pytftb Documentation

Release 0.0.1

Jaidev Deshpande

Dec 13, 2017

Contents

Contents:

CHAPTER 1

Introduction to PyTFTB

1.1 About PyTFTB

The PyTFTB project began as a Python implementation of the TFTB toolbox developed by François Auger, Olivier Lemoine, Paulo Gonçalvès and Patrick Flandrin. While the Python implementation (henceforth referred to as PyTFTB) and the MATLAB implementation (henceforth referred to as TFTB) are similar in the core algorithms and the basic code organization, the very nature of the Python programming language has motivated a very different approach in architecture of PyTFTB (differences between the two packages have been discussed in detail in the next section). Thus, someone who is familiar with TFTB should find the PyTFTB API comfortably within grasp, and someone who is beginning with PyTFTB should find it a fully self contained library to use.

1.2 Comparison of TFTB and PyTFTB

TFTB is broadly a collection of MATLAB functions and demos that demonstrate the use of these functions. A detailed reference of these functions can be found here. The fact that Python is a general purpose programming language affords the users and the developers a lot of freedom, especially with regard to code reuse and interfacing. The important differences in implementation are as follows:

1. PyTFTB makes heavy use of Python's object oriented design. This allows for code reuse and interfacing. Algorithms that are very closely related to each other can inherit from thew same base class and reuse each others methods.

2. In TFTB, visualization of time frequency distributions is handled by dedicated functions like *tfrview* and *tfrqview* whereas in PyTFTB, they are tightly coupled to the specific representation being computed.

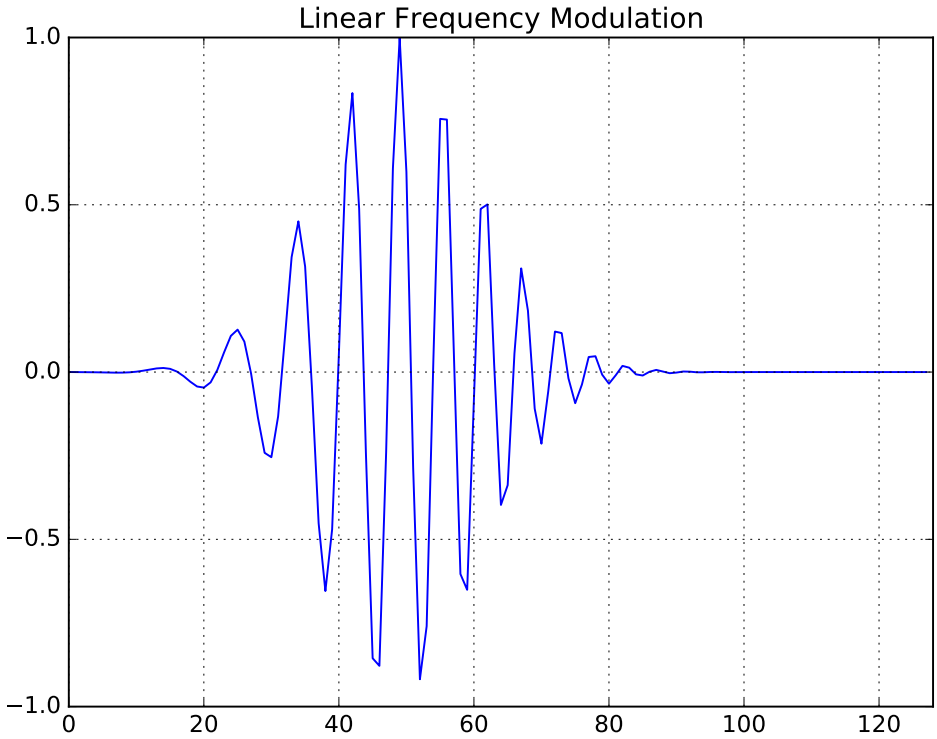
3. PyTFTB is heavily dependent on the SciPy stack - especially the NumPy and the SciPy libraries. Whichever piece of code can be delegated to these libraries is delegated to them.

1.3 Quick Start

1.3.1 Example 1: Linear Frequency Modulation

Let us consider first a signal with constant amplitude, and with a linear frequency modulation - i.e. a signal such that its amplitude remains constant, but frequency increases linearly with time - varying from 0 to 0.5 in normalized frequency (ratio of the frequency in Hertz to the sampling frequency, with respect to the Shannon sampling theorem). This signal is called a chirp, and as its frequency content is varying with time, it is a non-stationary signal. To obtain such a signal, we can use the function `tftb.generators.fmlin`, which generates a linear frequency modulation.

```
>>> from tftb.generators import amgauss, fmlin
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> z = amgauss(128, 50, 40) * fmlin(128, 0.05, 0.3, 50)[0]
>>> plt.plot(np.real(z))
>>> plt.title("Linear Frequency Modulation")
>>> plt.show()
```

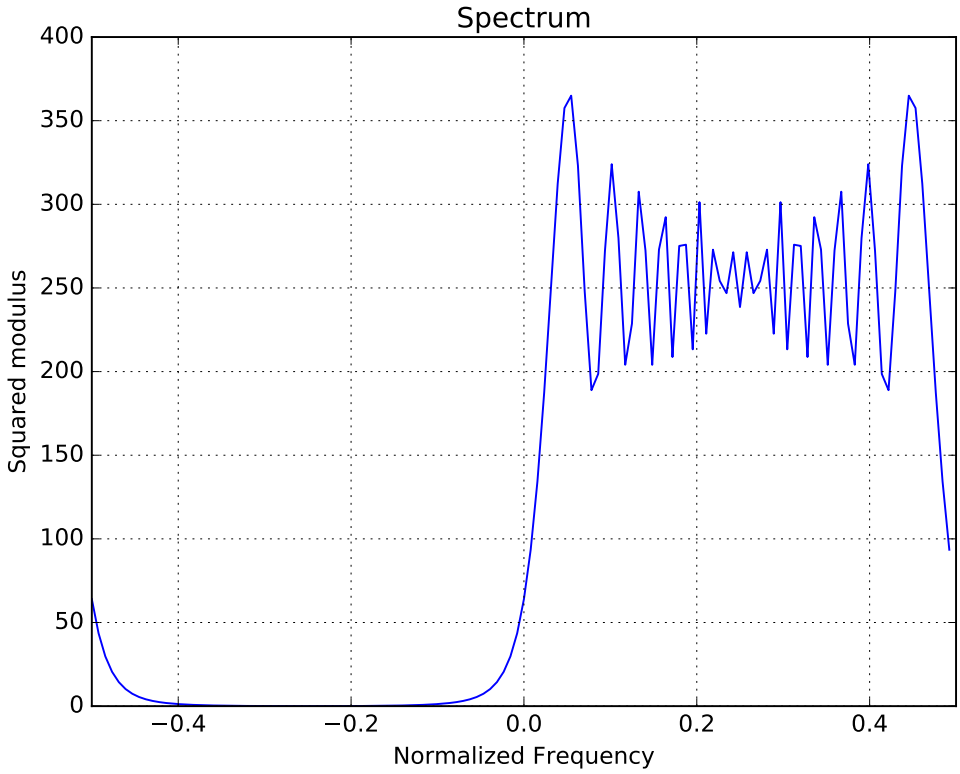


From this time-domain representation, it is difficult to say what kind of modulation is contained in this signal: what are the initial and final frequencies, is it a linear, parabolic, hyperbolic... frequency modulation?

If we now consider the energy spectrum of the signal `z` by squaring the modulus of its Fourier transform (using the `numpy.fft.fft` function):

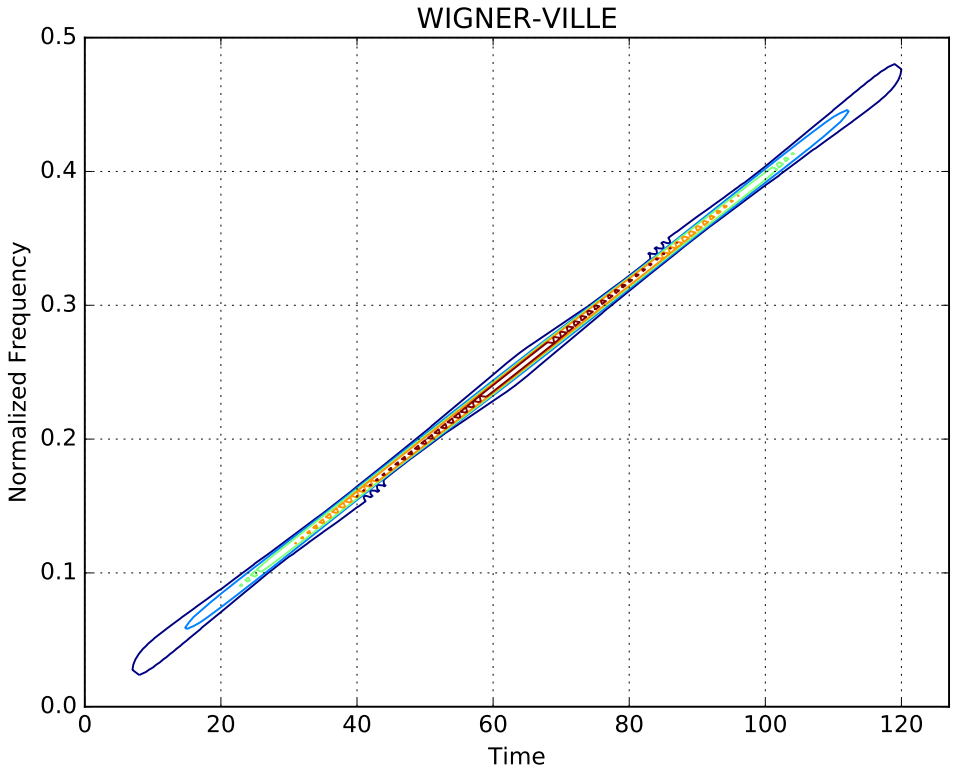
```
>>> import numpy as np
>>> dsp1 = np.fft.fftshift(np.abs(np.fft.fft(z)) ** 2)
>>> plt.plot(np.arange(-64, 64, dtype=float) / 128.0, dsp1)
>>> plt.xlim(-0.5, 0.5)
```

```
>>> plt.title('Spectrum')
>>> plt.ylabel('Squared modulus')
>>> plt.xlabel('Normalized Frequency')
>>> plt.grid()
>>> plt.show()
```



we still can not say, from this plot, anything about the evolution in time of the frequency content. This is due to the fact that the Fourier transform is a decomposition on complex exponentials, which are of infinite duration and completely unlocalized in time. Time information is in fact encoded in the phase of the Fourier transform (which is simply ignored by the energy spectrum), but their interpretation is not straightforward and their direct extraction is faced with a number of difficulties such as phase unwrapping. In order to have a more informative description of such signals, it would be better to directly represent their frequency content while still keeping the time description parameter. This is precisely the aim of time-frequency analysis. To illustrate this, let us try the Wigner-Ville distribution on this signal.

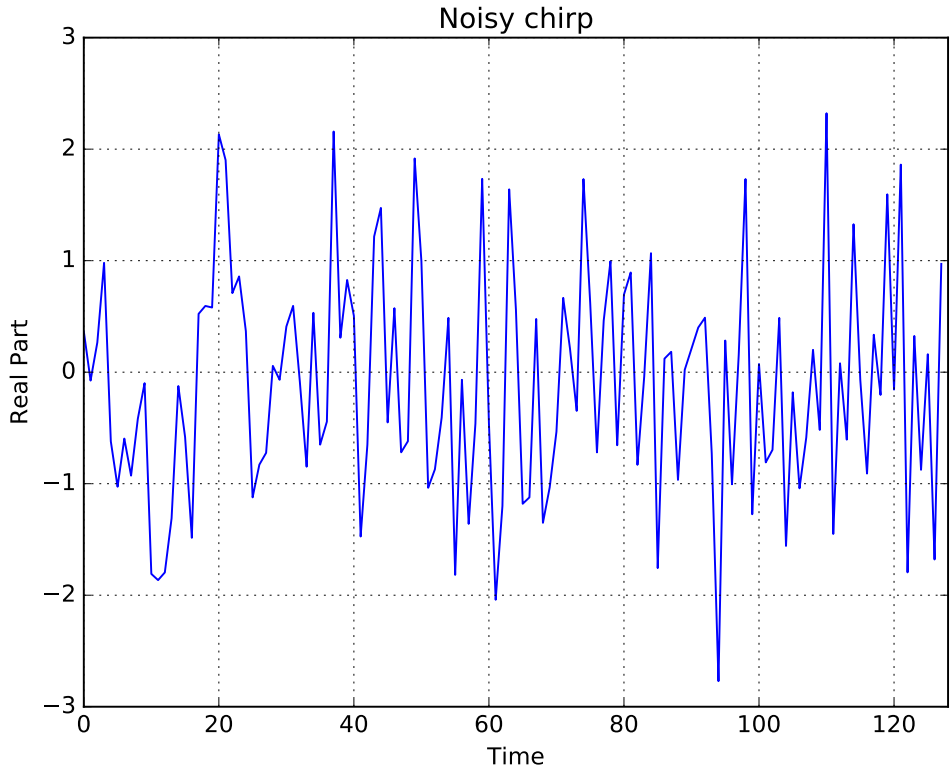
```
>>> from tftb.processing import WignerVilleDistribution
>>> wvd = WignerVilleDistribution(z)
>>> wvd.run()
>>> wvd.plot(kind='contour', extent=[0, n_points, fmin, fmax])
```



we can see that the linear progression of the frequency with time, from 0 to 0.5, is clearly shown.

