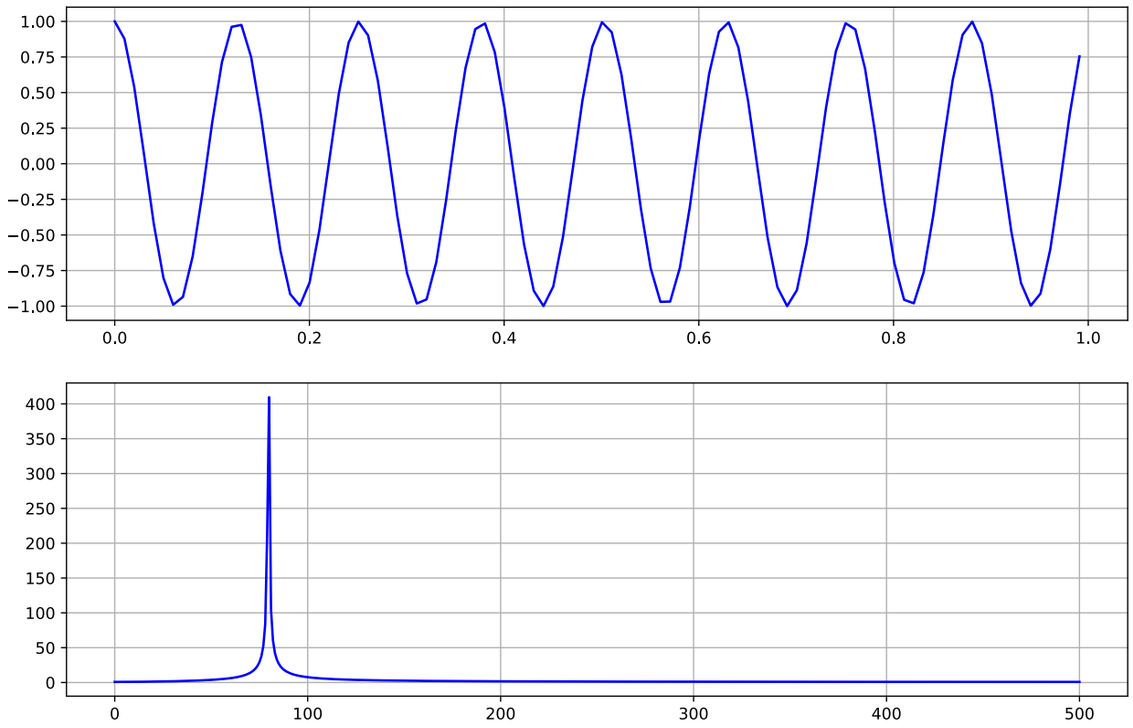
Fourier transform (or **FT**) is defined as a transformation of continuous functions $f$ from $R$ to $C$ ), they have to be integrable over $]-\infty \ldots \infty[$, and can be extended to the limit to distributions which somehow drops this later condition.

What we're doing here is very different, it is another transform, called digital Fourier transform (or **DFT**), perfectly defined in mathematical terms, but very different in its form, that applies to finite series of values $y_k$. DFT applies in the computer, were we are going to compute of vectors of values $x[k]$ as a representation of the series $x[k]$. In the computer, DFT transforms thus a vector into another vector, as it is a linear operation, it can be represented by a (usually square) matrix, and would take a burden proportional to $N^2$ to compute for a vector of length.

there is an very efficient algorithm that does it in $Nlog_2(N)$ operations provided N is a power of two ($N = 2^k$), and which is called Fast Fourier transform (or **FFT**). FFT and DFT are strictly equivalent, as there are now effecient implementations that work well for nearly all values.

In [6]:
```
freq = 50.0
y = np.cos(freq*x)
YY = fft.rfft(y) # rfft() is the FT of a real vector (here y) - the result is complex
print('YY type:',YY.dtype) # to check type
f, (ax1,ax2) = plt.subplots(nrows=2,figsize=(12,8)) # a multiple plot, with 2 "rows"
ax1.plot(x[0:100], y[0:100],'b') # y[0:100] is a way of telling - only the first 100 points,
ax2.plot(abs(YY),'b'); # abs(YY) is the module of the complex YY
ax1.grid();ax2.grid()
```

```
YY type: complex128
```




Here we see the **two reciprocal domains**

- direct (time for instance) domain aboveem
- indirect (frequency for instance) domain below

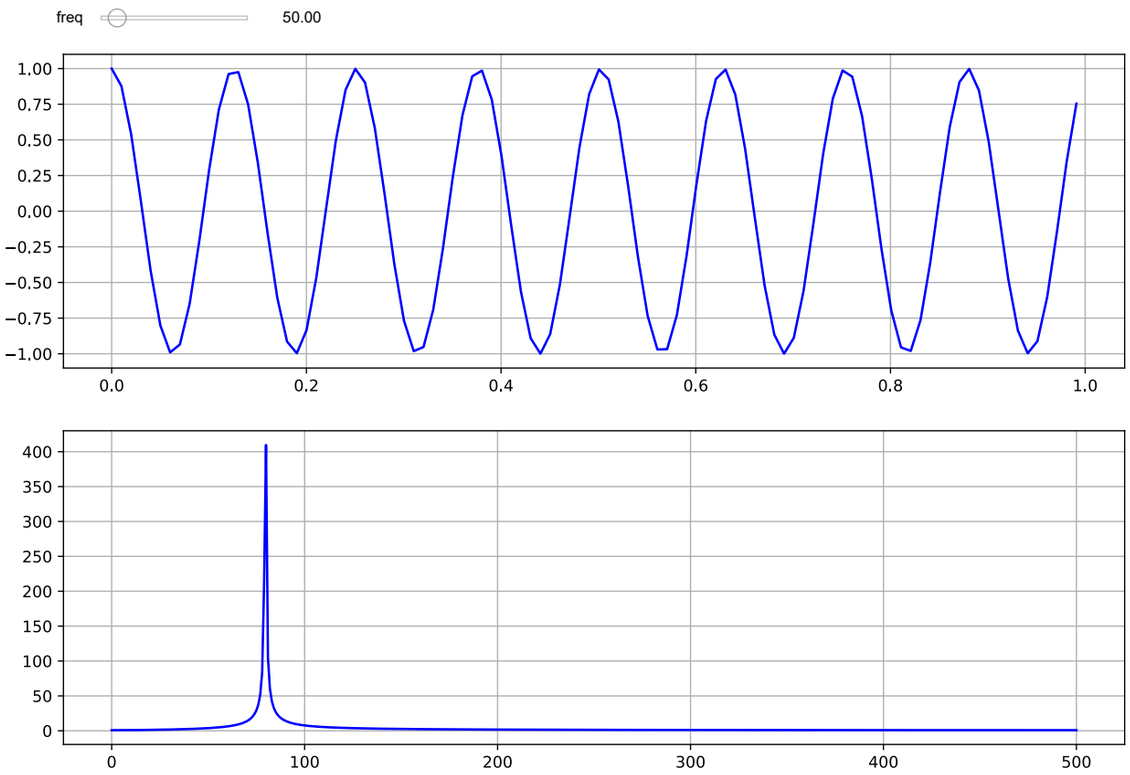
we do not have to give a frequency axis, because it is implicitely defined, not also that the axis used for the plot is arbitrary, only counting points (yes, 500 = half of the points from 'y' , we'll come to this later)

**frequency limits - Aliasing - Nyquist frequency**

Let's use an interactive version of the above to find the frequency limits.

In [7]:
```
# load the interactive tool
from ipywidgets import interact, interactive, widgets, fixed
try:
  from ipywidgets import Layout
except:
  pass # we'll do without it
```

In [8]:
```
# we define a function fta() which does the same as the lines above
def fta(freq = 50.0):
  "showing aliasing effect - and Nyquist frequency"
  y = np.cos(freq*x)
  YY = fft.rfft(y)
  f, (ax1,ax2) = plt.subplots(nrows=2,figsize=(12,8))
  ax1.plot(x[0:100], y[0:100],'b')
  ax1.set_ylim(ymin=-1.1, ymax=1.1)
  ax2.plot(abs(YY),'b');ax1.grid();ax2.grid()
  # then use it interactively,
interactive( fta, freq=(0.0,500.0))
# interactive_plot
```




The same as above, with the frequency of the time signal defined by a cursor by playing with the $freq$ cursor, try to

- see what happens for low / high frequencies
- detect a strange behavior (frequency inversion ?) for high frequencies
- determine as precisely as possible the frequency at which the behavior changes
- observe the stroboscopic effects around this peculiar frequency
- find what is the special content of the $y$ vector at this frequency

1. As you can observe, at high frequency, the line in the Fourier spectrum folds back to lower frequency, and is thus located at a wrong position in the spectrum. This effect is called **Aliasing**
2. the frequency at which the folding occurs when the sampled signal $y$ oscillates up and down for exactly each sampling point. It means that the frequency of the signal is exactly half of the sampling frequency (there are exactly 2 measures per signal period).

