In this hands-on session, we generate some synthetic sounds and perform some time-frequency analysis. We will *hear* the Uncertainty Relation at work.

In [1]:

```
# standard python libs for math, signal processing and plotting
import numpy as np
import scipy.signal as ss
import matplotlib.pyplot as plt
from math import log10

# python libs for the notebook interaction
from ipywidgets import interact
import ipywidgets as widgets

# custom modules
import scipy.io.wavfile as wav
import simpleaudio as sa    # generates sounds
from tftb.processing import Spectrogram    # computes a time-frequency "map" of a s

# with the following, the plots are opened as separate windows in full resolution
#%matplotlib

fs = 44100          # sampling frequency in audio cards is 44.1 kHz for CD-quality a
samples = 65536     # samples used for the FFT
```

The Fourier Transform

Theory recap

The Fourier Transform $S(f)$ of a time-dependent signal $s(t)$ is defined as:

$$S(f) := \int_{-\infty}^{+\infty} s(t)e^{-i2\pi ft}dt$$

The Discrete Fourier Transform $S(m)$ of a finite sequence $s(n)$ is defined as:

$$S(k) = \sum_{n=0}^{N-1} s(n)e^{-i2\pi kn/N} \quad k = 0, 1, \ldots, N-1$$

Where N is the length of the $s(n)$ sequence, as well as the length of the $S(m)$ sequence of frequency samples.

Testing live

Let's explore the properties of some simple audio signals live. The `quantizeandplay()` function is a simple quantizer to play the signal with either 8-bit or 16-bit levels. Note that the quantization process introduces an error equal to half the least significant bit, therefore the Signal-to-Noise ratio (SNR) is:

$$SNR_{dB} = 2 \cdot 10\ log_{10}(2^n) \approx 6n\ [dB]$$

Where n is the number of bits. Therefore with 8 bit (typical for radio-quality or mobile-phone-quality audio), the SNR is 48 dB. With 16 bit (the standard for CD-quality audio), the SNR is 96 dB.

In [4]:
```
def plottimefreq(s, duration, small=False):
    # plot the time series of the signal s
    plt.subplots(figsize=(25, 5))
    ax = plt.subplot(1, 3 - int(small), 1)
    plt.plot(np.real(s))
    plt.xlim(0)
    ax.set_xlabel('t [ms]')
    maxx = int(duration*10 + 1)*100
    ax.set_xticks(np.arange(0, maxx*fs/1000, maxx*fs/10000, dtype=int))
    ax.set_xticklabels(np.arange(0, maxx, maxx/10, dtype=int))
    plt.grid(which='major')
    plt.title("Wave packet")

    # also compute and plot power spectral density
    ax = plt.subplot(1, 3 - int(small), 2)
    s = np.pad(s, (0, samples-s.size), mode='constant')
    W = np.abs(np.fft.fft(s) ** 2)
    f = np.fft.fftfreq(s.size, 1/fs)
    plt.plot(f, W)
    plt.xlim(20, 10000)
    #formatter = LogFormatter(labelOnlyBase=False, minor_thresholds=(1, 0.1))
    #ax.get_xaxis().set_minor_formatter(formatter)
    ax.set_xlabel('f [Hz]')
    plt.ylim(1E-4)
    plt.xscale('log')
    plt.yscale('log')
    plt.grid(which='both')
    plt.title("Power spectral density (log/log)")
    plt.show()

def quantizeandplay(s, quant):
    if quant not in [4, 8, 16]:
        raise Exception('Unsupported quantization')
    qfactor = (2**(quant-1) - 1) * 1.0 / np.max(np.abs(s))
    playable = s * qfactor
    playable = playable.astype(np.int16 if quant == 16 else np.int8)
    # stop any ongoing play
    sa.stop_all()
    # play on a single audio channel with either 1 or 2 bytes per sample according
    sa.play_buffer(playable, 1, 2 if quant == 16 else 1, fs)
    # rescale the quantized signal
    return playable.astype(float) / qfactor
```

The interactive code below creates simple audio signals, plays them and shows the plots from the above function

In [5]:

```
@interact(f=widgets.FloatLogSlider(min=log10(20), max=log10(20000), value=440, con
          duration=widgets.FloatSlider(min=0.1, max=1.4, value=1, step=0.01, conti
          #quantization=widgets.RadioButtons(options=[16, 8, 4, None]),
         )
def playwavelet(f, duration):
    t = np.linspace(0, duration, int(duration * fs), False)
    quantization = 16

    # generate the wave
    s = np.sin(2 * np.pi * f * t)

    # play it if required, and plot it
    if not quantization:
        q = s
    else:
        q = quantizeandplay(s, quantization)
    plottimefreq(q, duration)
```

The Uncertainty Relation in action

In the following, we plot a "single-frequency" wavelet and its power spectral density, and compare their time vs. frequency spreads:

- by varying the duration of the wavelet
- by varying the enveloping or windowing signal (*rectangular* i.e. no window, *Hann*, *Hamming*, or *Gaussian*)
- by varying the fraction of the signal that is smoothed by the window

We can "see" how a short time spread yields a large frequency spread, and we can "hear" how the sound progressively becomes a *tic* with no clear pitch! Also, note how the absence of a windowing signal produces a *click* at the beginning and at the end of the signal, whereas the smoothest sound comes with the Gaussian windowing.