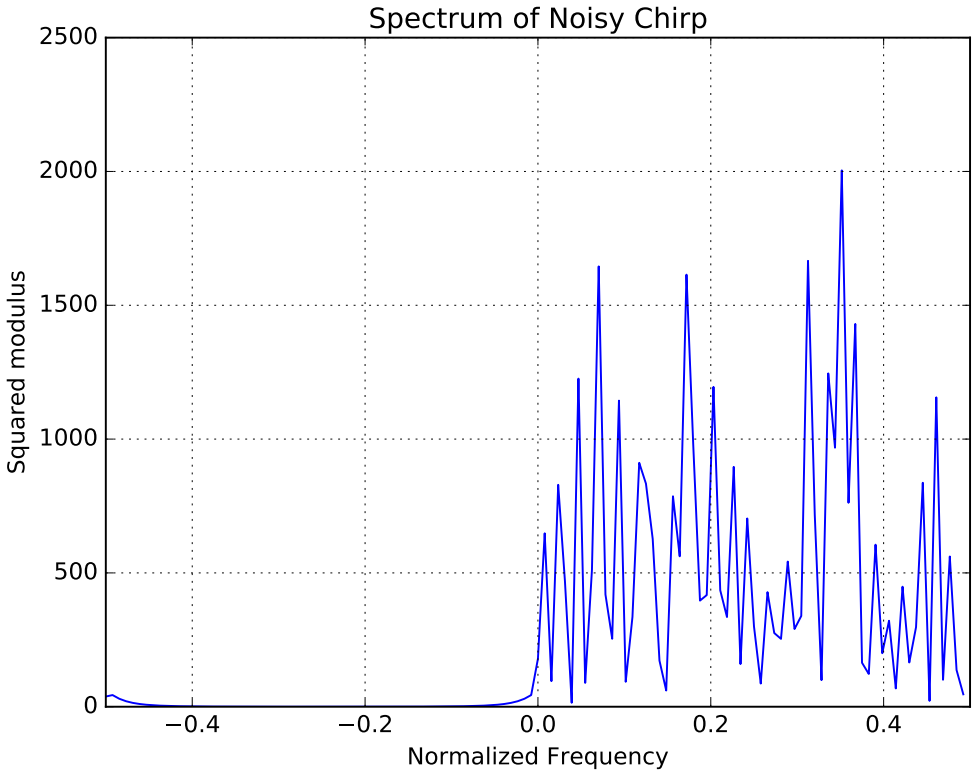
If we now add some complex white gaussian noise on this signal,

```
>>> from tftb.generators import sigmerge, noisecg
>>> noisy_signal = sigmerge(z, noisecg(128), 0)
>>> plt.plot(np.real(noisy_signal))
>>> plt.xlim(0, 128)
>>> plt.title('Noisy chirp')
>>> plt.ylabel('Real Part')
>>> plt.xlabel('Time')
>>> plt.grid()
>>> plt.show()
```



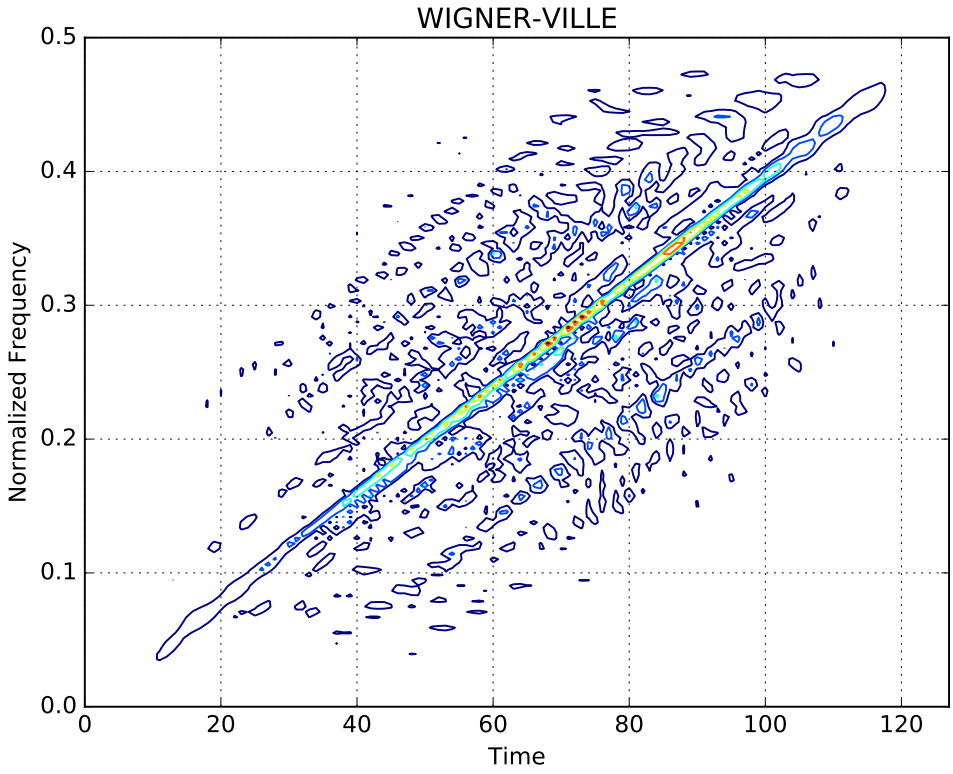
and consider the spectrum of it,

```
>>> dsp1 = np.fft.fftshift(np.abs(np.fft.fft(noisy_signal)) ** 2)
>>> plt.plot(np.arange(-64, 64, dtype=float) / 128.0, dsp1)
>>> plt.xlim(-0.5, 0.5)
>>> plt.title('Spectrum of Noisy Chirp')
>>> plt.ylabel('Squared modulus')
>>> plt.xlabel('Normalized Frequency')
>>> plt.grid()
>>> plt.show()
```



it is worse than before to interpret these plots. On the other hand, the Wigner-Ville distribution still show quite clearly the linear progression of the frequency with time.

```
>>> wvd = WignerVilleDistribution(noisy_signal)
>>> wvd.run()
>>> wvd.plot(kind='contour')
```

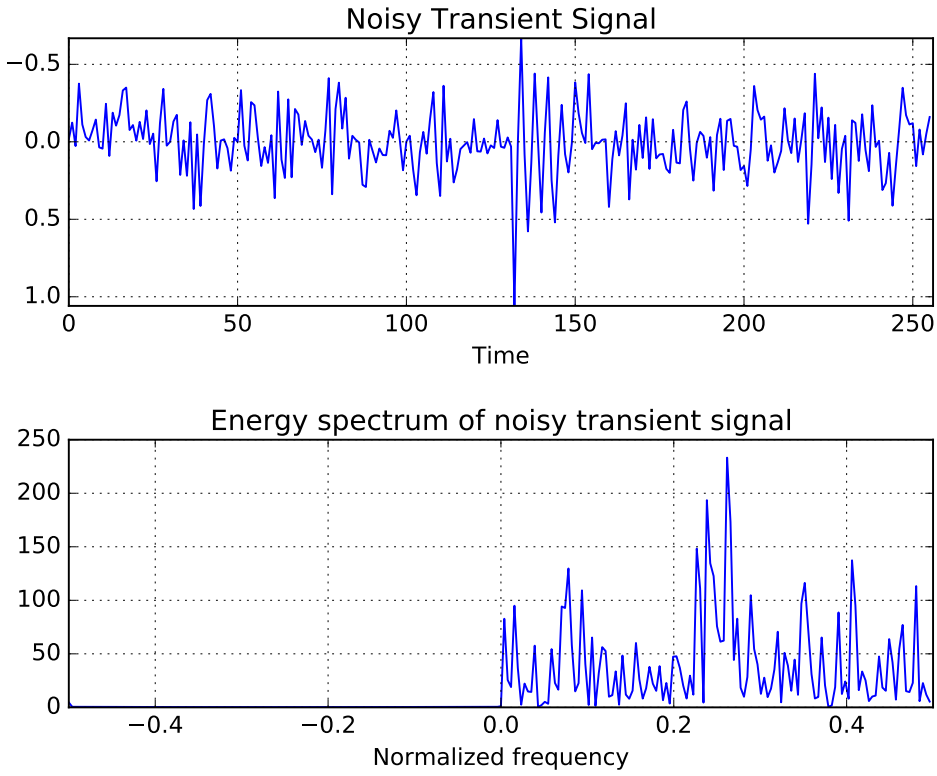


1.3.2 Example 2: Noisy Transient Signal

The second introductory example is a transient signal embedded in a -5 dB white gaussian noise. This transient signal is a constant frequency modulated by a one-sided exponential amplitude. The signal and its spectrum are generated as follows:

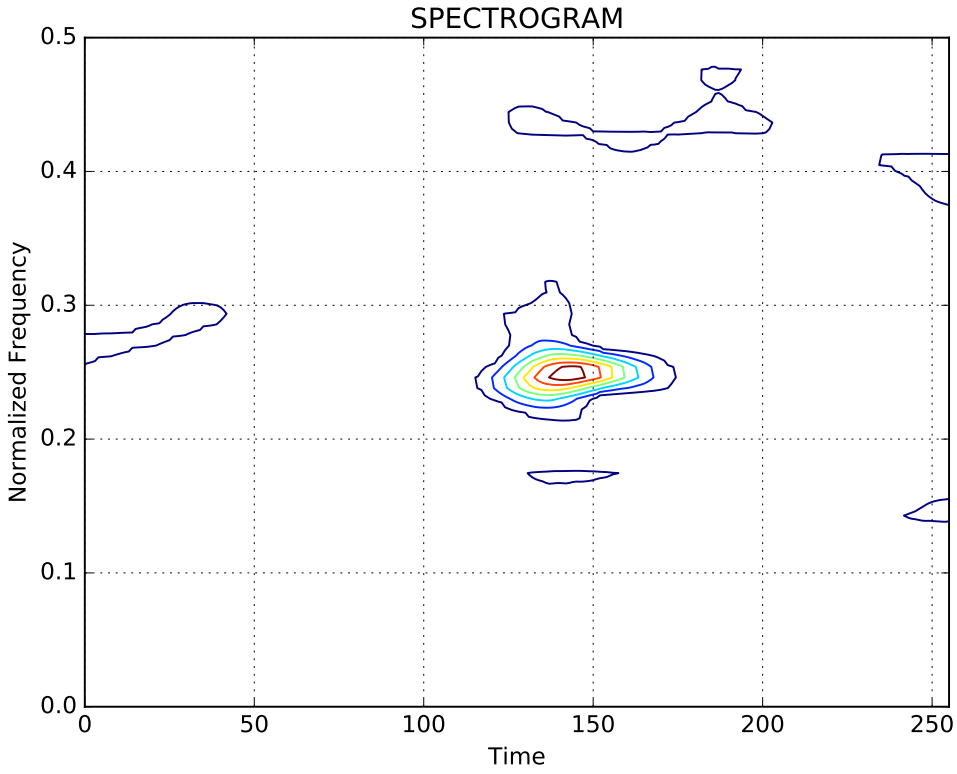
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from tftb.generators import amexpos, fmconst, sigmerge, noisecg
>>>
>>> # Generate a noisy transient signal.
>>> transsig = amexpos(64, kind='unilateral') * fmconst(64)[0]
>>> signal = np.hstack((np.zeros((100,)), transsig, np.zeros((92,))))
>>> signal = sigmerge(signal, noisecg(256), -5)
>>> fig, ax = plt.subplots(2, 1)
>>> ax1, ax2 = ax
>>> ax1.plot(np.real(signal))
>>> ax1.grid()
>>> ax1.set_title('Noisy Transient Signal')
>>> ax1.set_xlabel('Time')
>>> ax1.set_xlim((0, 256))
>>> ax1.set_ylim((np.real(signal).max(), np.real(signal.min())))
>>>
>>> # Energy spectrum of the signal
>>> dsp = np.fft.fftshift(np.abs(np.fft.fft(signal)) ** 2)
>>> ax2.plot(np.arange(-128, 128, dtype=float) / 256, dsp)
>>> ax2.set_title('Energy spectrum of noisy transient signal')
>>> ax2.set_xlabel('Normalized frequency')
```

```
>>> ax2.grid()
>>> ax2.set_xlim(-0.5, 0.5)
>>>
>>> plt.subplots_adjust(hspace=0.5)
>>>
>>> plt.show()
```



From these representations, it is difficult to localize precisely the signal in the time-domain as well as in the frequency domain. Now let us have a look at the spectrogram of this signal:

```
>>> from scipy.signal import hamming
>>> from tftb.processing import Spectrogram
>>> fwindow = hamming(65)
>>> spec = Spectrogram(signal, n_fbins=128, fwindow=fwindow)
>>> spec.run()
>>> spec.plot(kind="contour", threshold=0.1, show_tf=False)
```



the transient signal appears distinctly around the normalized frequency 0.25, and between time points 125 and 160.

CHAPTER 2

Non Stationary Signals

2.1 Frequency Domain Representations

The most straightforward frequency domain representation of a signal is obtained by the Fourier transform of the signal, defined as:

$$X(\nu) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi\nu t} dt$$

The spectrum $X(\nu)$ is perfectly valid, but the Fourier transform is essentially an integral over time. Thus, we lose all information that varies with time. All we can tell from the spectrum is that the signal has two distinct frequency components. In other words, we can comment on what happens a signal, not when it happens. Consider a song as the signal under consideration. If you were not interested in time, the whole point of processing that signal would be lost. Rhythm and timing are the very heart of good music, after all. In this case, we want to know when the drums kicked i , as well as what notes were being played on the guitar. If we perform only frequency analysis, all time information would be lost and the only information we would have would be about what frequencies were played in the song, and what their respective amplitudes were, averaged over the duration of the entire song. So even if the drums stop playing after the second stanza, the frequency spectrum would show them playing throughout the song. Conversely, if we were only interested in the time information, we would be hardly better off than simply listening to the song.

The solution to this problem is essentially time-frequency analysis which is a field that deals with signal processing in both time and frequency domain. It consists of a collection of methods that allow us to make tradeoffs between time and frequency processing of a signal, depending on what makes more sense for a particular application, as we shall see through the rest of this tutorial.

2.2 The Heisenberg-Gabor Inequality

Before delving into joint time frequency representations, it is necessary to understand that any signal is characterized in the time-frequncy space by two quantities:

1. The *mean* position of the signal, defined as pair of two figures: average time (t_m) and average frequency (ν_m)
2. The energy localization of the signal in the time-frequency space, whose area is proportional to the *Time-Bandwidth product*. An important constraint related to this quantity is called the Heisenberg-Gabor inequality, which we shall explore later in this section.

If a signal $x(t)$ has finite energy, i.e.

$$E_x = \int_{-\infty}^{\infty} |x(t)|^2 \, dt < \infty$$

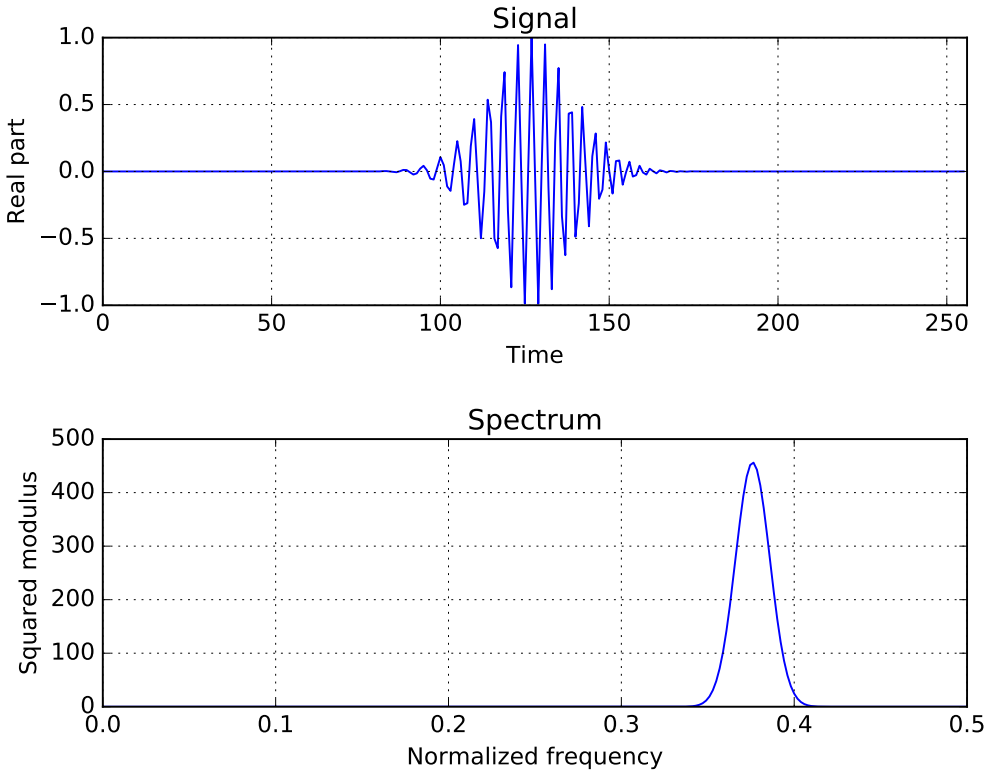
then the time and frequency domain energies of the signal can be considered as probability distributions, and their respective means and standard deviations can be used to estimate the time and frequency localizations and dispersions of the signal.

- Average time: $t_m = \frac{1}{E_x} \int_{-\infty}^{\infty} t \, |x(t)|^2 \, dt$
- Average frequency: $\nu_m = \frac{1}{E_x} \int_{-\infty}^{\infty} \nu \, |X(\nu)|^2 \, d\nu$
- Time spreading: $T^2 = \frac{4\pi}{E_x} \int_{-\infty}^{\infty} (t - t_m)^2 \, |x(t)|^2 \, dt$
- Frequency spreading: $B^2 = \frac{4\pi}{E_x} \int_{-\infty}^{\infty} (\nu - \nu_m)^2 \, |X(\nu)|^2 \, d\nu$

Let's play around with these values with some examples.

2.2.1 Example: Time and Frequency Localizations

Time and frequency localizations can be calculated with the functions `tftb.processing.loctime` and `tftb.processing.locfreq`. Consider a linear chirp with Gaussian amplitude modulation as an example, shown below:




```
>>> from tftb.generators import fmlin, amgauss
>>> from tftb.processing import loctime, locfreq
>>> sig = fmlin(256)[0] * amgauss(256)
>>> t_mean, time_spreading = loctime(sig)
>>> print t_mean, time_spreading
127.0 32.0
>>> f_mean, freq_spreading = locfreq(sig)
>>> print f_mean, freq_spreading
0.249019607843 0.0700964323482
```

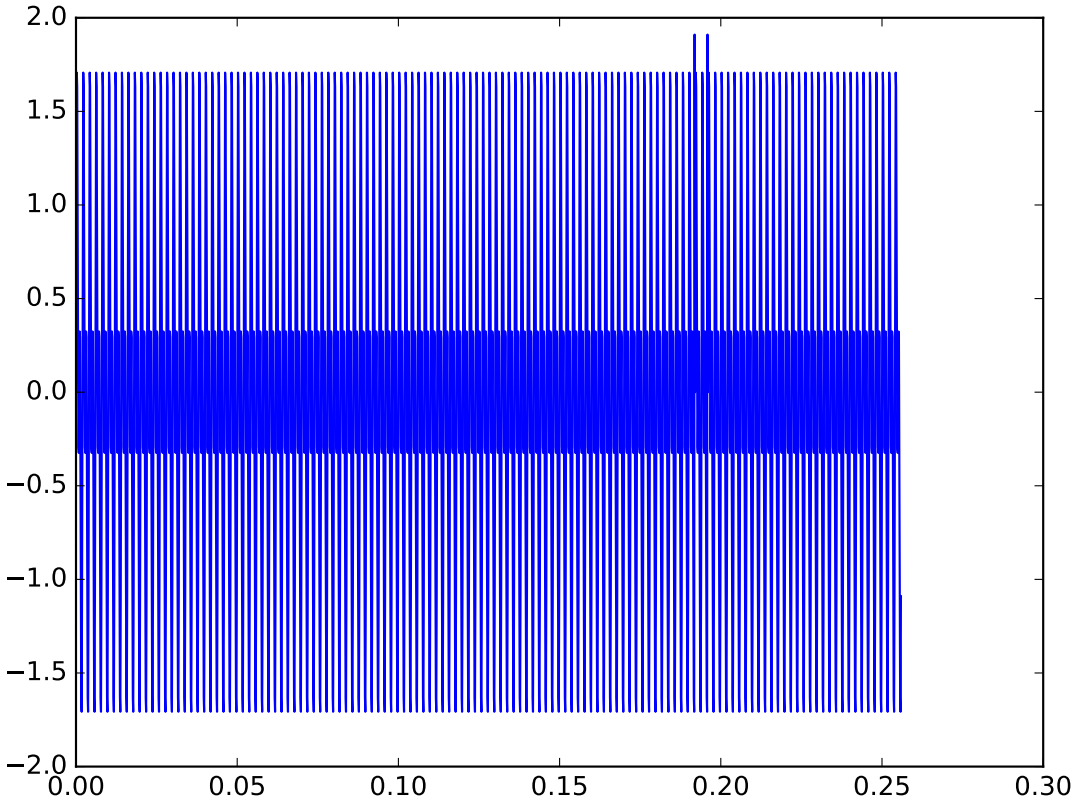
The time-bandwidth product of the signal can be obtained by multiplying the `time_spreading` and `frequency_spreading` variables in the snippet above. An important inequality concering the time bandwidth product is called the uncertainty principle. which states that a signal cannot be localized simultaneously in both time and frequency with arbitrariy high resolution. The next example demonstrates this concept.

2.2.2 Example: The Uncertainty Principle

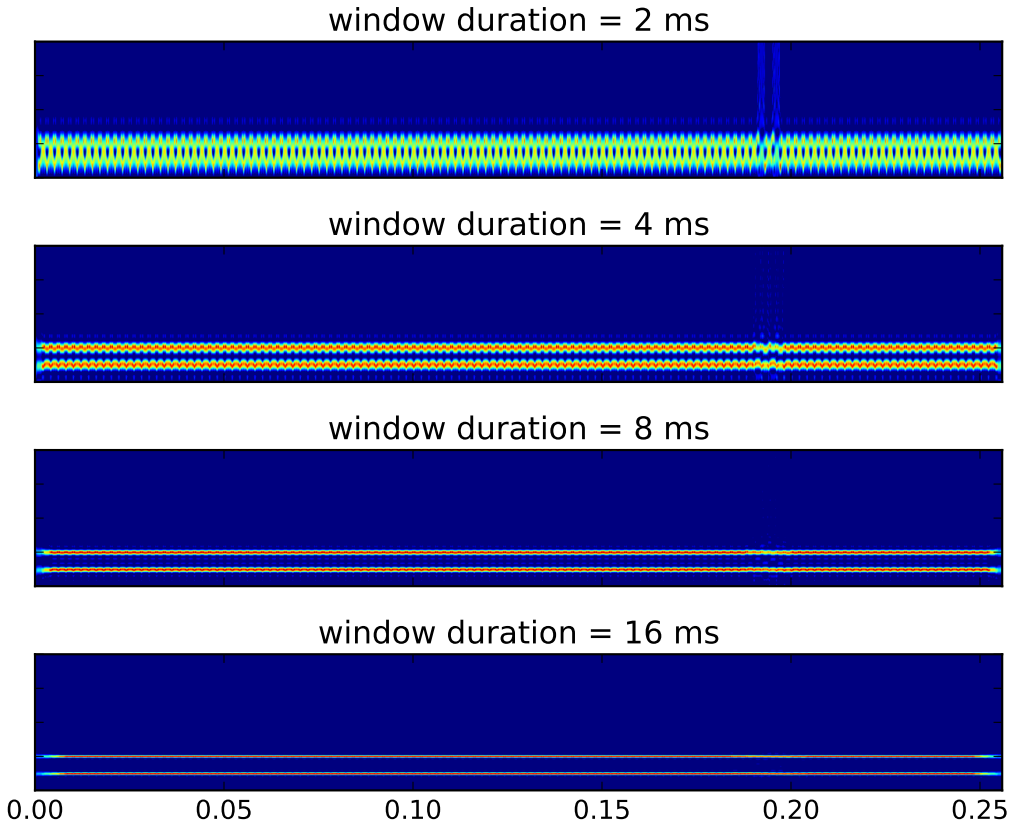
The uncertainty principle is a very manifest limitation of the Fourier transform. Consider the signal shown here:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> f1, f2 = 500, 1000
>>> t1, t2 = 0.192, 0.196
>>> f_sample = 8000
>>> n_points = 2048
>>> ts = np.arange(n_points, dtype=float) / f_sample
>>> signal = np.sin(2 * np.pi * f1 * ts) + np.sin(2 * np.pi * f2 * ts)
```

```
>>> signal[int(t1 * f_sample) - 1] += 3
>>> signal[int(t2 * f_sample) - 1] += 3
>>> plt.plot(ts, signal)
>>> plt.show()
```



It is a sum of two sinusiodal signals of frequencies 500 Hz and 1000 Hz. It has two spikes at $t = 0.192$s and $t = 0.196$s. The purpose of a time frequency distribution would be to clearly identify both the frequencies and both the spikes, thus resolving events in both frequency and time. Let's check out the spectrograms of of the signal with four different window lengths:



As can be clearly seen, resolution in time and frequency cannot be obtained simultaneously. In the last (bottom) image, where the window length is high, the STFT manages to discriminate between frequencies of 500 Hz and 1000 Hz very clearly, but the time resolution between the events at t = 0.192 s and t = 0.196 s is ambiguous. As we reduce the length of the window function, the resolution between the time events goes on becoming better, but only at the cost of resolution in frequencies.

Informally, the uncertainty principle states that arbitrarily high resolution cannot be obtained in both time and frequency. This is a consequence of the definition of the Fourier transform. The definition insists that a signal be represented as a weighted sum of sinusoids, and therefore identifies frequency information that is globally prevalent. As a workaround to this interpretation, we use the STFT which performs the Fourier transform on limited periods of the signals. Mathematically this uncertainty can be quantified with the Heisenberg-Gabor Inequality (also sometimes called the Gabor limit):

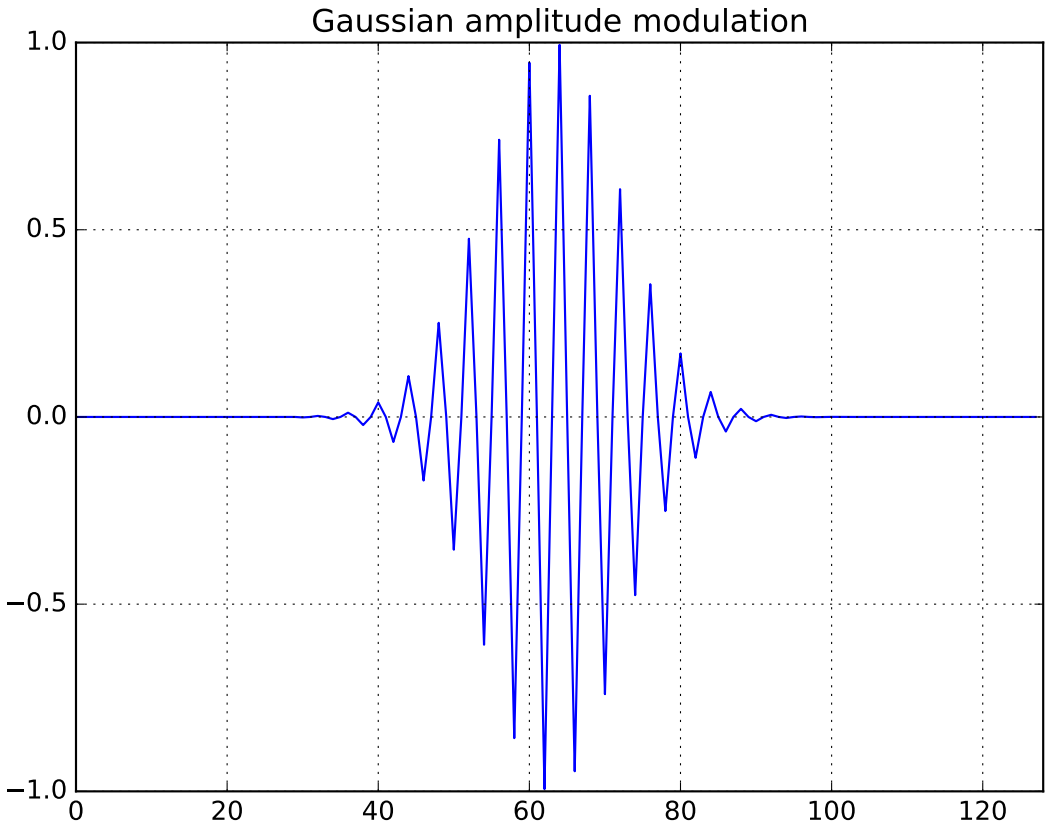
Heisenberg - Gabor Inequality

If T and B are standard deviations of the time characteristics and the bandwidth respectively of a signal $s(t)$, then

$$TB1$$

The expression states that the time-bandwidth product of a signal is lower bounded by unity. Gaussian functions satisfy the equality condition in the equation. This can be verified as follows:

```
>>> from tftb.generators import fmconst, amgauss
>>> x = gen.amgauss(128) * gen.fmconst(128)[0]
>>> plot(real(x))
```




```
>>> from tftb.processing import loctime, locfreq
>>> time_mean, time_duration = loctime(x)
>>> freq_center, bandwidth = locfreq(x)
>>> time_duration * bandwidth
1.0
```

A remarkably insightful commentary on the Uncertainty principle is provided in[1], which states that the Uncertainty principle is a statement about two variables whose associated operators do not mutually commute. This helps us apply the Uncertainty principle in signal processing in the same way as in quantum physics.

2.3 Instantaneous Frequency

An alternative way to localize a signal in time and frequency is its instantaneous frequency. Instantaneous frequencies are defined for analytic signals, which are defined as follows

$$x_a(t) = x(t) + jH(x(t))$$

[1] http://www.amazon.com/Time-Frequency-Analysis-Theory-Applications/dp/0135945321

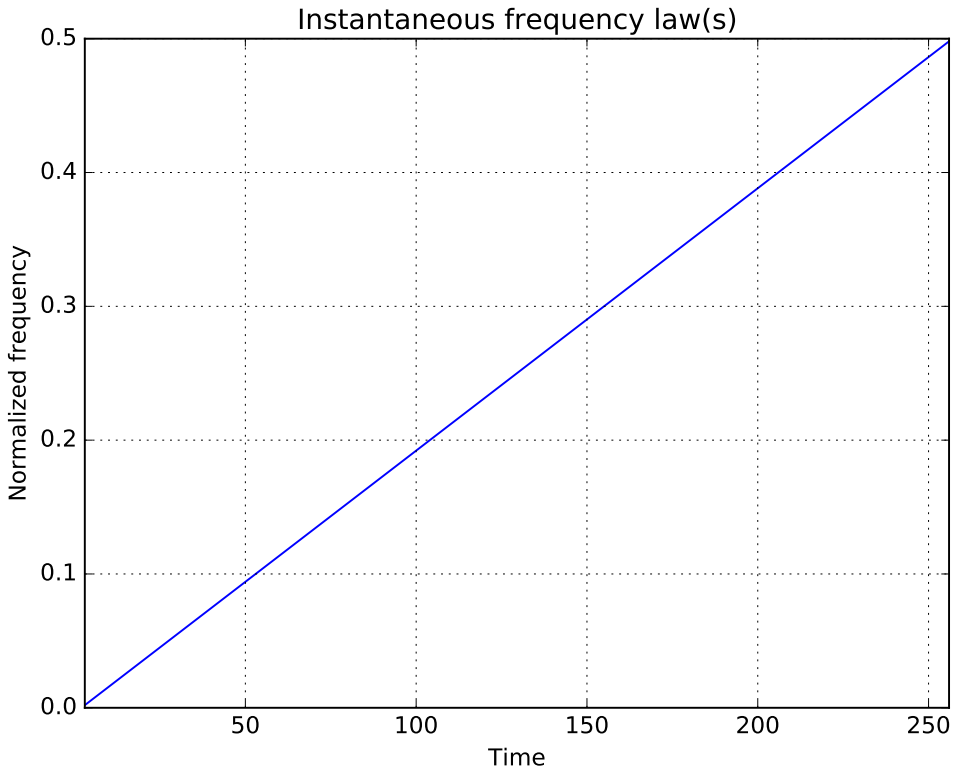
where $x(t)$ is a real valued time domain signal and H denotes the Hilbert transform (`scipy.signal.hilbert`). From this defition of the analytic signal, the following quantities can be derived:

- Instantaneous amplitude $a(t) = |x_a(t)|$
- Instantaneous frequency $f(t) = \frac{1}{2\pi}\frac{d}{dt}arg(x_a(t))$

An implementation of these functions can be found in `tftb.processing.instfreq`

2.3.1 Example: Instantaneous Frequency

```
>>> from tftb.processign import inst_freq, plotifl
>>> signal, _ = fmlin(256)
>>> time_samples = np.arange(3, 257)
>>> ifr = inst_freq(signal)[0]
>>> plotifl(time_samples, ifr)
```