This special frequency, called the **Nyquist frequency**, corresponds to the highest frequency which can be measured in this regularly sampled signal; it is half of the sampling frequency. Using $SW$ as the spectral width, $SW$ as the sampling frequency, and $\Delta t$ the sampling period, this is commonly noted :

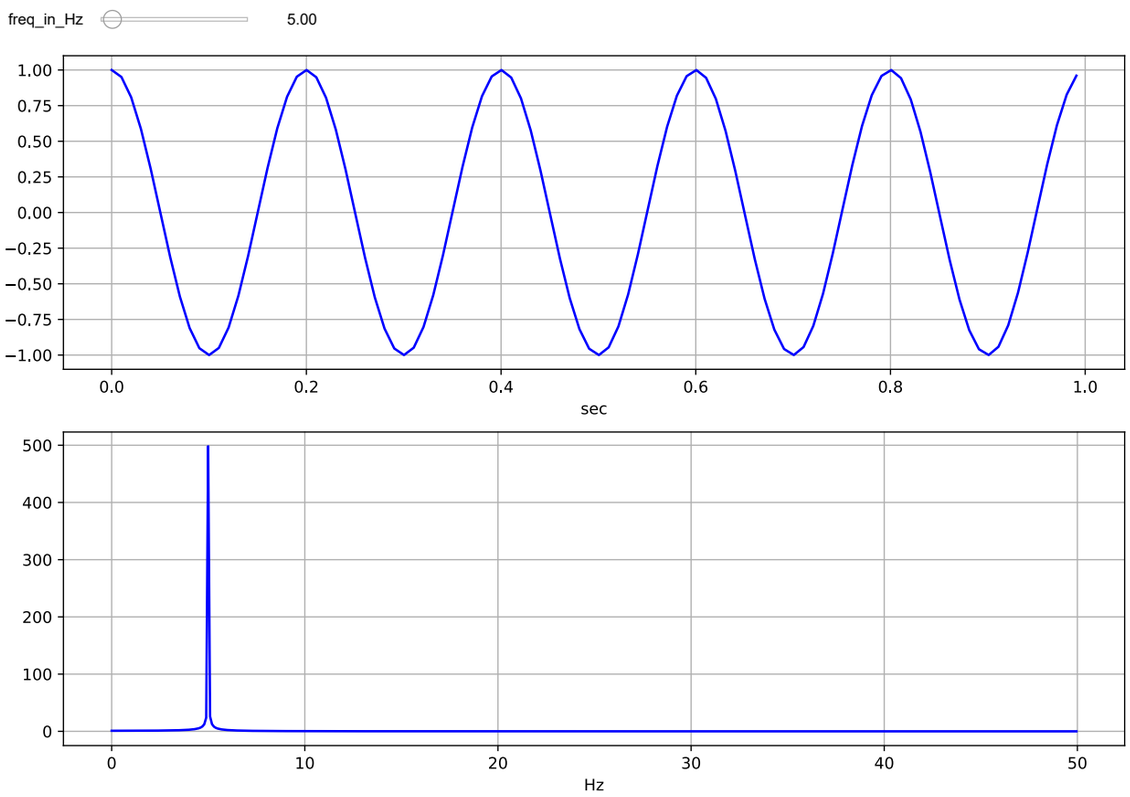
$$SW = \frac{1}{2} SF \tag{1}$$

$$SW = \frac{1}{2\Delta t} \tag{2}$$

sometimes called the Nyquist-Shannon theorem (or just Shannon theorem).

So, the $rfft()$ function creates a spectrum from 0 to $SW$. Here, the $x$ vector (the sampling) contains 1000 values over 10 sec., so we're sampling at $\frac{1}{100} sec$, hence $SW = 50Hz$. We find the folding for a cursor arond 314.0 which corresponds exactly to the expected value $2\pi \times 50Hz$. (Here $cos()$ expects values in radian not in Hz, thus the $2\pi$).

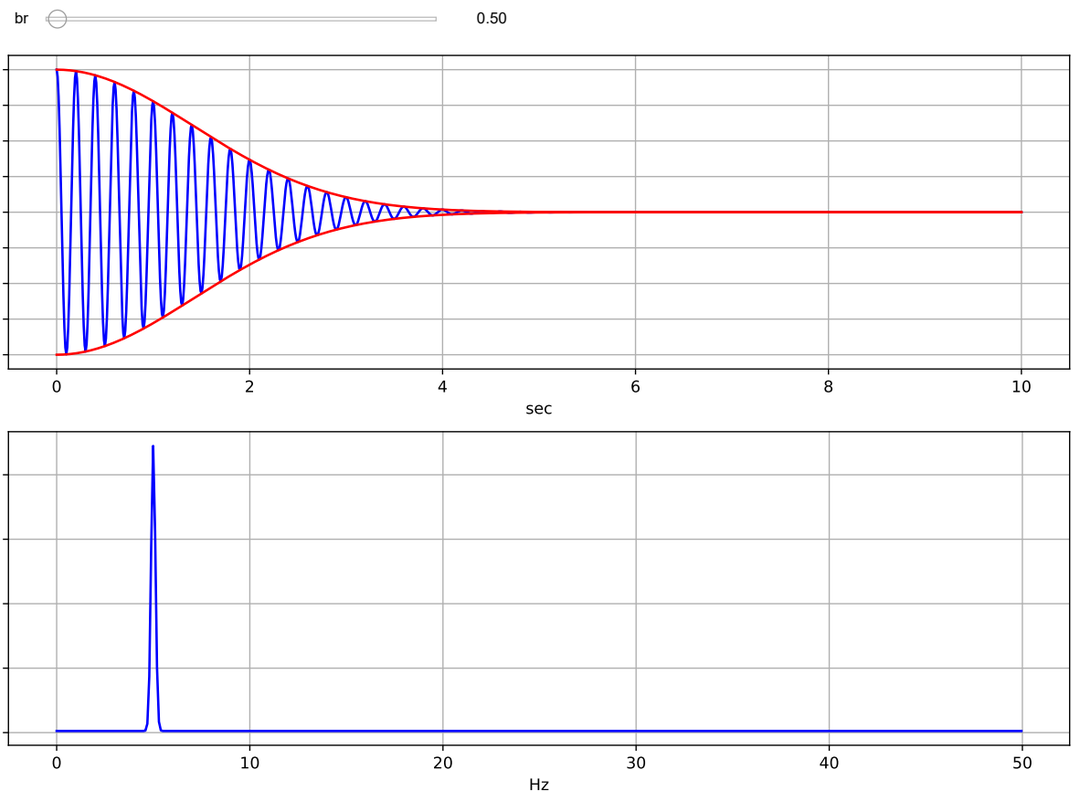
In [9]:
```
# We can now redraw the same picture, with correct labels:
def fta2(freq_in_Hz = 5.0):
  "showing aliasing effect - and Nyquist frequency"
  y = np.cos(2*np.pi*freq_in_Hz*x) # this time in Hz
  YY = fft.rfft(y)
  deltat = x[1] # as x[0] is 0, x[1] = \Delta t
  faxis = np.linspace(0,1/(2*deltat), len(YY))
  f, (ax1,ax2) = plt.subplots(nrows=2,figsize=(12,8))
  ax1.plot(x[0:100], y[0:100],'b')
  ax1.set_xlabel('sec')
  ax2.plot(faxis, abs(YY),'b')
  ax2.set_xlabel('Hz');ax1.grid();ax2.grid()
# then use it interactively,
interactive( fta2, freq_in_Hz=(0.0,70.0))
```