In [8]:

```
@interact(f=widgets.FloatLogSlider(min=log10(20), max=log10(20000), value=100, con
          duration=widgets.FloatSlider(min=0.01, max=0.5, value=0.2, step=0.01, co
          window=widgets.RadioButtons(options=['rect', 'hann', 'hamming', 'gaussia
          gauss_stdev=widgets.IntSlider(min=100, max=8000, value=5000, continuous_
          win_frac=widgets.FloatSlider(min=0.01, max=1.00, value=1.00, step=0.01,
          #quantization=widgets.RadioButtons(options=[16, 8, 4, None]),
         )
def playwavelet(f, duration, window, gauss_stdev, win_frac):
    t = np.linspace(0, duration, int(duration * fs), False)
    quantization = 16

    # generate the fundamental wave
    s = np.sin(2 * np.pi * f * t)
    # use a window to smooth begin and end
    if window == 'hann':
        w = np.hanning(s.size * win_frac)
    elif window == 'hamming':
        w = np.hamming(s.size * win_frac)
    elif window == 'gaussian':
        w = ss.gaussian(int(s.size * win_frac), duration*win_frac*gauss_stdev)
    if window != 'rect':
        # apply the window at the ramp up and ramp down of the signal
        for i in range(int(w.size/2)):
            s[i] *= w[i]
            s[s.size-int(w.size/2)+i] *= w[int(w.size/2)+i]

    # play it if required, and plot it
    if not quantization:
        p = s
    else:
        p = quantizeandplay(s, quantization)
    plottimefreq(p, duration)
```

Time-Frequency Analysis

The energy distribution of a transient signal can be obtained for instance with the *spectrogram*, defined from the *Short-Time Fourier Transform*:

$$S_{st}(t, f) = \int_{-\infty}^{+\infty} s(\tau) w(\tau - t) e^{-i2\pi f\tau} d\tau$$

Where $w(t)$ is a windowing function, typically Hamming or Gaussian. The spectrogram is defined as $|S_{st}(t, f)|^2$.

The Uncertainty Relation can be seen at play in the time-frequency plane as the "area" occupied by a signal cannot be arbitrarily small: a signal can either be localized over the t axis or over the f axis, not both. Again, the maximum localization in both axis, i.e. the minimal area, is achieved with a Gaussian signal.

In [7]:

```
@interact(f=widgets.FloatLogSlider(min=log10(100), max=log10(1000), value=300, con
          duration_ms=widgets.IntSlider(min=1, max=500, value=100, step=1, continu
          window=widgets.RadioButtons(options=['rect', 'hann', 'hamming', 'gaussia
          win_frac=widgets.FloatSlider(min=0.01, max=1.00, value=1.00, step=0.01,
         )
def playwavelet(f, duration_ms, window, win_frac):
    t = np.linspace(0, duration_ms/1000, int(duration_ms/1000 * fs), False)

    # generate the fundamental wave
    s = 1000 * np.sin(2 * np.pi * f * t)
    # use a window to smooth begin and end
    if window == 'hann':
        w = np.hanning(s.size * win_frac)
    elif window == 'hamming':
        w = np.hamming(s.size * win_frac)
    elif window == 'gaussian':
        w = ss.gaussian(int(s.size * win_frac), duration_ms*win_frac*4)
    if window != 'rect':
        # apply the window at the ramp up and ramp down of the signal
        for i in range(int(w.size/2)):
            s[i] *= w[i]
            s[s.size-int(w.size/2)+i] *= w[int(w.size/2)+i]

    # pad the signal, resample to speed up computation, and compute the spectrogra
    # the window function is 200 times shorter than the input signal to accurately
    padding = int((samples/2-s.size)/2)
    sp = Spectrogram(ss.resample(np.pad(s, (padding, padding), mode='constant'), 2
                     fwindow=np.hamming(int(s.size/100)+1))    # ss.gaussian(int(s.
    sp.run()
    sp.plot(kind='contour', scale='log', threshold=0.01)

    # add the usual time domain and frequency domain plots for reference
    plottimefreq(s, duration_ms/1000, small=True)
```