2.4 Group Delay

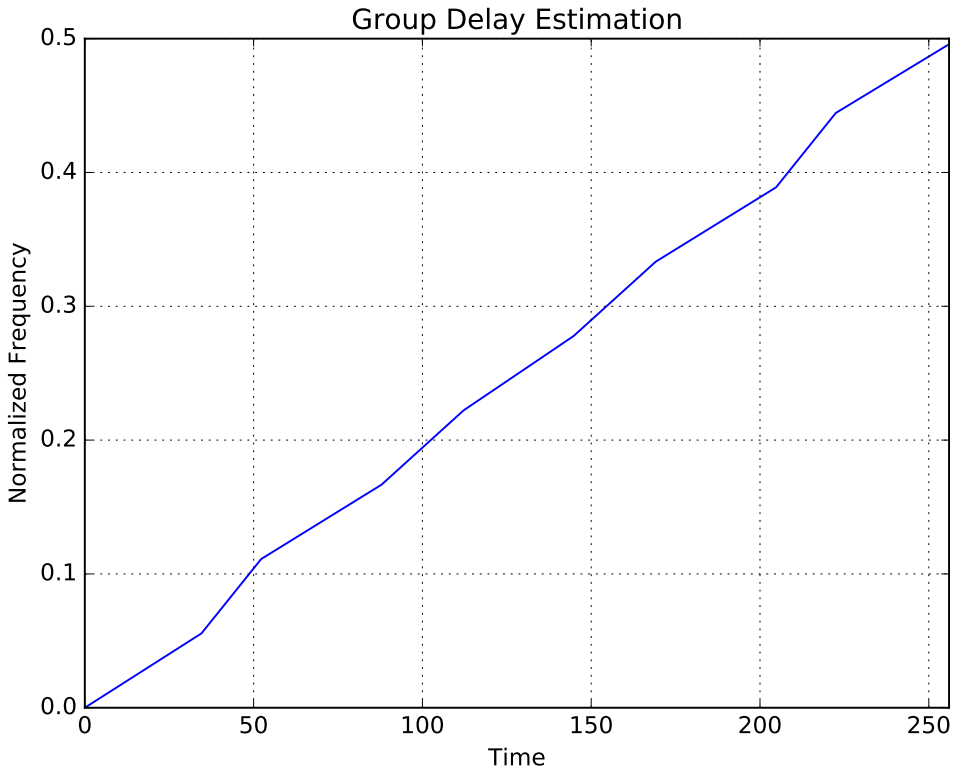
The frequency domain equivalent of instantaneous frequency is called group delay, which localizes time characteristics of a signal as function of the frequency.

$$t_x(\nu) = -\frac{1}{2\pi}\frac{d}{d\nu}arg(X_a(\nu))$$

2.4.1 Example: Group Delay

The group delay of the signal in the previous example can be obtained as follows

```
>>> from tftb.processign import group_delay
>>> fnorm = np.linspace(0, .5, 10)
>>> gd = group_delay(signal, fnorm)
>>> plt.plot(gd, fnorm)
```

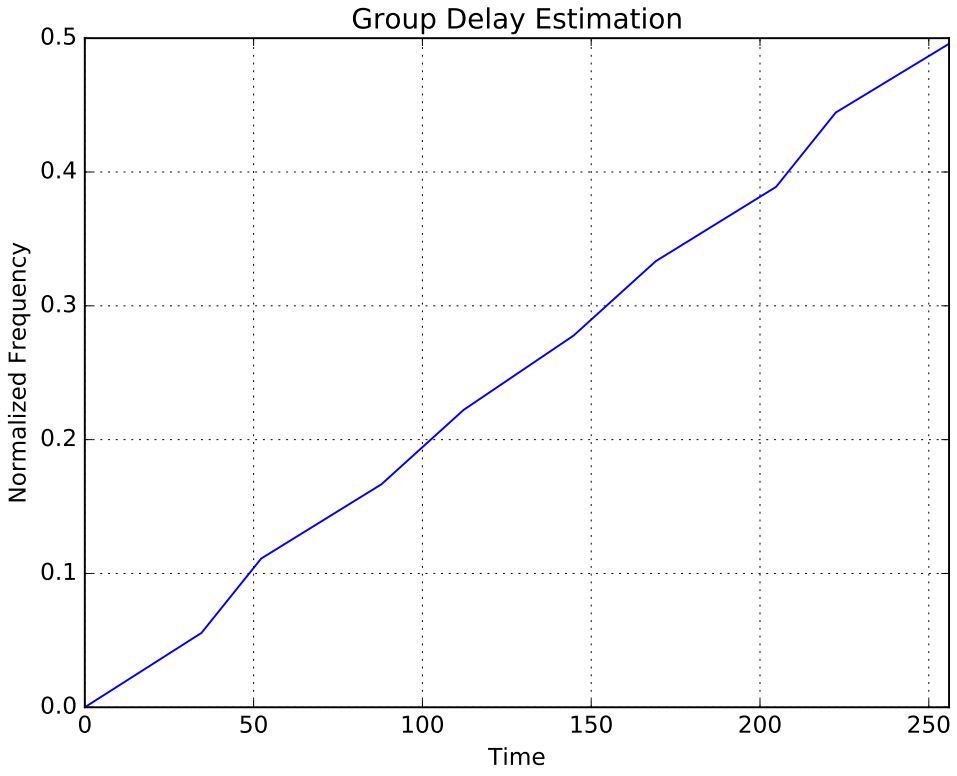


2.4.2 Example: Comparison of Instantaneous Frequency and Group Delay

Ideally, for a signal localized perfectly in time and frequency, its instantaneous frequency and group delay would be expected to be identical. However, mathematically they are two different fuctions in the time-frequency space, and only coincide for signals with high time-bandwidth products. This makes sense, since a high time-bandwidth product implies that the signal would be pushed away from the Heisenberg-Gabor inequality, thereby leading to lesser ambiguity. Consider the following example, where we construct two signals - one with a high time-bandwidth product, and one with a low one - and then estimate their respective instantaneous frequencies and group delays.

```
>>> # generate a signal with a high TB
>>> time_instants = np.arange(2, 256)
>>> sig1 = amgauss(256, 128, 90) * fmlin(256)[0]
>>> tm, T1 = loctime(sig1)
>>> fm, B1 = locfreq(sig1)
>>> print T1 * B1
15.9138
>>> ifr1 = inst_freq(sig1, time_instants)[0]
>>> f1 = np.linspace(0, 0.5 - 1.0 / 256, 256)
```

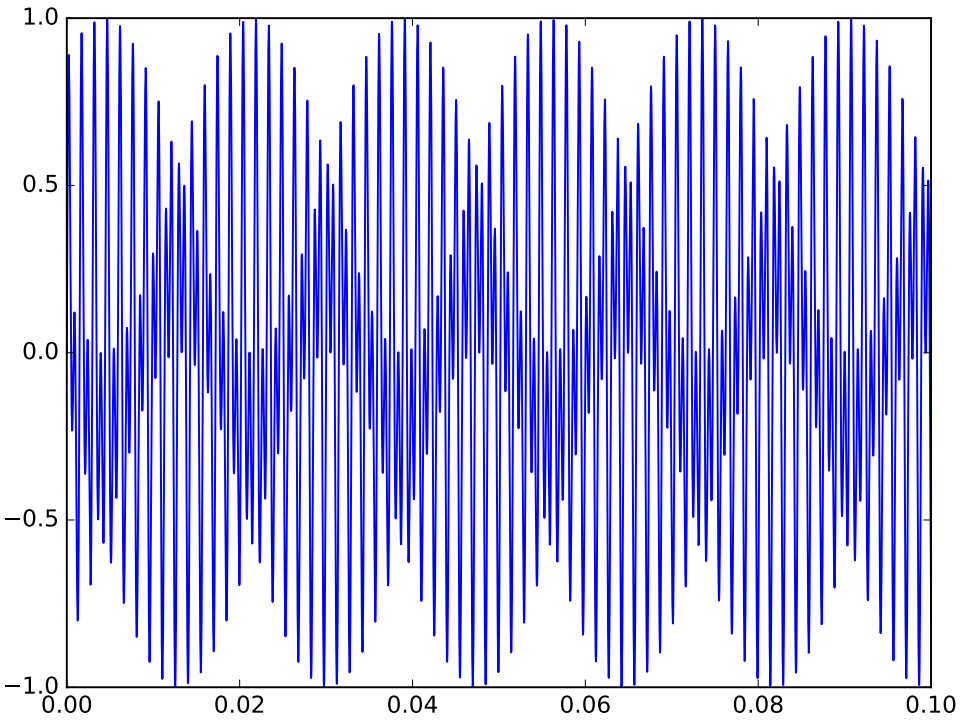
```
>>> gd1 = group_delay(sig1, f1)
>>>
>>> plt.subplot(211)
>>> plt.plot(time_instants, ifr1, '*', label='inst_freq')
>>> plt.plot(gd1, f1, '-', label='group delay')
>>>
>>> # generate a signal with low TB
>>> sig2 = amgauss(256, 128, 30) * fmlin(256, 0.2, 0.4)[0]
>>> tm, T2 = loctime(sig2)
>>> fm, B2 = locfreq(sig2)
>>> print T2 * B2
1.224
>>> ifr2 = inst_freq(sig2, time_instants)[0]
>>> f2 = np.linspace(0.02, 0.4, 256)
>>> gd2 = group_delay(sig2, f2)
>>>
>>> plt.subplot(212)
>>> plt.plot(time_instants, ifr2, '*', label='inst_freq')
>>> plt.plot(gd2, f2, '-', label='group delay')
```



2.5 A Note on Stationarity

As per Wikipedia, a "stationary" process is one whose joint probability distribution does not change with time (or space). Let's try and see what a stationary process looks like. Consider a signal generated as follows:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> fs = 32768
>>> ts = np.linspace(0, 1, fs)
>>> y1 = np.sin(2 * np.pi * 697 * ts)
>>> y2 = np.sin(2 * np.pi * 1336 * ts)
>>> y = (y1 + y2) / 2
>>> plt.plot(ts, y)
>>> plt.xlim(0, 0.1)
>>> plt.show()
```



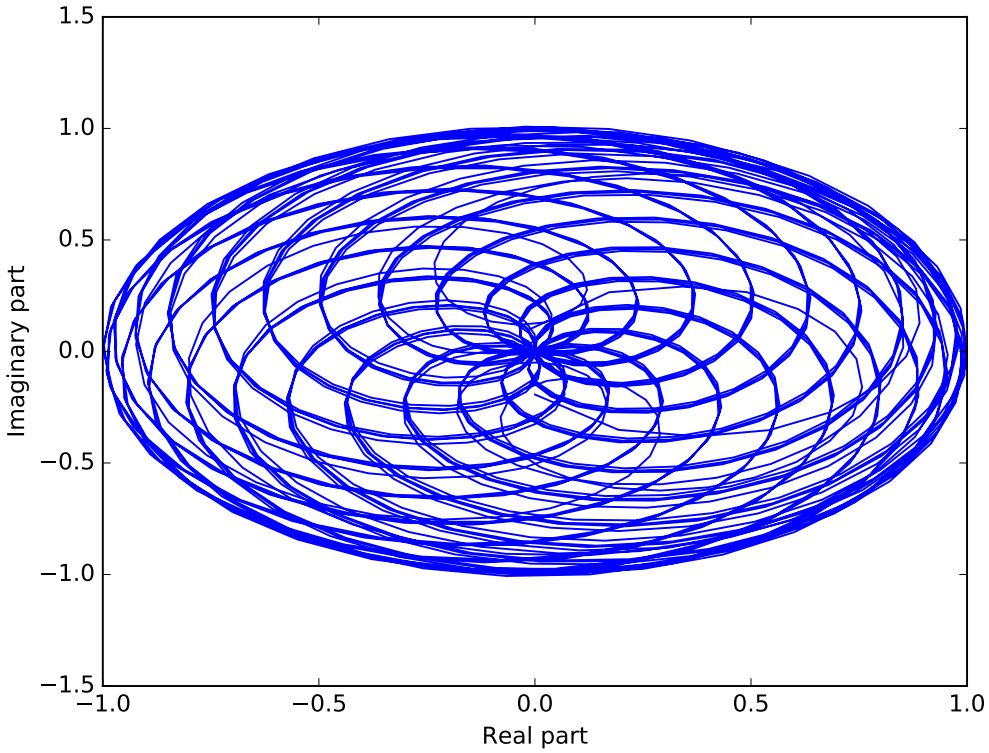
The plot shows a slice of a touchtone signal with the duration of a tenth of a second. This is the signal you hear when you press the "2" key on a telephone dialing pad. (You can save the generated signal as a WAV file as follows:

```
>>> from scipy.io import wavfile
>>> wavfile.write("tone.wav", fs, y)
```

and listen to the file `tone.wav` with your favourite music player.)

Since the signal is composed of two sinusoids, `y1` and `y2`, we would expect it to be stationary. Let's try and assert this qualitatively. Let's try to plot the signal in its phase space. In order to do so, we will first need to construct an analytic representation of the signal. For simplicity, we shall only consider a part of the original signal

```
>>> y = y[:(fs / 16)]
>>> y_analytic = hilbert(y)
>>> plt.plot(np.real(y_analytic), np.imag(y_analytic))
>>> plt.xlabel("Real part")
>>> plt.ylabel("Imaginary part")
>>> plt.show()
```

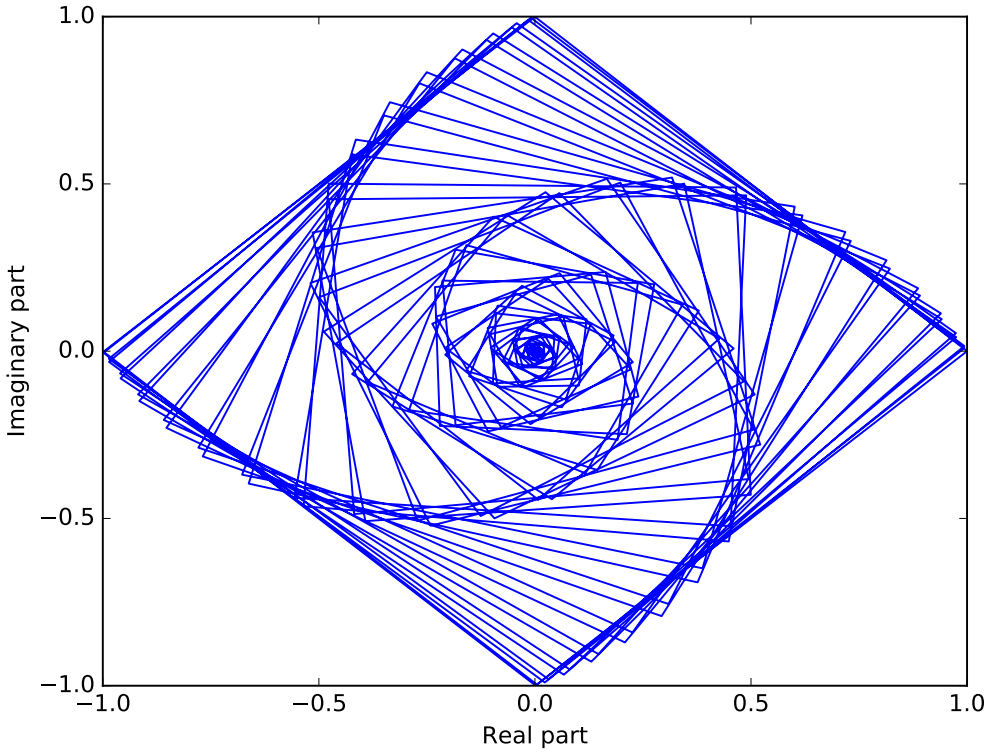


This visualization can be interpreted as follows. Imagine that there is a vector centered at the origin of this plot which traces out the signal as it rotates about the origin. Then, at any time t, the angle which the vector makes with the real axis is the instantaneous phase of the signal, $\theta(t)$. The angular speed with which the phasor rotates is the instantaneous frequency of the signal:

$$\omega(t) = \frac{d}{dt}\theta(t)$$

Now, let's compare this phase plot with that of a known nonstationary signal.