**duration / width - compaction properties - Gabor Limit - uncertainty theorem**

let's try to modify the signal to see what happens

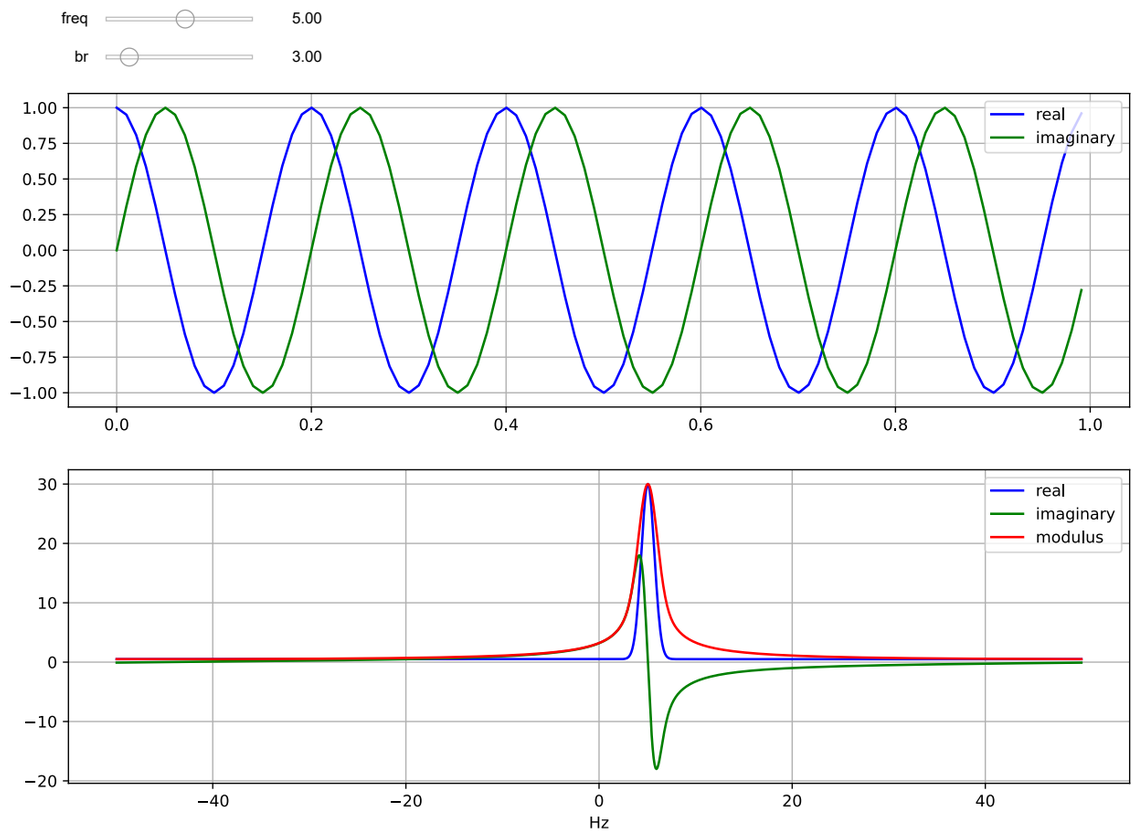
In [10]:
```
def ftb(br = 1.0):
  "function showing the effect of broadening"
  freq = 5.0 # a fixed frequency
  y = np.cos(2*np.pi*freq*x)
  gauss = np.exp(-(br*x)**2) # this is the decay with a gaussian shape
  yg = y*gauss
  YY = fft.rfft(yg)
  f, (ax1,ax2) = plt.subplots(nrows=2,figsize=(12,8))
  ax1.plot(x, yg,'b')
  ax1.plot(x, gauss, 'r') # draw the enveloppe
  ax1.plot(x, -gauss, 'r') # draw the enveloppe
  ax1.set_xlabel('sec')
  deltat = x[1] # as x[0] is 0, x[1] = \Delta t
  faxis = np.linspace(0,1/(2*deltat), len(YY))
  ax2.plot(faxis, YY.real,'b')
  ax2.set_xlabel('Hz');ax1.grid();ax2.grid()
# we used a detailed widget here, to have a better control
try:
  w=interactive( ftb, br=widgets.FloatSlider(min=0,max=20,value=.5,step=0.01,layout=Layout(width='50%')))
except:
  w=interactive( ftb, br=(0.0, 100.0, 0.5))
w
```




**Complex signal - phase properties**

In [12]:

```
def ftc(freq = 5.0, br = 3.0):
  "function showing the complex part of the signal and of the spectrum"
  y = np.cos(2*np.pi*freq*x) + 1j * np.sin(2*np.pi*freq*x) # a complex signal
  gauss = np.exp(-(br*x)**2)
  yg = y*gauss
  YY = fft.fftshift(fft.fft(yg)) # fftshift() ensures the 0 freq in the center
  f, (ax1,ax2) = plt.subplots(nrows=2,figsize=(12,8))
  ax1.plot(x[0:100], y[0:100].real, 'b', label='real') # .real is for real part; 'b' is for blue
  ax1.plot(x[0:100], y[0:100].imag, 'g', label='imaginary')
  ax1.legend(loc=1)
  ax1.set_ylim(ymin=-1.1,ymax=1.1)
  deltat = x[1] # as x[0] is 0, x[1] = \Delta t
  faxis = np.linspace(-1/(2*deltat),1/(2*deltat), len(YY))
  ax2.plot(faxis, YY.real, 'b', label='real')
  ax2.plot(faxis, YY.imag, 'g', label='imaginary')
  ax2.plot(faxis, abs(YY), 'r', label='modulus')
  ax2.set_xlabel('Hz')
  ax2.legend(loc=1);ax1.grid();ax2.grid()
interactive_plot = interactive( ftc, freq=(-70.0,70.), br=(0.0,20.0))
# output = interactive_plot.children[-1]
# output.layout.height = '550px'
interactive_plot
```



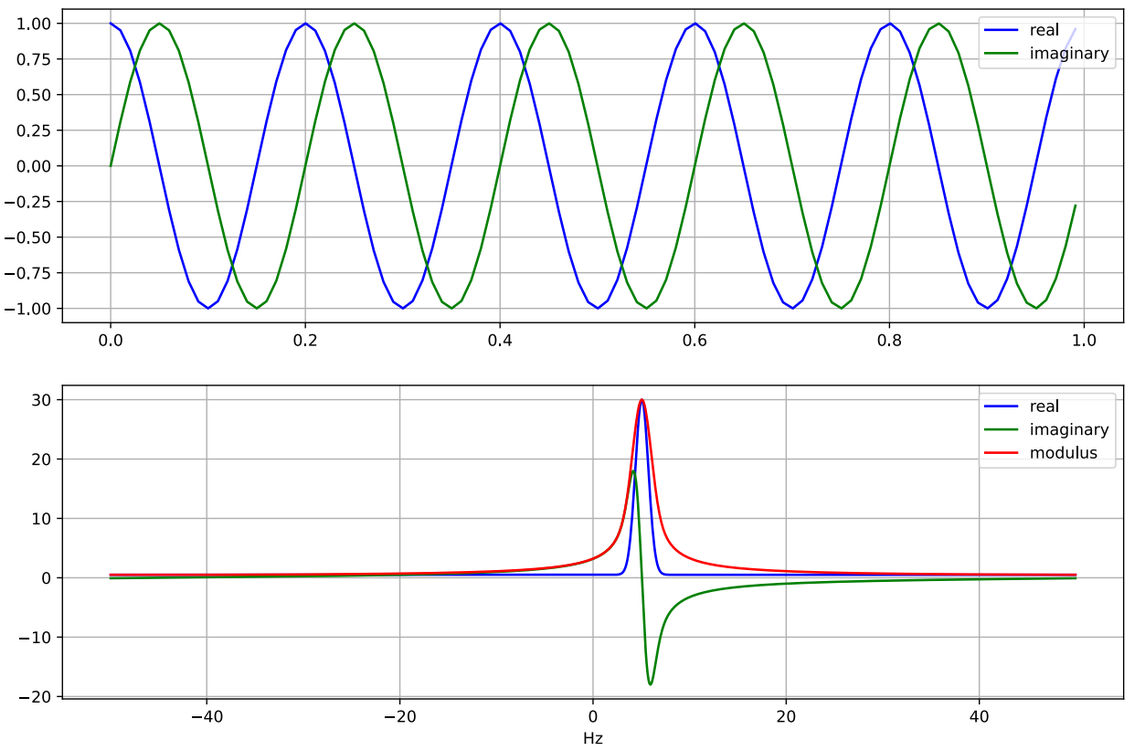
In [16]:

```
def ftc2(freq = 5.0, br = 3.0, theta = 0.0):
  "function showing the effect of a phase rotation"
  phase = theta*2*np.pi/360
  y = np.cos(2*np.pi*freq*x + phase) + 1j * np.sin(2*np.pi*freq*x + phase) # a complex signal
  gauss = np.exp(-(br*x)**2)
  yg = y*gauss
  YY = fft.fftshift(fft.fft(yg)) # fftshift() ensures the 0 freq in the center
  f, (ax1,ax2) = plt.subplots(nrows=2, figsize=(12,8))
  ax1.plot(x[0:100], y[0:100].real, 'b', label='real') # .real is for real part; 'b' is for blue
  ax1.plot(x[0:100], y[0:100].imag, 'g', label='imaginary')
  ax1.legend(loc=1);
  deltat = x[1] # as x[0] is 0, x[1] = \Delta t
  faxis = np.linspace(-1/(2*deltat),1/(2*deltat), len(YY))
  ax2.plot(faxis, YY.real, 'b', label='real')
  ax2.plot(faxis, YY.imag, 'g', label='imaginary')
  ax2.plot(faxis, abs(YY), 'r', label='modulus')
  ax2.set_xlabel('Hz')
  ax2.legend(loc=1);ax1.grid();ax2.grid()
# the angle theta is given in degrees !
interactive( ftc2, freq=(-70.0,70.), br=(0.0,20.0), theta=(0,360))
```

freq 5.00

br 3.00

theta 0

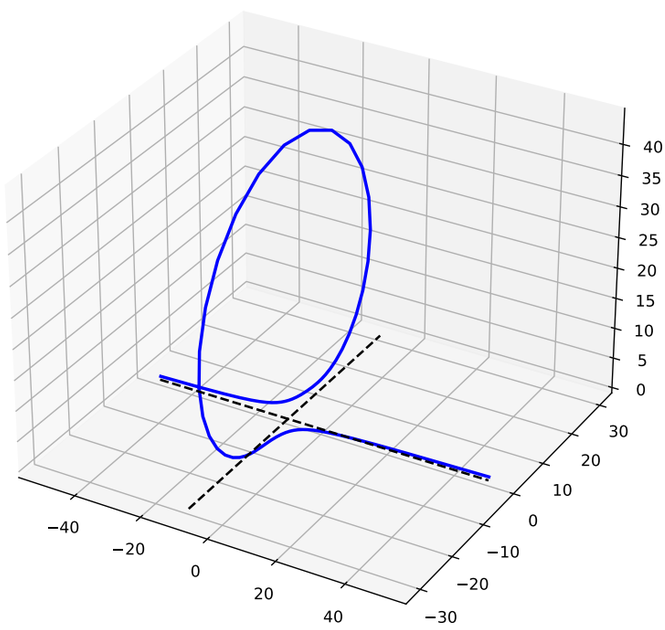


In [17]:
```
from mpl_toolkits.mplot3d import Axes3D

def ftc3(freq = -10.0,elevation=30.0, azimuth=60.0):
  br = 2
  y = np.cos(2*np.pi*freq*x) + 1j * np.sin(2*np.pi*freq*x) # a complex signal
  gauss = np.exp(-(br*x)**2)
  yg = y*gauss
  YY = fft.fft(yg)
  fig = plt.figure(figsize=(12,8))
  ax = fig.gca(projection='3d',elev=elevation, azim=-azimuth)
  deltat = x[1] # as x[0] is 0, x[1] = \Delta t
  faxis = np.linspace(-1/(2*deltat),1/(2*deltat), len(YY))
  ax.plot(faxis,fft.fftshift(YY).imag,fft.fftshift(YY).real,'b',lw=2)
  ax.plot([-50,50],[0,0],[0,0],'--k')
  ax.plot([freq,freq],[-30,30],[0,0],'--k');ax.grid()
interactive( ftc3, freq=(-40.0,40.),elevation=(1.0,90), azimuth=(0.0,180))
```

freq -10.00

elevation 30.00

azimuth 60.00




# Fourier Transform

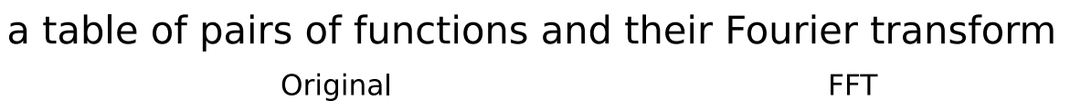
**Fourier Transform** is a matter of transforming functions. It is very useful to know beforehand the result of the FT of typical functions. From this knowledge, and with some techniques which will be shown later (linearity, inversion and convolution), and some practice, it becomes easy to have a good idea of the result of FT for a given signal.

This might be useful for instance, when trying to understand the reason of a given artefact in the spectrum.

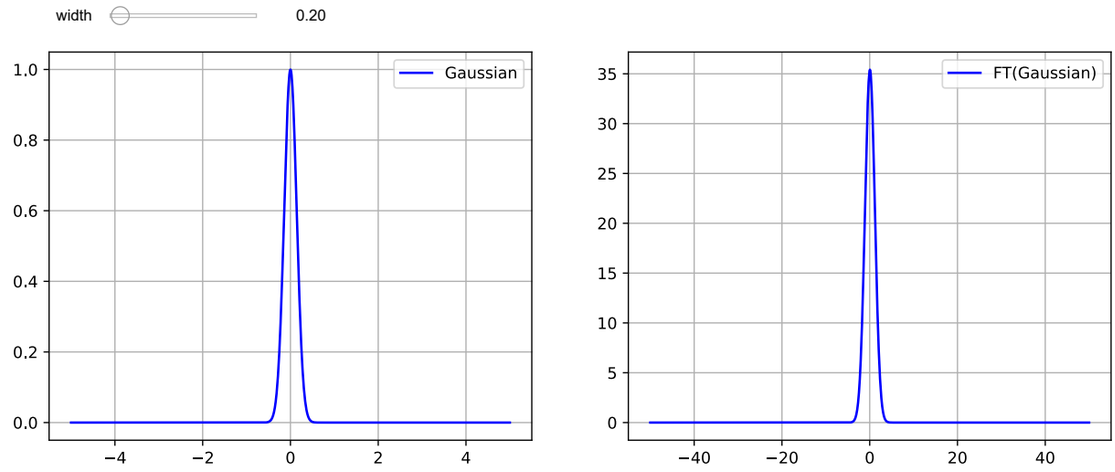
For this reason, I present know a series of "celebrity couples" of functions useful for spectroscopy.

In [18]:
```
# some hard coding first, don't worry if you don't get it
def gate(width=10):
  "return a gate function over x"
  r = np.zeros_like(x)
  r[:width] = 1.0
  r[-width:] = 1.0
  return r
def gauss(width=1.0):
  "return a centered gaussian function over x"
  r = np.exp(-((x-5)/width)**2)
  return fft.fftshift(r)
def exp(width=1.0):
  "return a centered exp function over x"
  r = np.exp(-abs(x-5)/width)
  return fft.fftshift(r)
def noise(width):
  np.random.seed(width)
  return np.random.randn(len(x))
def position(width):
  "the delta function"
  r = np.zeros_like(x)
  r[int(width*100)] = 1.0
  return r
def draw(width, f, name):
  "builds the nice drawing"
  fig, (ax1,ax2) = plt.subplots(ncols=2, figsize=(12,4.5))
  y = f(width=width)
  xax = np.linspace(-5,5,1000)
  yax = np.linspace(-50,50,1000)
  YY = fft.fftshift(fft.fft(y))
  ax1.plot(xax, fft.fftshift(y), 'b', label=name)
  ax2.plot(yax, YY.real, 'b', label='FT('+name+')')
  ax1.legend(loc=1)
  ax2.legend(loc=1)
  ax1.grid()
  ax2.grid()
  return fig
```

In [19]:
```
fig, ax1= plt.subplots( figsize=(12,0.5))
fig.text(0.1,0.9,"a table of pairs of functions and their Fourier transform",fontdict={'size': 24,})
fig.text(0.3,0,'Original',fontdict={'size': 18,}); fig.text(0.7,0,'FFT',fontdict={'size': 18,})
ax1.set_axis_off()
```

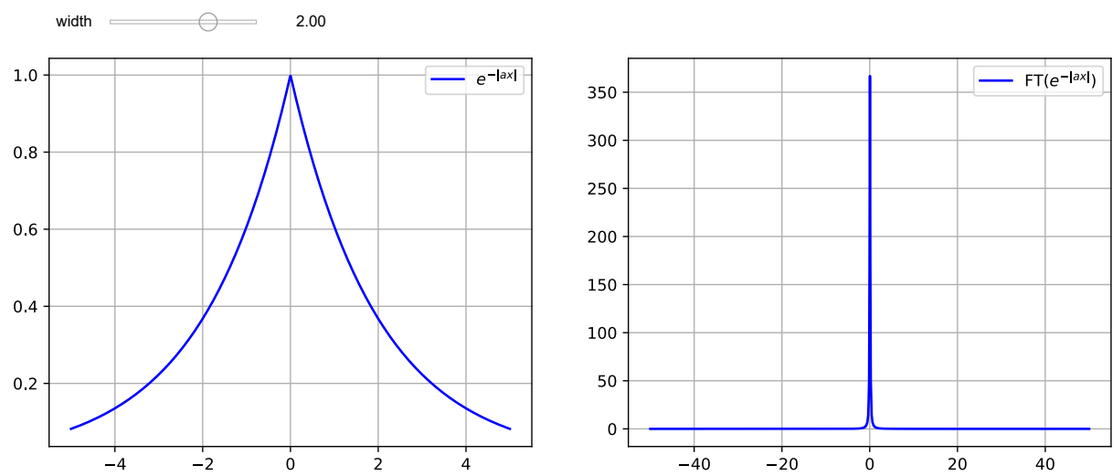



In [20]:
```
interactive(draw, width=widgets.FloatSlider(min=0.01,max=3,step=0.01,value=0.2), f=fixed(gauss), name=fixed('Gaussian'))
```




the FT of a Gaussian is another Gaussian

In [21]:
```
interactive(draw, width=widgets.FloatSlider(min=0.01,max=3,step=0.01,value=2), f=fixed(exp), name=fixed('$e^{-|ax|}$'))
```