```
>>> from tftb.generators import fmlin, amgauss
>>> y_ns, _ = fmlin(2048)  # Already analytic, no need of Hilbert transorm
>>> y_nonstat = y_ns * amgauss(2048)
>>> plt.plot(np.real(y_nonstat), np.imag(y_nonstat))
>>> plt.xlabel("Real part")
>>> plt.ylabel("Imaginary part")
>>> plt.show()
```



Notice that the second plot has a lot of rough edges and sharp angles. This means that when the signal vector rotates through the phase space, it will have to make sharp jumps in order to trace the signal. Moreover, by the definition of instantaneous frequency, when the instantaneous phase is not differentiable, the instantaneous frequency will be indeterminate. By contrast, the phase plot for the stationary signal is a lot smoother, and we can expect that the instantaneous frequency will be finite at all times. Physically, this means that in the nonstationary signal, the variation in frequency components has no structure, and these components can change arbitrarily.

This phenomenon of arbitrary, unstructured changes in frequency over time is a symptom of nonstationarity, and will become increasingly relevant as we proceed.

CHAPTER 3

Gallery of PyTFTB Examples

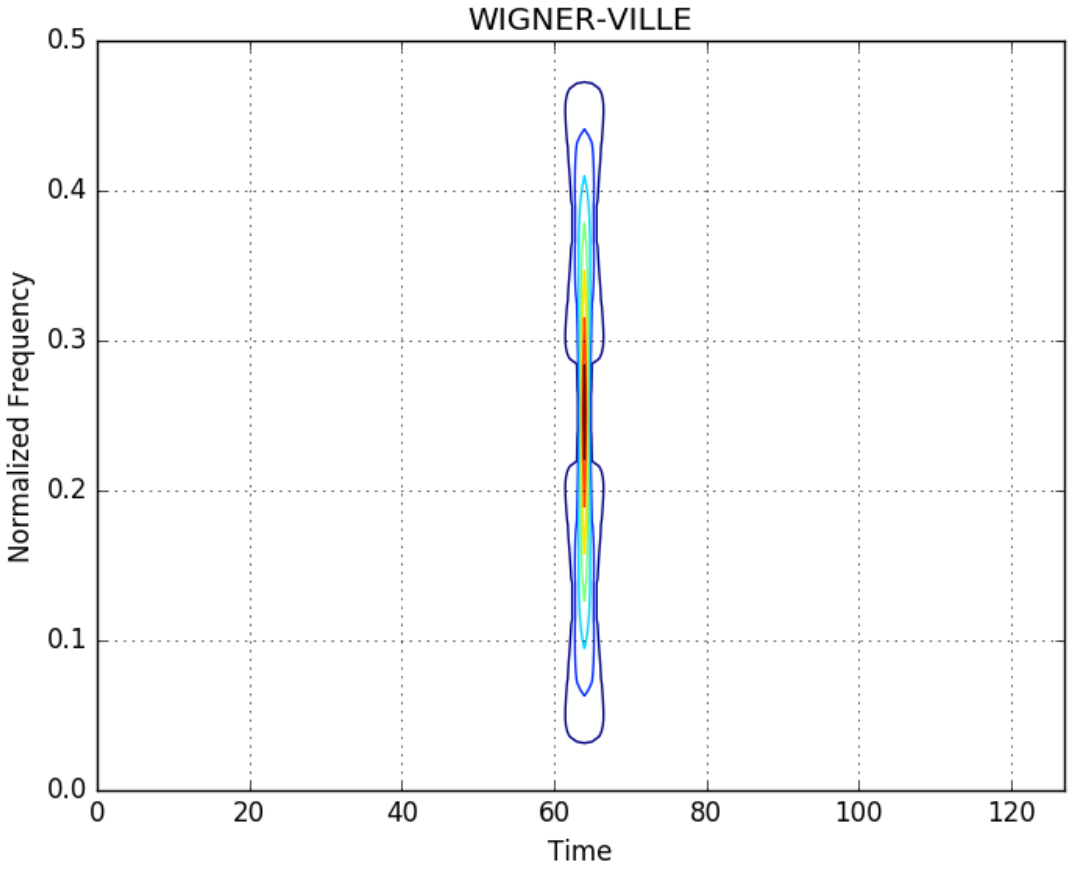
3.1 General examples

General-purpose and introductory examples for PyTFTB

3.1.1 Wigner-Ville Distribution of a Dirac Impulse Function

This example demonstrates the Wigner-Ville distribution of a Dirac impulse function, and shows the limitations of the WV distribution when applied to broadband signals.

Figure 4.24 from the tutorial.



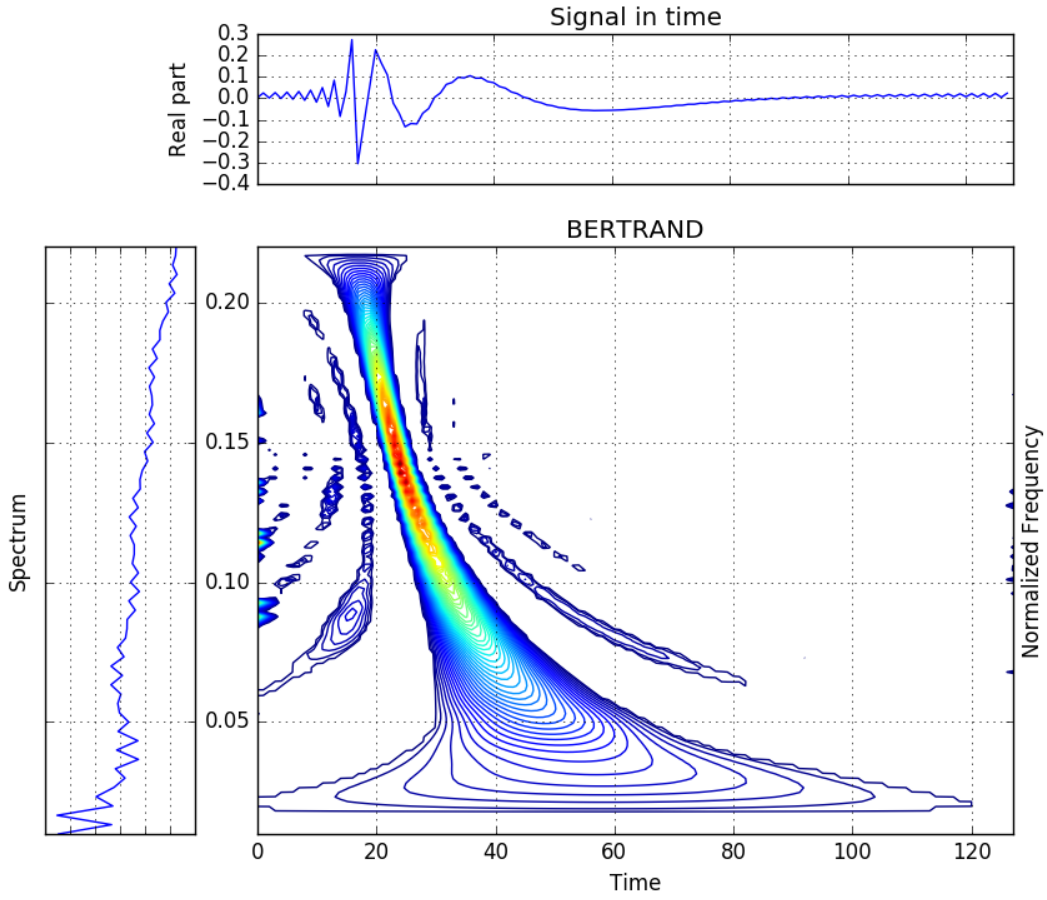

```
from tftb.generators import anapulse
from tftb.processing import WignerVilleDistribution

sig = anapulse(128)
wvd = WignerVilleDistribution(sig)
wvd.run()
wvd.plot(kind="contour", scale="log")
```

Total running time of the script: (0 minutes 0.597 seconds)

3.1.2 Bertrand Distribution of a Hyperbolic Group Delay Signal

This example shows the Bertrand distribution of a signal with hyperbolic group delay. The distribution is well localized around the hyperbola, but not perfectly. The Bertrand distribution operates only on a part of the frequency range between two bounds f_{min} and f_{max}.

Figure 4.21 from the tutorial.



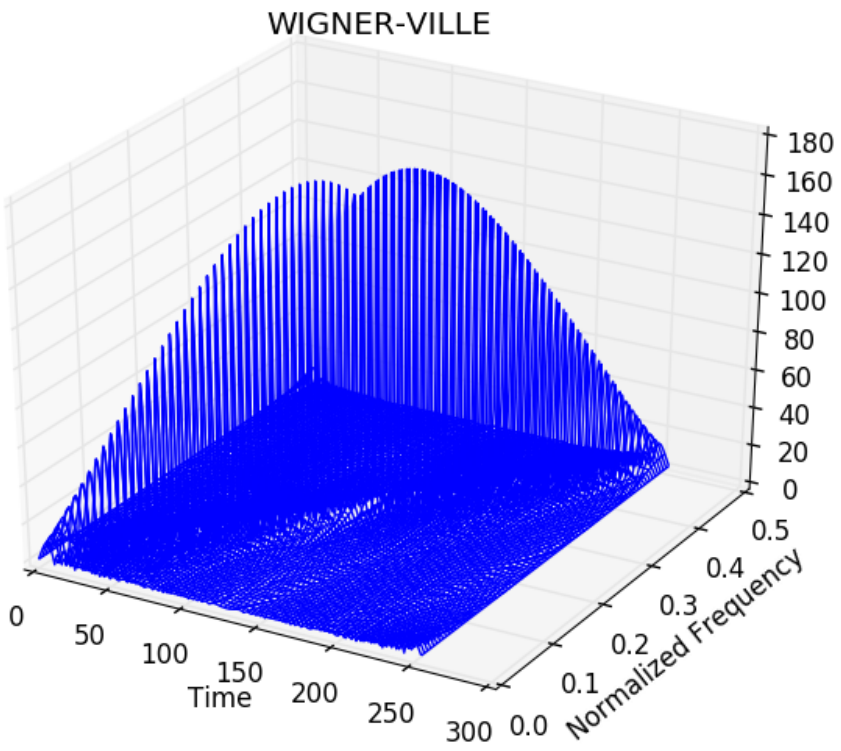

```
from tftb.processing.affine import BertrandDistribution
from tftb.generators import gdpower


sig = gdpower(128)[0]
bert = BertrandDistribution(sig, fmin=0.01, fmax=0.22)
bert.run()
bert.plot()
```

Total running time of the script: (0 minutes 0.428 seconds)

3.1.3 Wigner-Ville distribution of a Chirp

This example shows the wireframe plot of the Wigner-Ville distribution of a chirp. The WV distribution can take negative values, and has almost perfect localization in the time-frequency plane.

Figure 4.1 from the tutorial.



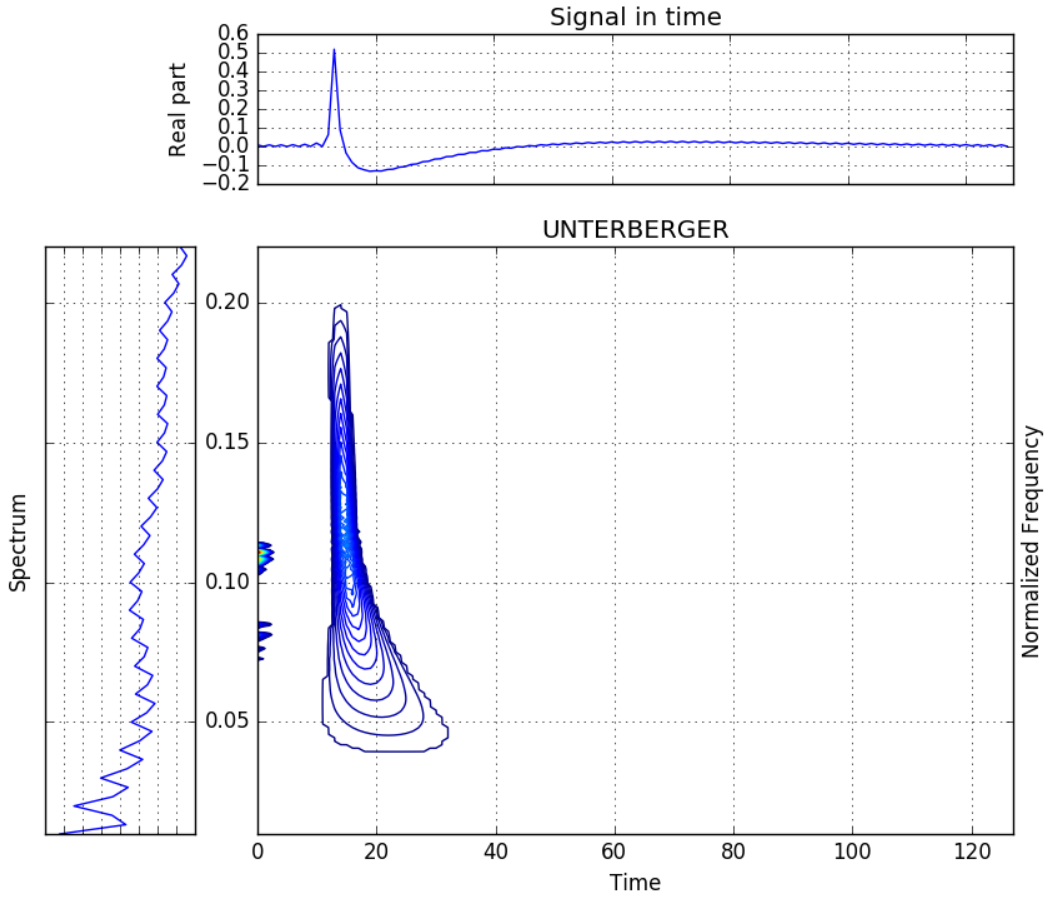

```
from tftb.generators import fmlin
from tftb.processing import WignerVilleDistribution

sig = fmlin(256)[0]
tfr = WignerVilleDistribution(sig)
tfr.run()
tfr.plot(threshold=0.0, kind='wireframe')
```

Total running time of the script: (0 minutes 0.228 seconds)

3.1.4 Unterberger distribution of a hyperbolic group delay signal

The active Unterberger distribution is the only localized bi-frequency kernel distribution which localizes perfectly signals having a group delay in $1/\nu^2$

Figure 4.23 from the tutorial.



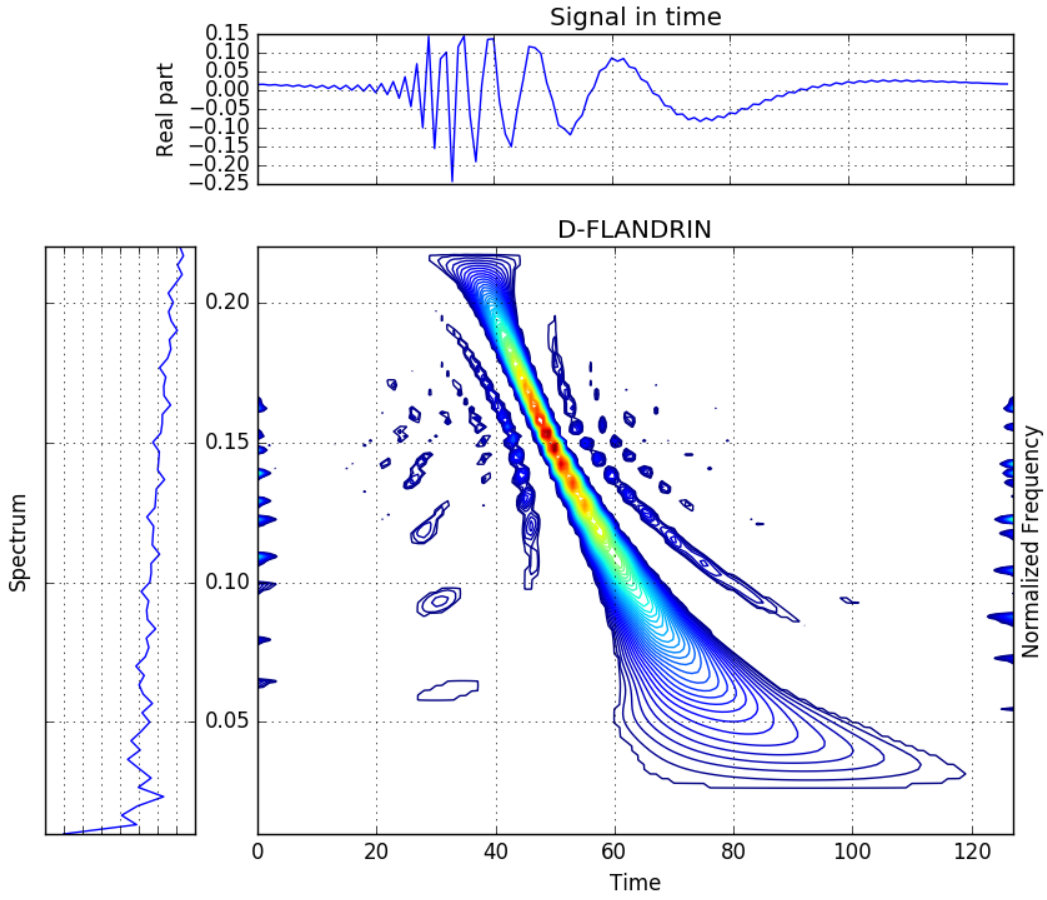