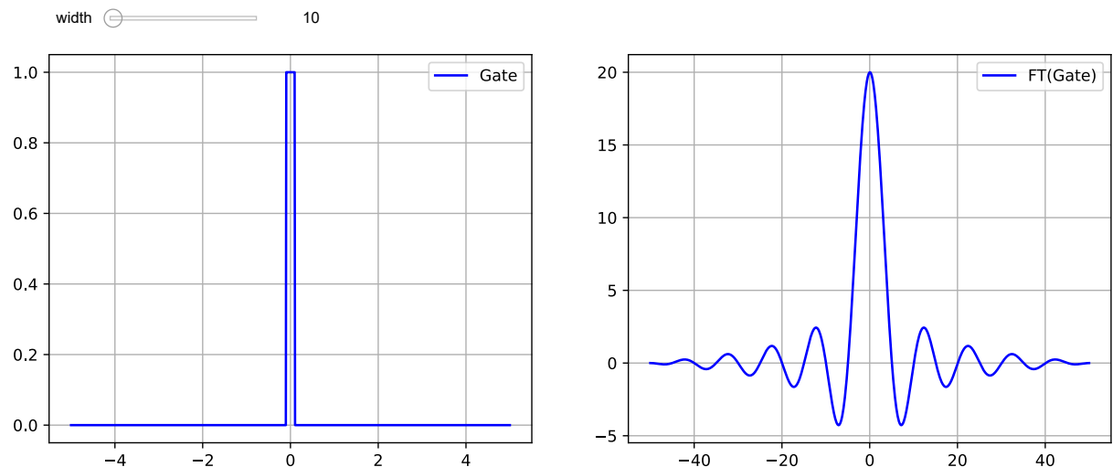



the FT of the decaying exponential is a Lorenzian line-shape, found in many spectroscopies, with generic expression for the absorptive shape:

$$F(\omega) = \frac{2A}{A^2 + (\omega - \omega o)^2} \tag{3}$$

(here $\omega o$ = 0 - no modulation)

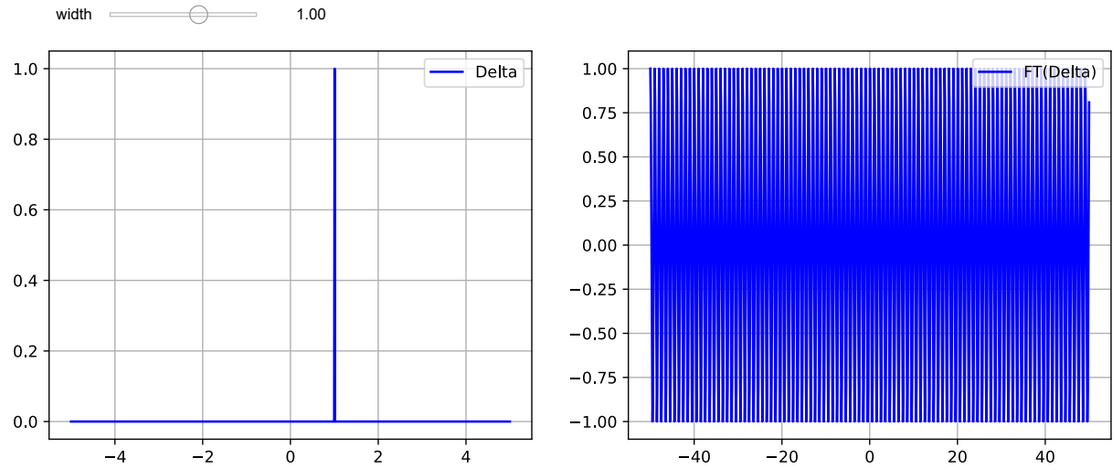
In [22]:
```
interactive(draw, width=widgets.IntSlider(min=1,max=1000,value=10), f=fixed(gate), name=fixed('Gate'))
```




the gate (1.0 for a period,0 everywhere else) has for Fourier transform the function $sinc(x) = \frac{sin(x)}{x}$.

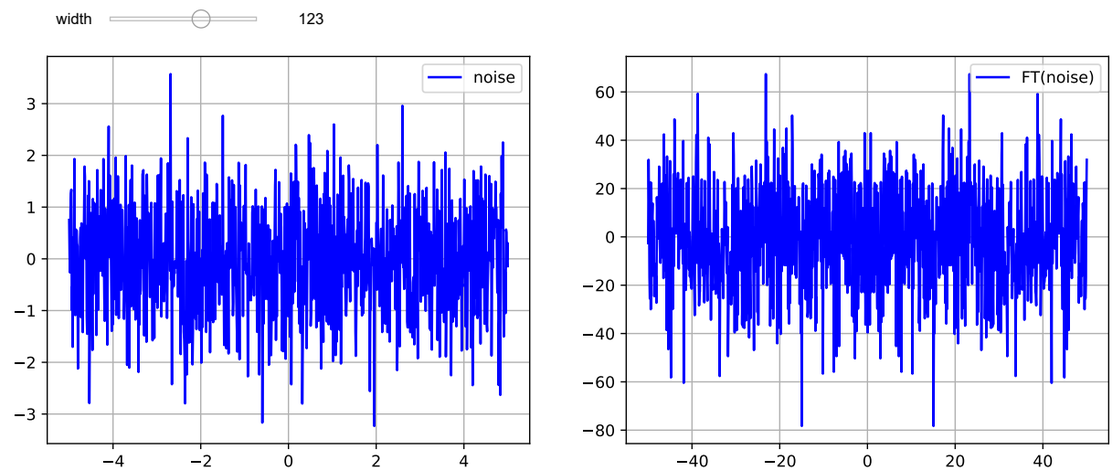
observe how the sphape on the right remains the same (size of the wiggles) and only the width is changing.

In [23]:
```
interactive(draw, width=widgets.FloatSlider(min=-5,max=5,value=1), f=fixed(position), name=fixed('Delta'))
```




the FT of a $\delta$ function at position $x_0$ is the complex sinusoid with frequency $x_0$.

In [24]:
```
interactive(draw, width=widgets.IntSlider(min=0,max=200,value=123), f=fixed(noise), name=fixed('noise'))
```




FT of a uncorrelated, centered random process with normal law (white noise), is white noise ! (width is used here as a seed).