```
from tftb.processing import UnterbergerDistribution
from tftb.generators import gdpower

sig = gdpower(128, -1)[0]
dist = UnterbergerDistribution(sig, fmin=0.01, fmax=0.22, n_voices=172)
dist.run()
dist.plot()
```

Total running time of the script: (0 minutes 0.493 seconds)

3.1.5 D-Flandrin Distribution of a Hyperbolic Group Delay Signal

This example shows the D-Flandrin distribution of a signal having hyperbolic group delay. This is the only type of distribution that almost perfectly localizes signals having a group delay in $1/\sqrt{\nu}$

Figure 4.22 from the tutorial.



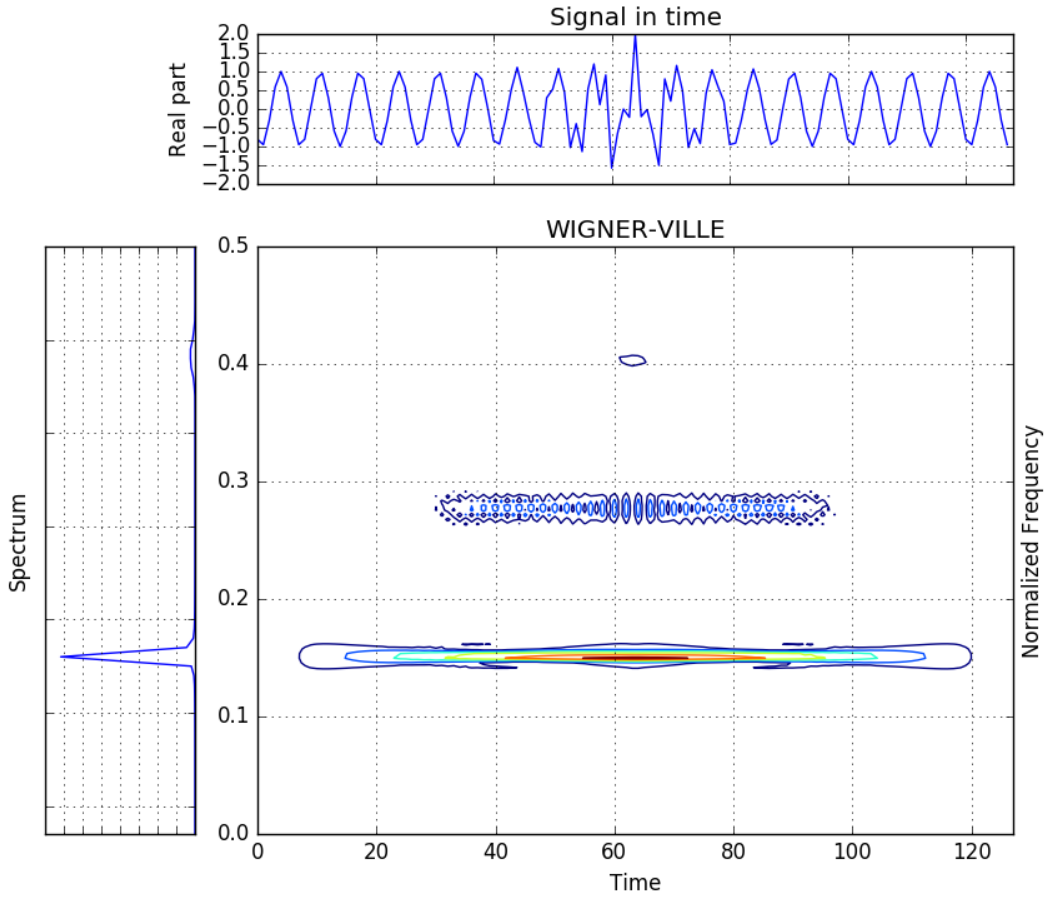

```
from tftb.processing import DFlandrinDistribution
from tftb.generators import gdpower

sig = gdpower(128, 1.0 / 2)[0]
spec = DFlandrinDistribution(sig, fmin=0.01, fmax=0.22, n_voices=128)
spec.run()
spec.plot()
```

Total running time of the script: (0 minutes 0.516 seconds)

3.1.6 Wigner Ville distribution of a Gaussian Atom and a Complex Sinusoid

This example demonstrates the Wigner Ville distribution of a signal composed from a Gaussian atom and a complex sinusoid with constant frequency modulation. Although the representation does isolate the atom and the sinusoid as independent phenomena in the signal, it also produces some interference between them.

Figure 4.8 from the tutorial.



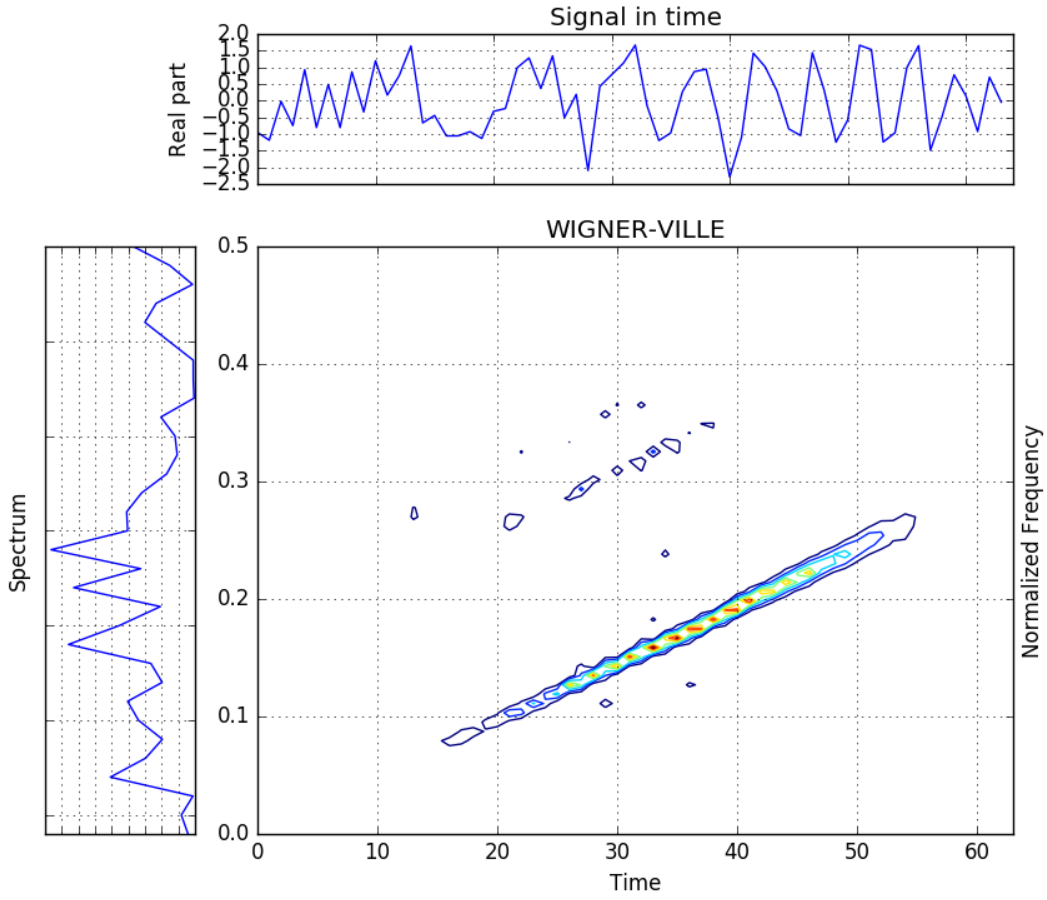

```
from tftb.generators import fmconst, amgauss
from tftb.processing import WignerVilleDistribution

sig = fmconst(128, 0.15)[0] + amgauss(128) * fmconst(128, 0.4)[0]
tfr = WignerVilleDistribution(sig)
tfr.run()
tfr.plot(show_tf=True, kind='contour')
```

Total running time of the script: (0 minutes 0.258 seconds)

3.1.7 Wigner-Ville Distribution of a Noisy Chirp

This example shows the Wigner-Ville distribution of a noisy chirp signal. The linear frequency increase is undetectable in the time domain, but a straight line can be seen in the distribution.

Figure 5.3 from the tutorial.



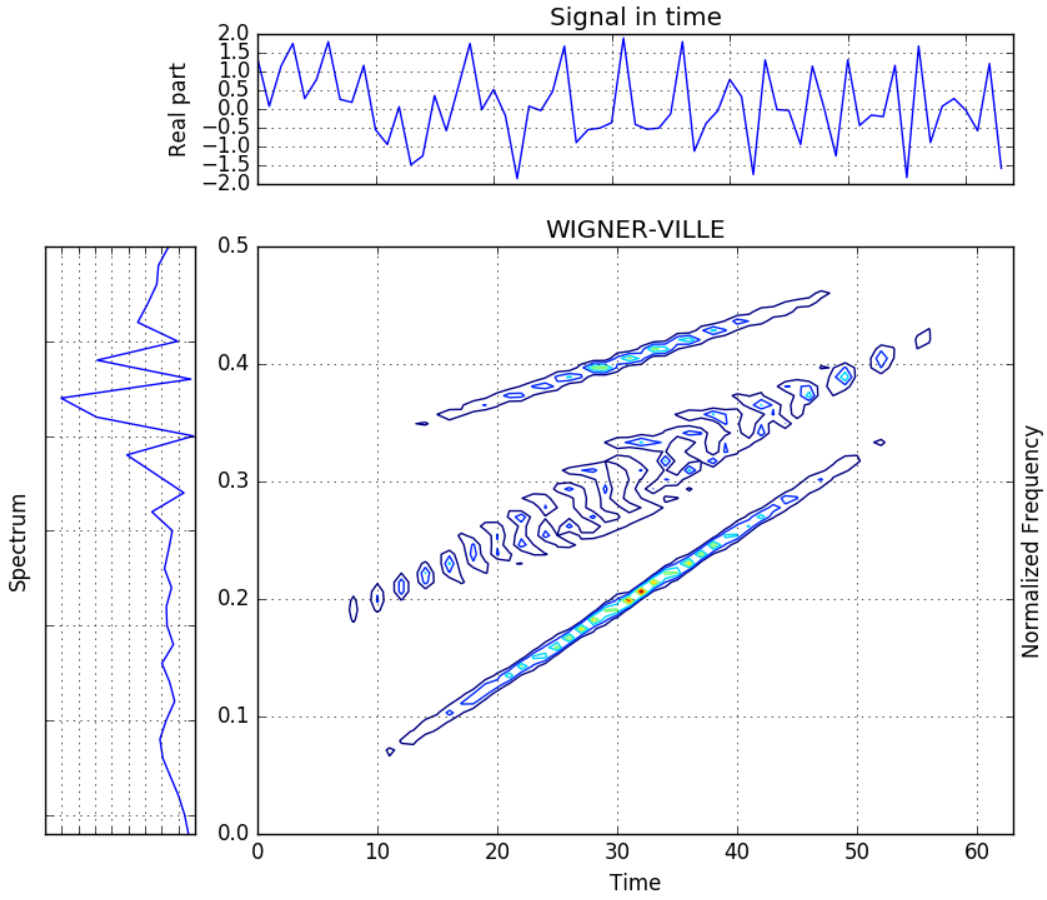

```
from tftb.generators import noisecg, sigmerge, fmlin
from tftb.processing.cohen import WignerVilleDistribution

N = 64
sig = sigmerge(fmlin(N, 0, 0.3)[0], noisecg(N), 1)
wvd = WignerVilleDistribution(sig)
wvd.run()
wvd.plot(kind='contour', show_tf=True, sqmod=True)
```

Total running time of the script: (0 minutes 0.336 seconds)

3.1.8 Wigner Ville Distribution of Two Simultaneous Chirps

The signal to be analyzed contains two linear frequency modulations, each with a different slope. Note the interference between them.

Figure 5.5 from the tutorial.



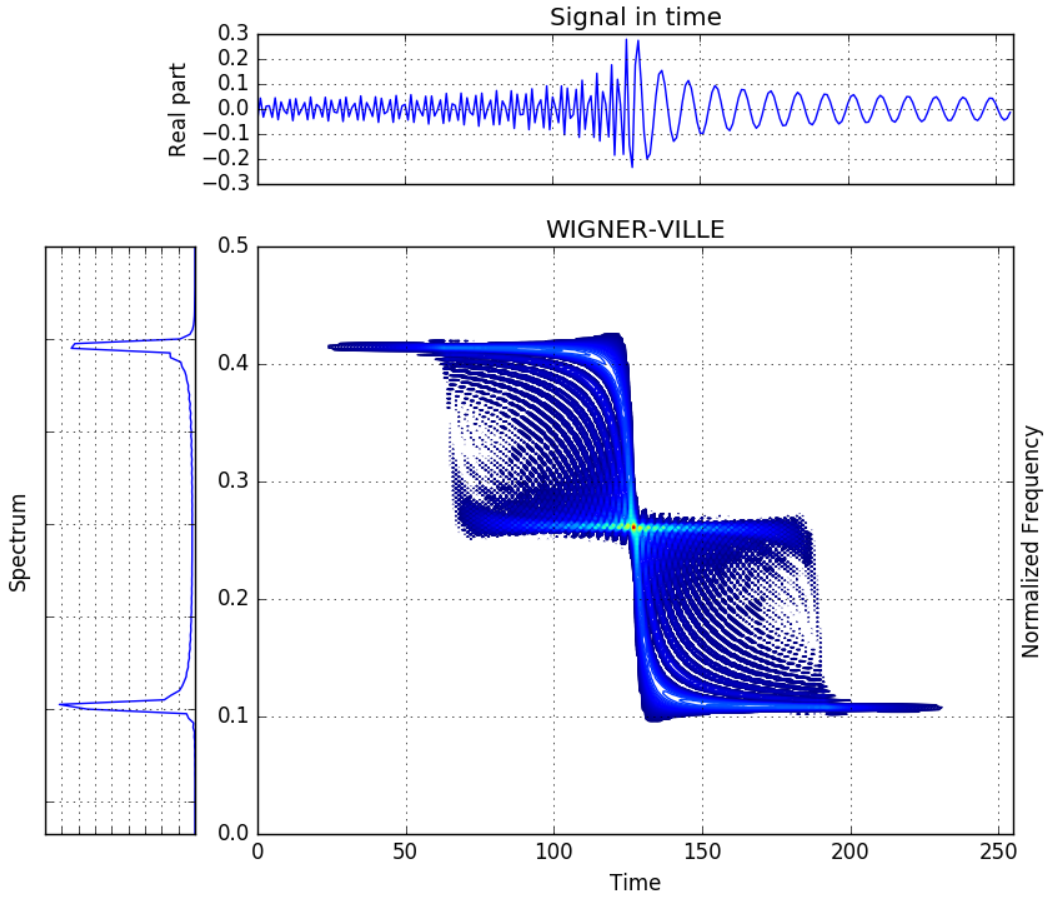

```
from tftb.generators import fmlin, sigmerge
from tftb.processing.cohen import WignerVilleDistribution

N = 64
sig = sigmerge(fmlin(N, 0, 0.4)[0], fmlin(N, 0.3, 0.5)[0], 1)
tfr = WignerVilleDistribution(sig)
tfr.run()
tfr.plot(kind='contour', sqmod=True, show_tf=True)
```

Total running time of the script: (0 minutes 0.240 seconds)

3.1.9 Wigner-Ville Distribution of a Doppler Signal

This example shows the Wigner-Ville distribution of a Doppler signal. The signal steadily rises and falls, but there are many interference terms present in the time-friequency plane, due to the bilinearity of the signal.

Figure 4.2 from the tutorial.



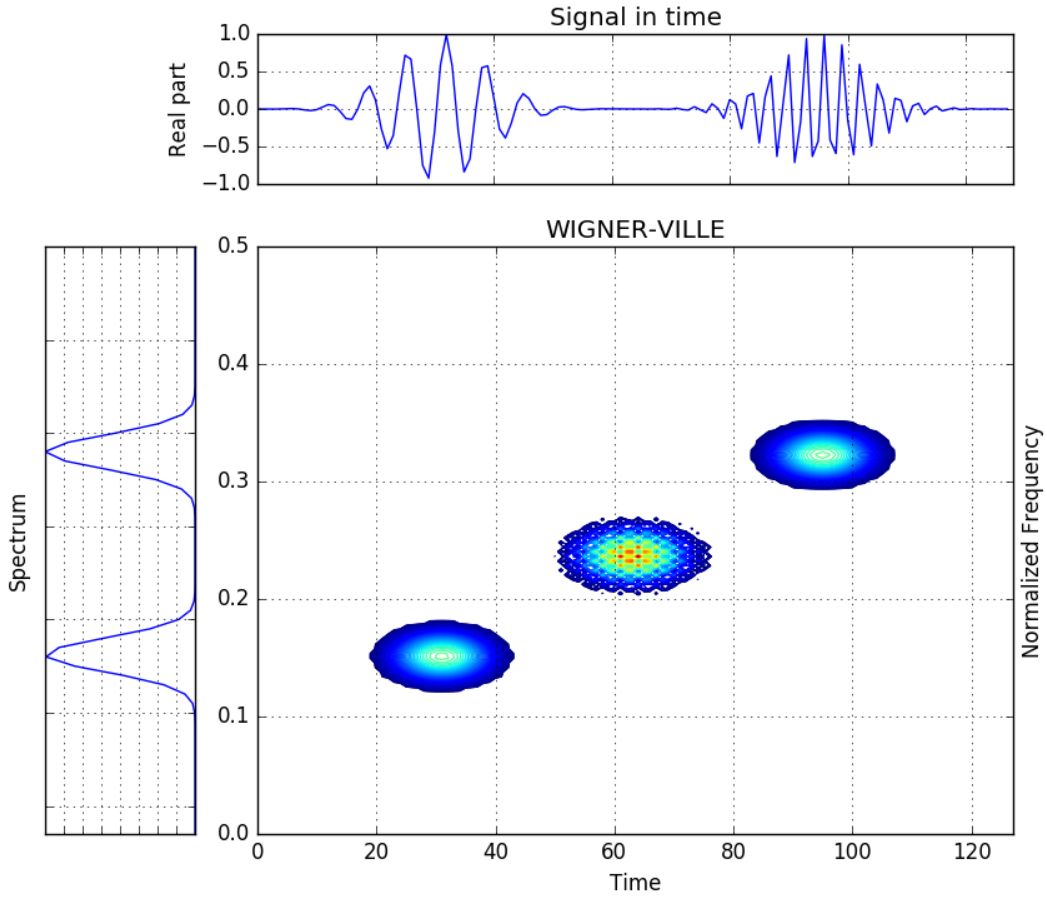

```
from tftb.generators import doppler
from tftb.processing import WignerVilleDistribution

fm, am, iflaw = doppler(256, 50.0, 13.0, 10.0, 200.0)
sig = am * fm
dist = WignerVilleDistribution(sig)
tfr, times, freqs = dist.run()
dist.plot(show_tf=True, kind="contour", scale="log")
```

Total running time of the script: (0 minutes 0.836 seconds)

3.1.10 Wigner Ville Distribution of Analytic Gaussian Atoms

This example shows the WVD of and analytic Gaussian atom. As seen in Figure 4.6, the WVD of a real valued signal may present interference terms and spectral aliasing. One of the ways to fix this is to use an analytic signal, which divides the spectral domain into two parts: real and imaginary. Thus, the number of interference terms is also halved. Secondly, analytic signals have no negative components, so the terms present in the negative half plane also vanish.

Figure 4.7 from the tutorial.



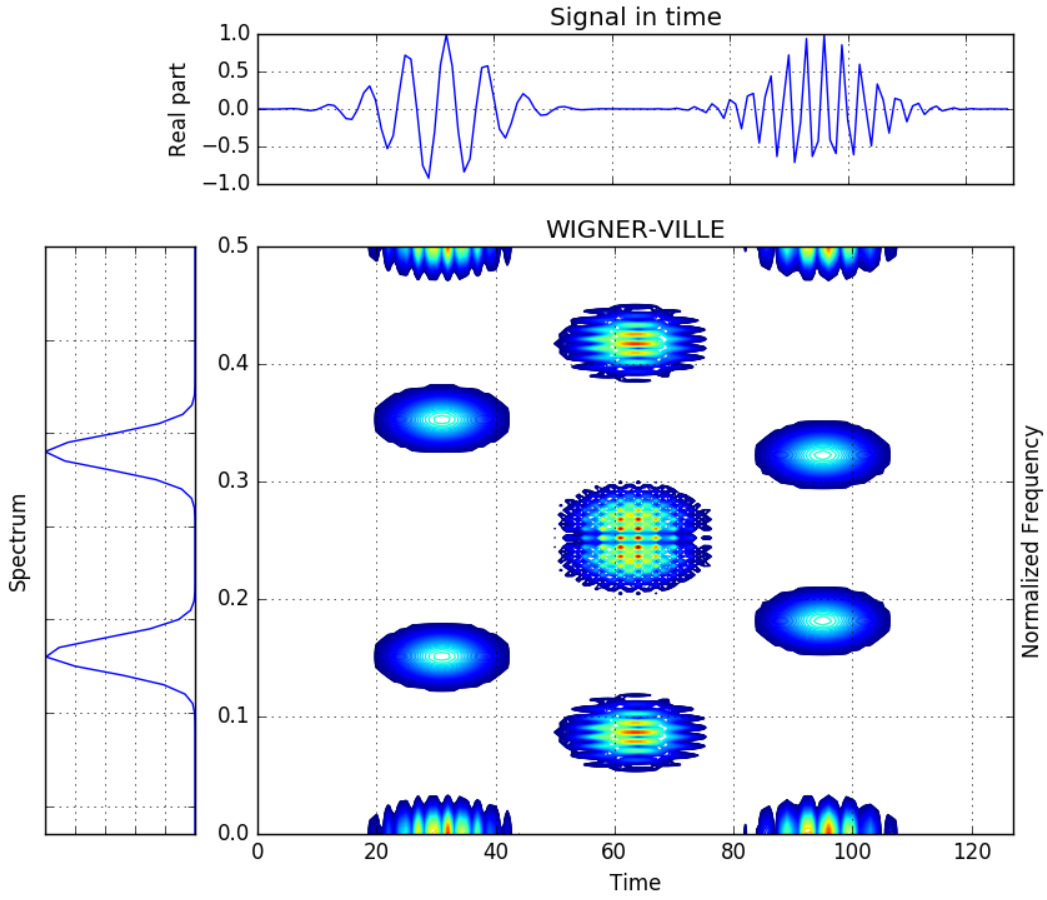

```
import numpy as np
from tftb.generators import atoms
from tftb.processing import WignerVilleDistribution

x = np.array([[32, .15, 20, 1],
              [96, .32, 20, 1]])
g = atoms(128, x)
spec = WignerVilleDistribution(g)
spec.run()
spec.plot(show_tf=True, kind="contour", scale="log")
```

Total running time of the script: (0 minutes 0.398 seconds)

3.1.11 Sampling Effects on the Wigner-Ville Distribution of a Real Valued Gaussian Atom

This example shows the Wigner-Ville distribution of a real valued Gaussian atom. If a signal is sampled at the Nyquist rate, the WVD is affected by spectral aliasing and many additional interferences. To fix this, either the signal may be oversampled, or an analytical signal may be used.

Figure 4.6 from the tutorial.



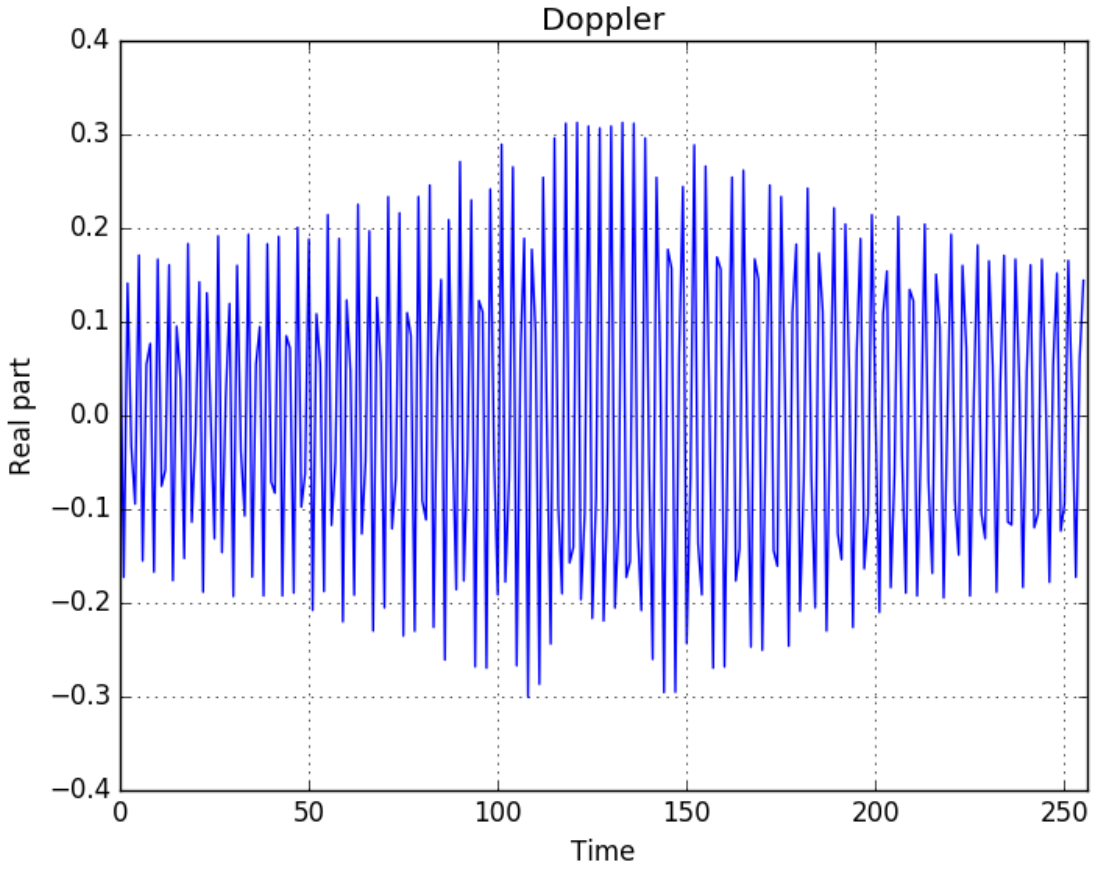

```
import numpy as np
from tftb.generators import atoms
from tftb.processing import WignerVilleDistribution

x = np.array([[32, .15, 20, 1],
              [96, .32, 20, 1]])
g = atoms(128, x)
spec = WignerVilleDistribution(np.real(g))
spec.run()
spec.plot(kind="contour", show_tf=True, scale="log")
```

Total running time of the script: (0 minutes 0.547 seconds)

3.1.12 Doppler Signal

Generate a Doppler Signal.

Figure 2.8 from the tutorial.

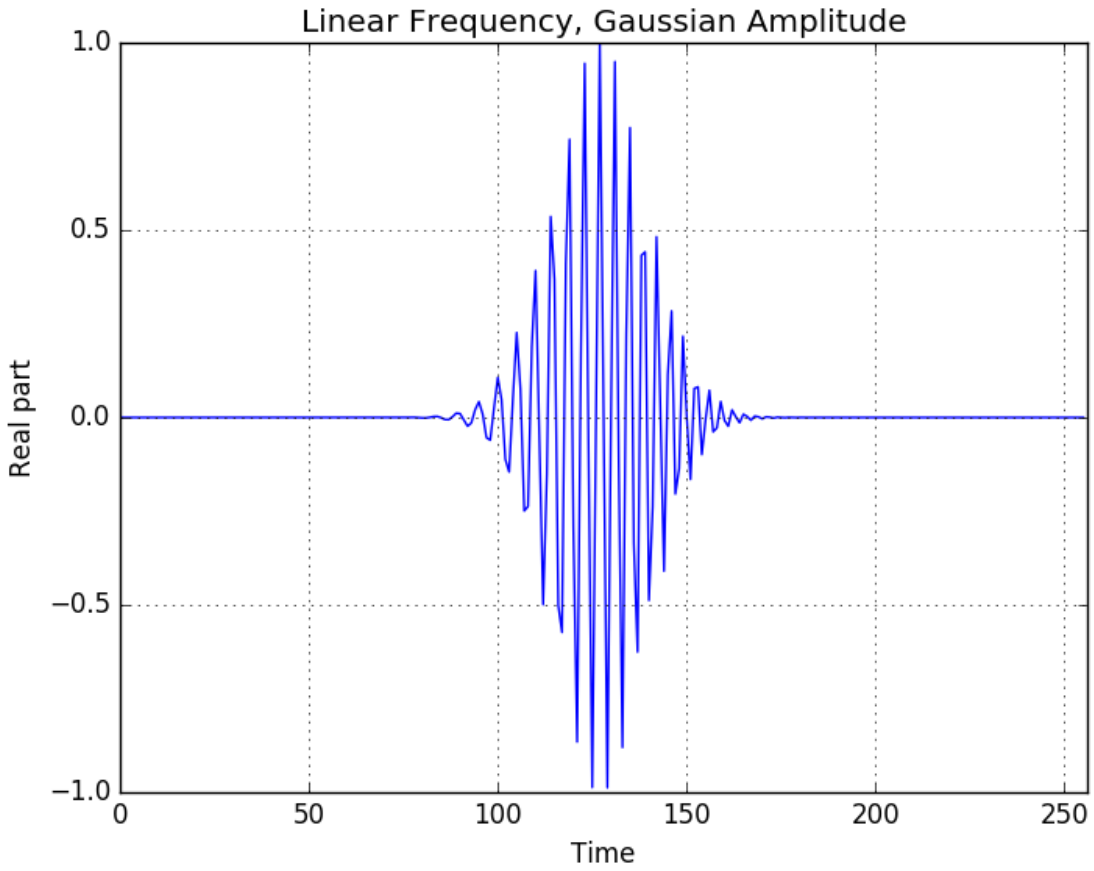



```
from tftb.generators import doppler
from numpy import real
import matplotlib.pyplot as plt

fm, am, _ = doppler(256.0, 200.0, 4000.0 / 60.0, 10.0, 50.0)
signal = am * fm

plt.plot(real(signal))
plt.xlabel('Time')
plt.ylabel('Real part')
plt.title('Doppler')
plt.xlim(0, 256)
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.072 seconds)

3.1.13 Linear Frequency and Gaussian Amplitude Modulation

Generate a mono-component nonstationary signal with linear frequency modulation and Gaussian amplitude modulation.