## some other properties of the Fourier transform

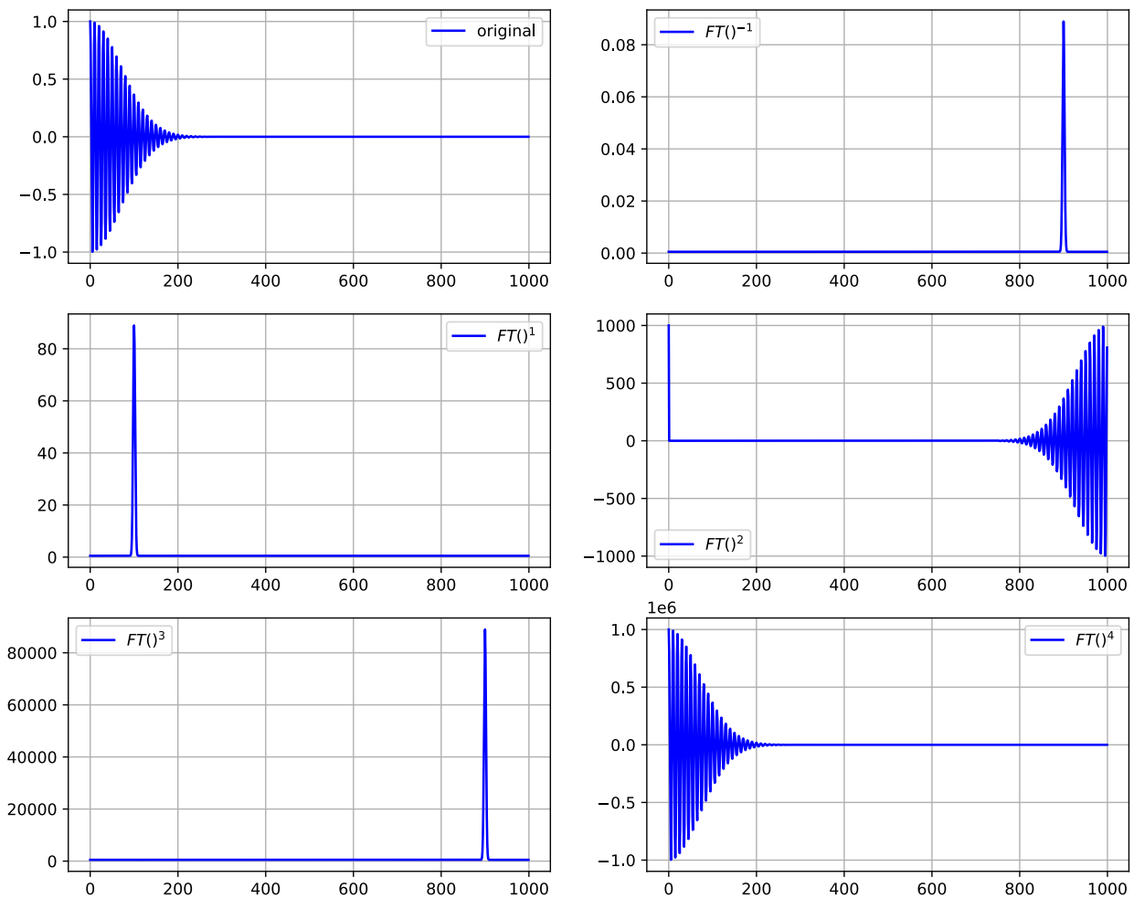
worth mentionning, and usually found in other FT courses, so I ought to put them here !

## FT is inversible

This means that no information is lost nor created by FT, it is just a different point of view and FT inverse is very similar to FT itself.

In [26]:

```
freq = 10.0
br = 1.0
y = np.cos(2*np.pi*freq*x) + 1j * np.sin(2*np.pi*freq*x) # a complex signal
gauss = np.exp(-(br*x)**2)
yg = y*gauss
f, ((ax1,ax2),(ax3,ax4),(ax5,ax6)) = plt.subplots(nrows=3,ncols=2, figsize=(12,10))
ax1.plot(yg.real,'b', label='original')
ax2.plot(fft.ifft(yg).real,'b', label='$FT()^{-1}$')
ax1.legend()
ax2.legend();ax1.grid();ax2.grid()
YY = yg
for i,ax in zip(range(4),[ax3,ax4,ax5,ax6]):
  YY = fft.fft(YY)
  ax.plot(YY.real,'b', label='$FT()^%d$'%(i+1))
  ax.legend();ax.grid()
```




as you can see, $FT()^-1$ (the inverse of the FT) is just $FT()$with the axis reversed. This means that the "celebrity couples" table can be seen right to left as well as left to right. $FT()^-1 \equiv FT()^3$ and $FT()^4$ is the identity.

This is very similar to $i$ with $i^3 = -i$ and $i^4 = 1$.

# FT is linear

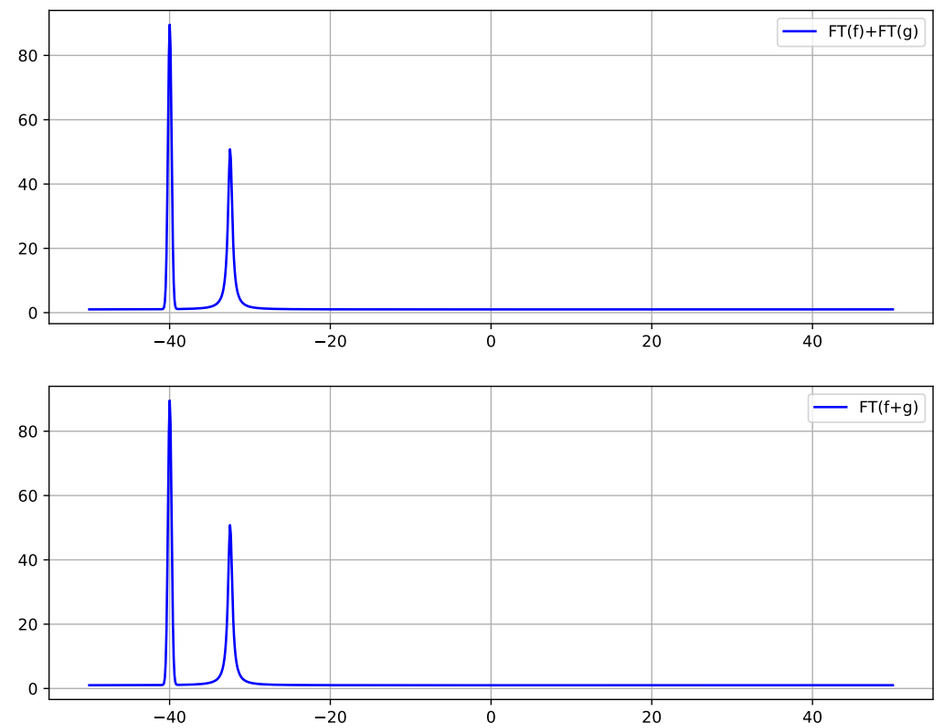
broadly speaking, it means that the FT of a sum is the sum of the FT:

$$FT(f+g) = FT(f) + FT(g) \tag{4}$$

$$FT(\lambda f) = \lambda FT(f) \tag{5}$$

where $\lambda$ is a scalar, and $f$ and $g$ are functions.

In [27]:
```
y2 = np.cos(3.5*np.pi*freq*x) + 1j * np.sin(3.5*np.pi*freq*x)
y2 = y2*np.exp(-2*x)
f,(ax1,ax2) = plt.subplots(nrows=2, figsize=(10,8))
yax = np.linspace(-50,50,1000)
ax1.plot(yax, fft.fft(y2).real+fft.fft(yg).real, 'b', label="FT(f)+FT(g)")
ax2.plot(yax, fft.fft(y2+yg).real, 'b', label="FT(f+g)")
ax1.legend()
ax2.legend();ax1.grid();ax2.grid()
```





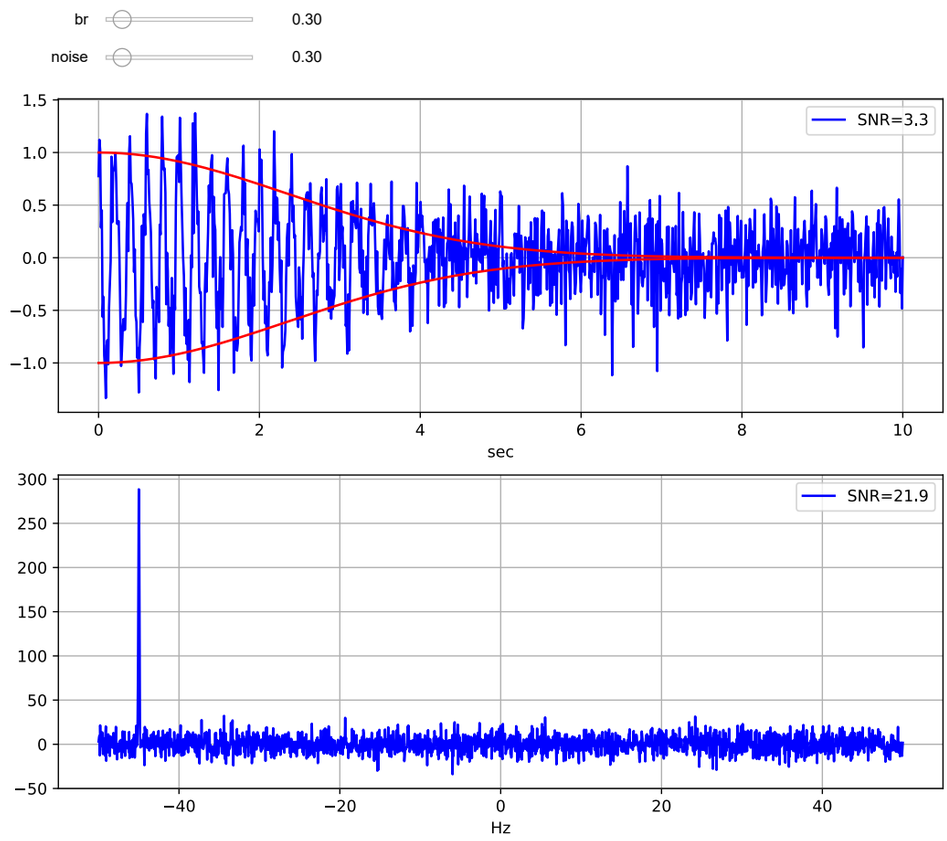


This has a several strong implications:

- you can easily estimate the FT of a composite function, expressed as the sum of simple functions
- in spectroscopy / image processing / you name it / adjacent signals/features do not interfere, they just add-up
- in measurement, there is always noise, and FT has a strong impact on signal/noise ratio

let's have a example:

In [28]:
```
def ftb2(br = 0.3, noise=0.3):
  "function showing the effect of broadening on SNR"
  freq = 5.0 # a fixed frequency
  y = np.cos(2*np.pi*freq*x) + 1j*np.sin(2*np.pi*freq*x)
  gauss = np.exp(-(br*x)**2) # this is the decay with a gaussian shape
  yg0 = y*gauss # noise free signal
  yg = yg0 + noise*(np.random.randn(len(y)) +1j*np.random.randn(len(y))) # np.random.randn() is a noise with standard
  YY = fft.fft(yg,n=2000)
  snr = max(abs(YY))/np.std(YY[len(YY)//2:]) # std is the standard deviation
  if snr < 3.0:
    SNR = 'N.A.'
  else:
    SNR = '%.1f'%snr
  f, (ax1,ax2) = plt.subplots(nrows=2 , figsize=(10,8))
  ax1.plot(x, yg.real,'b', label='SNR=%.1f'%(1/noise))
  ax1.plot(x, gauss, 'r') # draw the enveloppe
  ax1.plot(x, -gauss, 'r') # draw the enveloppe
  ax1.set_xlabel('sec')
  ax1.legend();ax1.grid()
  yax = np.linspace(-50,50,2000)
  ax2.plot(yax, YY.real,'b', label='SNR=%s'%(SNR))
  ax2.set_xlabel('Hz')
  ax2.legend();ax2.grid()
interactive( ftb2, br=(0.0,3.0,0.01), noise=(0.01,3.0))
```




Observe how the SNR after FT is most of the time higher than in time domain, (and sometime not). SNR increases for time domain signals with small broadening, and is maximum with no broadening.

Observe also how FT is able to extract a frequency from a signal completely buried into the noise, as long as this one lasts long enough.

The theoretical gain is in $\frac{\sqrt{(N)}}{2}$ where is the number of points on which the signal is observed. Here we have $\mathbf{1000}$ points, is it verified ?

A signal is considered to be non-detectable for a SNR below $\mathbf{3.0}$.

# Convolution

if linearity is for addition, convolution is for multiplication.

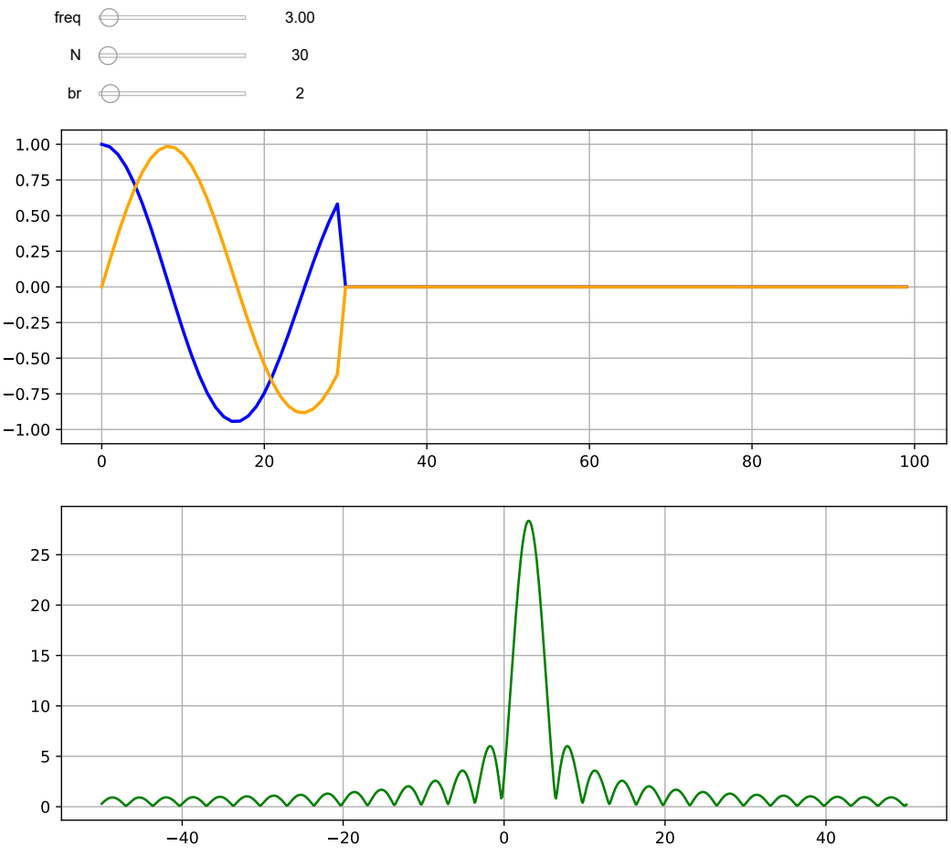
Convolution of two functions $f$ and $g$ is defined as: (noting it $\circledast$)

$$(f \circledast g)(t') = \int_{-\infty}^{\infty} f(t)g(t-t')dt \tag{6}$$

This is a symmetric operation for $f$ and $g$, and can be described as $f$ and $g$ sharing their shapes.

Let see:

In [29]:
```
def ftcv(freq = 3.0, N=30, br=2.0):
  "showing convolution"
  y = np.cos(2*np.pi*freq*x) + 1j * np.sin(2*np.pi*freq*x)
  g = np.zeros_like(x)
  g[0:N] = 1.0
  y = y*g
  y = y*np.exp(-br*x**2)
  YY = fft.fftshift(fft.fft(y))
  yax = np.linspace(-50,50,1000)
  f, (ax1,ax2) = plt.subplots(nrows=2 , figsize=(10,8))
  ax1.plot(y[0:100].real, 'b', lw=2)
  ax1.plot(y[0:100].imag, 'orange', lw=2)
  ax1.set_ylim(ymin=-1.1, ymax=1.1)
  ax2.plot(yax, abs(YY),'g');ax1.grid();ax2.grid()
interactive( ftcv, freq=(0.0,50.0), N=(4,500), br=(0,30))
```




in this example, we take the product of 3 different functions:

- a frequency (whose FT is a $\delta$ function)
- a gate (whose FT is a $sinc$ function)
- gaussian (whose FT is a $gaussian$ function)

**Observe**

- the ripples created by short gates, call wiggles
- how you can mix the $sinc$ and $gaussian$ the shapes,
- how the convolution by a $\delta$ (a multiplication by a frequency) is equivalent to a shift
- how the line moves without the shape changing
- how the shape changes without the line moving
- why it is a bad idea to have a line very close to the **Nyquist frequency**

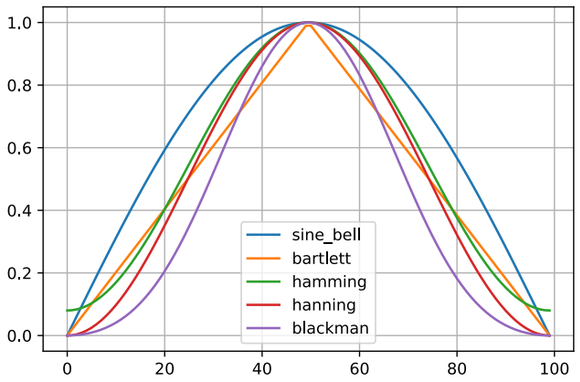
# convolution in practice

When the wiggles created by the gate are becoming a problem (because a too short observation window/time), it is usual to pre-process the data with a function which reduces these wiggles.This is call apodisation (sometimes, wrongly windowing).

Here is a list of the most common ones.

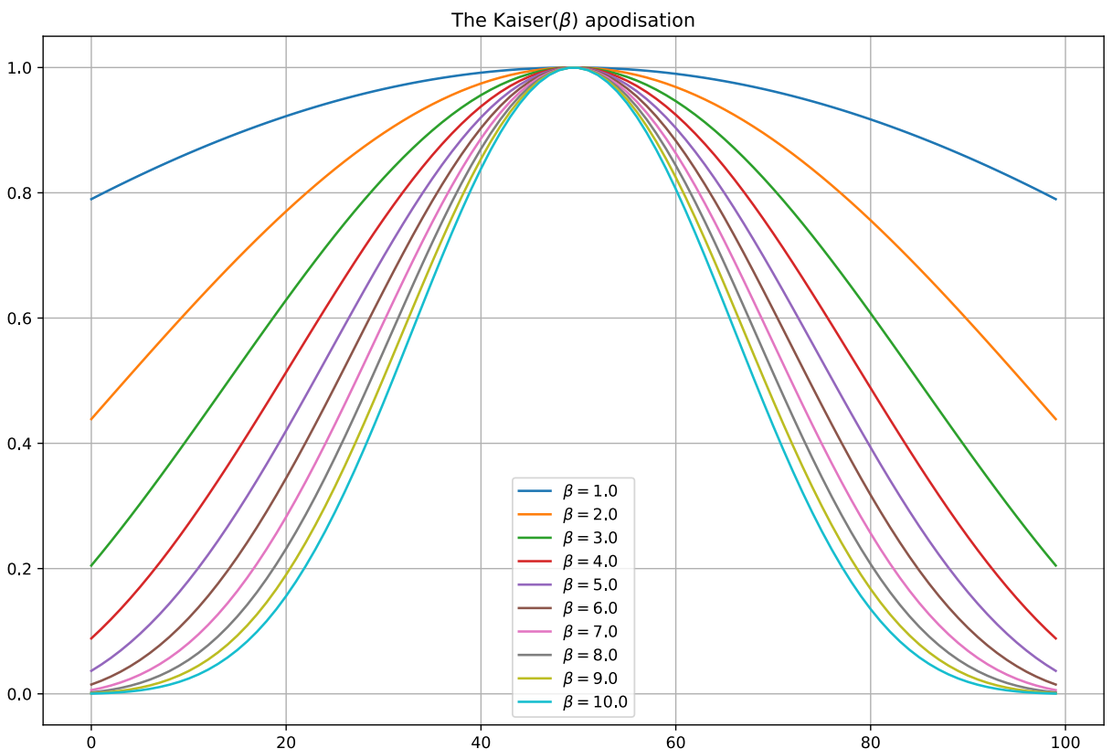
Note, these apodisations are designed for modulus spectra, when computing phased spectra, you have to use a different apodisation familly.

In [30]:
```
from numpy import blackman, hamming, hanning, bartlett, kaiser # these are pre-defined

def sine_bell(N): # this one is missing
  "defines the sine-bell apodisation window"
  return np.sin( np.linspace(0,np.pi,N) )

for apod in ("sine_bell", "bartlett", "hamming", "hanning", "blackman"):
    y = eval("%s(100)"%(apod))
    plt.plot(y, label=apod);plt.grid()
plt.legend();
```

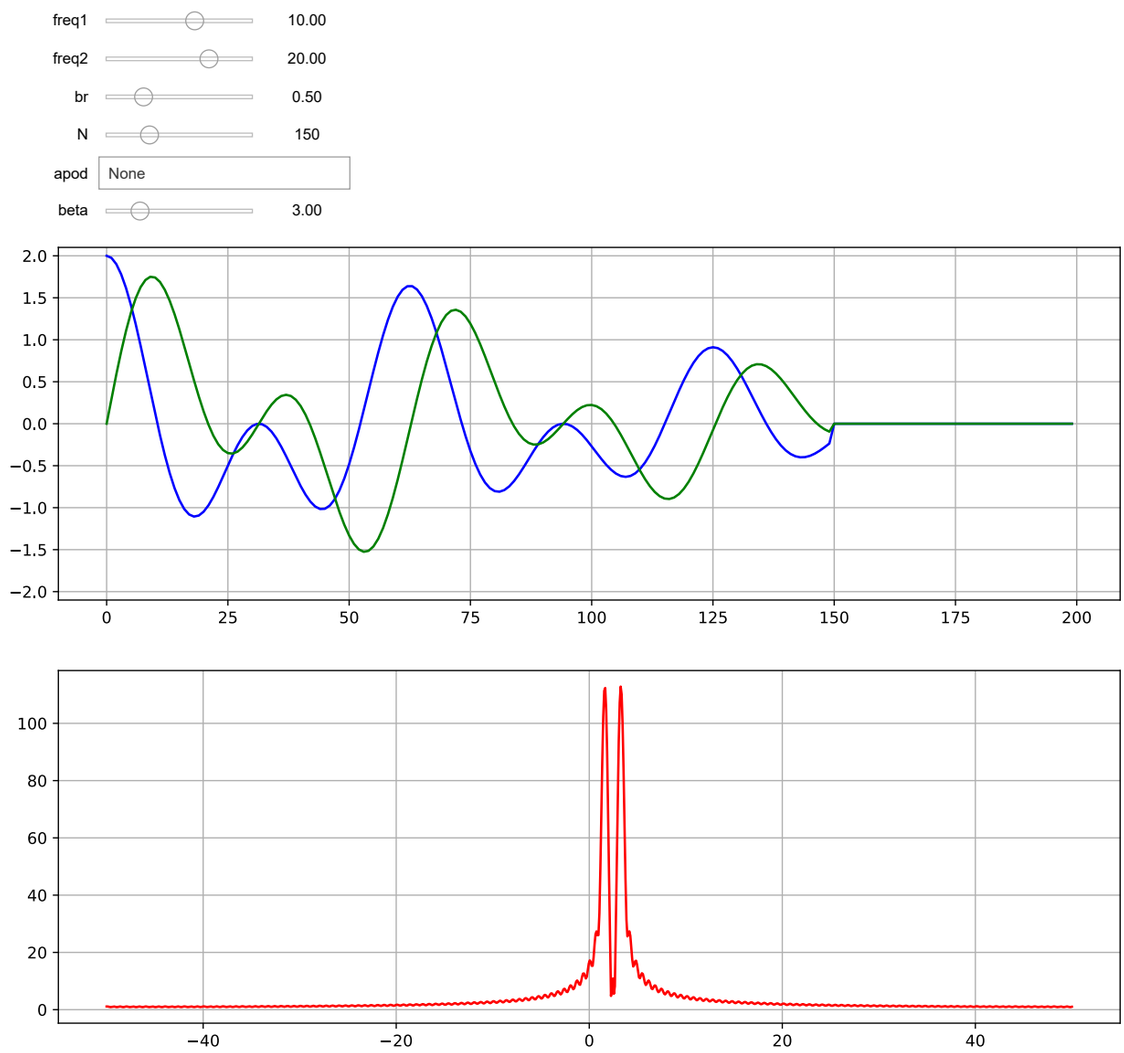



The **kaiser** function is also very usefull as a generic/tunable apodisation function.

In [31]:
```
plt.figure(figsize=(12,8))
for beta in range(1,11):
  plt.plot(kaiser(100, beta), label=r"$\beta=%.1f$"%beta)
plt.legend();plt.grid()
plt.title(r'The Kaiser($\beta$) apodisation');
```



In [32]:
```
apodlist = ["None", "sine_bell", "bartlett", "hamming", "hanning", "blackman", "kaiser"]

def ftapod(freq1 = 10.0, freq2 = 20.0, br=0.5, N=150, apod="empty", beta=3.0):
  "showing convolution"
  y = np.cos(freq1*x) + 1j * np.sin(freq1*x) + np.cos(freq2*x) + 1j * np.sin(freq2*x)
  g = np.zeros_like(x)
  if apod == 'None':
    g[0:N] = 1.0
  elif apod == 'kaiser':
    g[0:N] = eval("%s(%d,%f)"%(apod,N,beta))
  elif apod != 'None':
    g[0:N] = eval("%s(%d)"%(apod,N))
  y *= g*np.exp(-br*x**2)
  YY = fft.fftshift(fft.fft(y))
  yax = np.linspace(-50,50,1000)
  f, (ax1,ax2) = plt.subplots(nrows=2, figsize=(12,9))
  ax1.plot(y[0:200].real,'b')
  ax1.plot(y[0:200].imag,'g')
  ax1.set_ylim(ymin=-2.1, ymax=2.1)
  ax2.plot(yax, abs(YY),'r');ax1.grid();ax2.grid()
interactive( ftapod, freq1=(-50.0,50.0), freq2=(-50.0,50.0), br=(0.0,2.0), N=(4,500), apod=apodlist, beta=(1.0,10.0))
```




here, you can simulate a signal, and play with the apodisation window.

In this case, there are two lines, that you can control independently. The windows have been set more or less in increasing order of broadening. Note that only kaiser is controlled with the beta parameter, and covers most of the features of the other windows.

Try different combinaitions of line-width, separation, number of points and check the effect of each apodisation on it.

## Observe

- how it always a trade-off between resolution and nice line-shape.
- Consider shape, FWMH, separation, wiggles intensities, ...
- how apodisation may, in some cases, actually improve the line shape