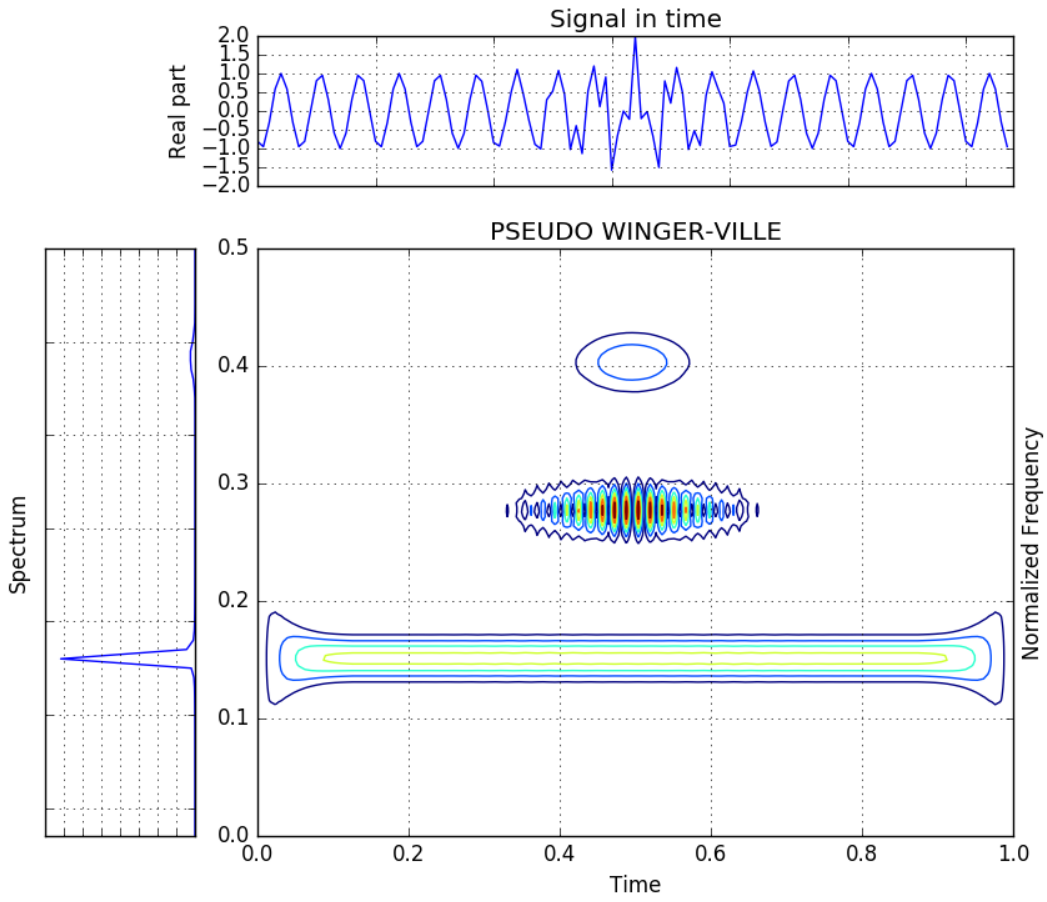

```
from tftb.generators import fmlin, amgauss
from numpy import real
import matplotlib.pyplot as plt


fm, _ = fmlin(256)
am = amgauss(256)
signal = fm * am
plt.plot(real(signal))
plt.xlabel('Time')
plt.ylabel('Real part')
plt.title('Linear Frequency, Gaussian Amplitude')
plt.xlim(0, 256)
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.058 seconds)

3.1.14 Pseudo-Wigner-Ville Distribution of a Gaussian Atom and a Complex Sinusoid

This example demonstrates the pseudo Wigner Ville distribution of a signal composed from a Gaussian atom and a complex sinusoid with constant frequency modulation. Note that the frequency resolution is relatively worse than that of the Wigner-Ville representation, and the interferences have not been resolved properly.

Figure 4.9 from the tutorial.



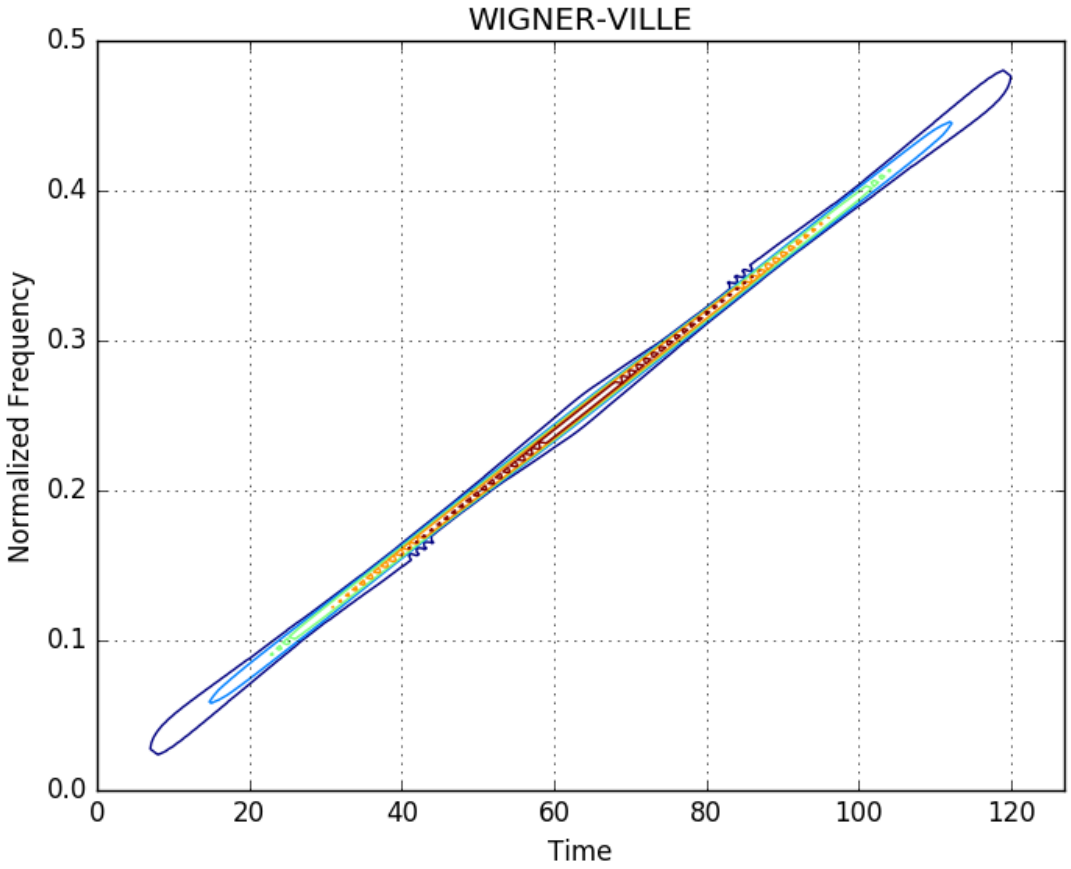

```
from tftb.generators import fmconst, amgauss
from tftb.processing import PseudoWignerVilleDistribution
import numpy as np

t = np.linspace(0, 1, 128)
sig = fmconst(128, 0.15)[0] + amgauss(128) * fmconst(128, 0.4)[0]
tfr = PseudoWignerVilleDistribution(sig, timestamps=t)
tfr.run()
tfr.plot(show_tf=True, kind="contour")
```

Total running time of the script: (0 minutes 0.272 seconds)

3.1.15 Wigner-Ville Distribution of a Chirp

Construct a chirp signal and visualize its Wigner-Ville distribution.

Figure 1.3 from the tutorial.



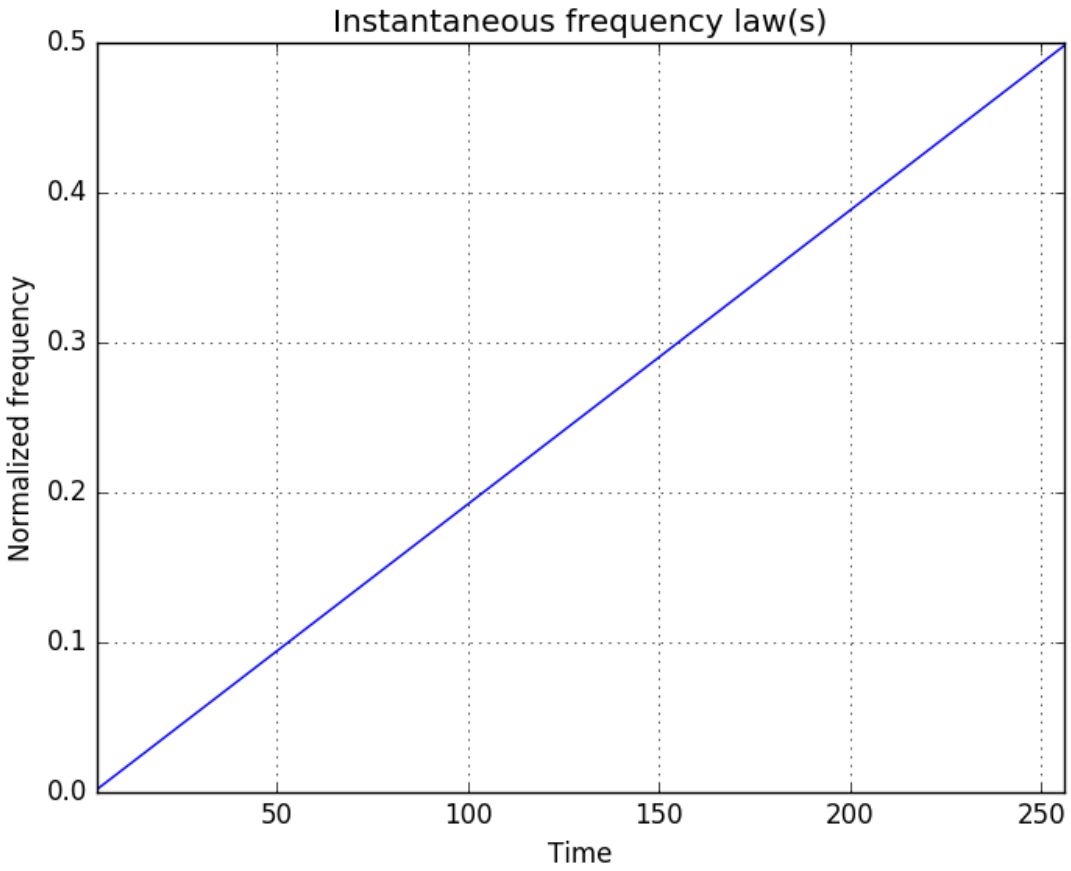

```
from tftb.generators import fmlin
from tftb.processing.cohen import WignerVilleDistribution

n_points = 128
fmin, fmax = 0.0, 0.5
signal, _ = fmlin(n_points, fmin, fmax)

# Wigner-Ville distribution of the chirp.

wvd = WignerVilleDistribution(signal)
wvd.run()
wvd.plot(kind='contour', extent=[0, n_points, fmin, fmax])
```

Total running time of the script: (0 minutes 0.077 seconds)

3.1.16 Estimate the Instantaneous Freuqncy of a Chirp

Construct a chirp and estimate its instantaneous frequency.

Figure 2.3 from the tutorial.



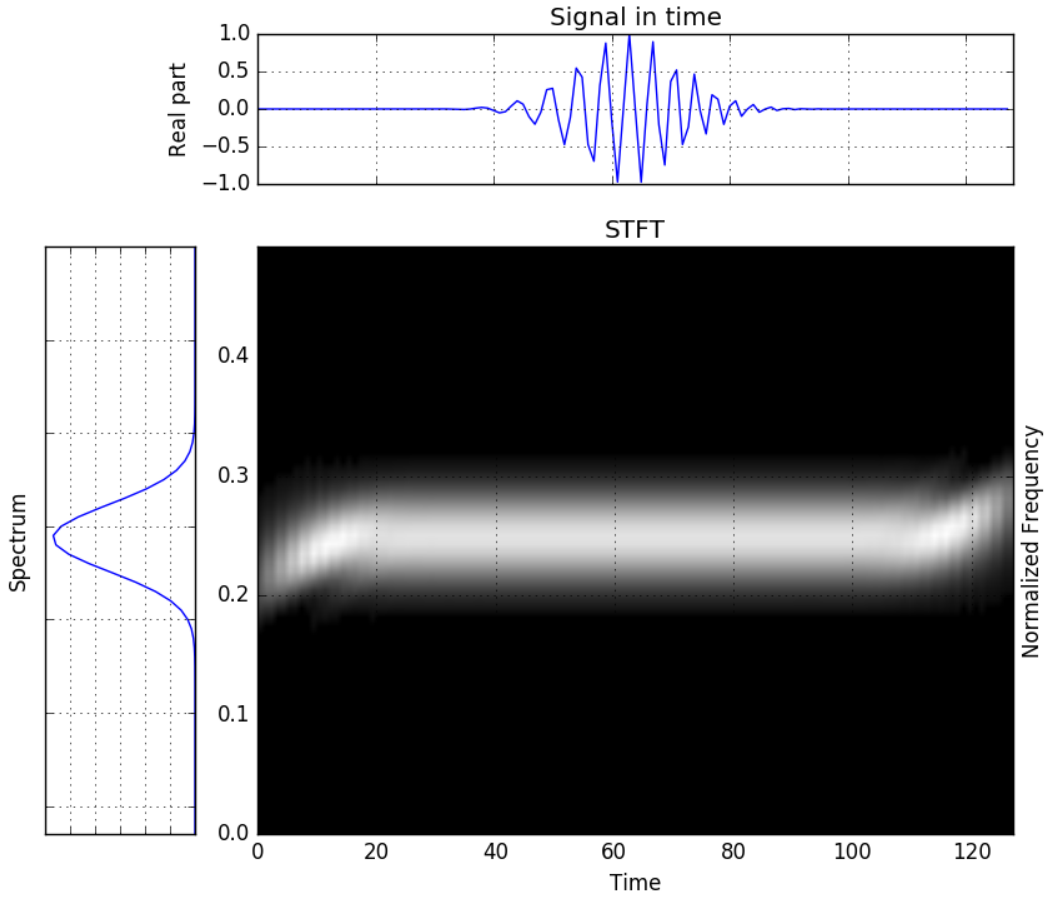

```
from tftb.generators import fmlin
from tftb.processing import plotifl, inst_freq
# TODO: There doesn't have to be something called `plotifl`. Put this into a
# separate visualization module.
import numpy as np


signal, _ = fmlin(256)
time_samples = np.arange(3, 257)
ifr = inst_freq(signal)[0]
plotifl(time_samples, ifr)
```

Total running time of the script: (0 minutes 0.066 seconds)

3.1.17 Time-frequency Resolution: Long Analysis Window

This example shows the effect of an analysis window which is long in time on the time-frequency resolution. Specifically, longer windows have good frequency resolutions but poor time resolutions.

Figure 3.6 from the tutorial.



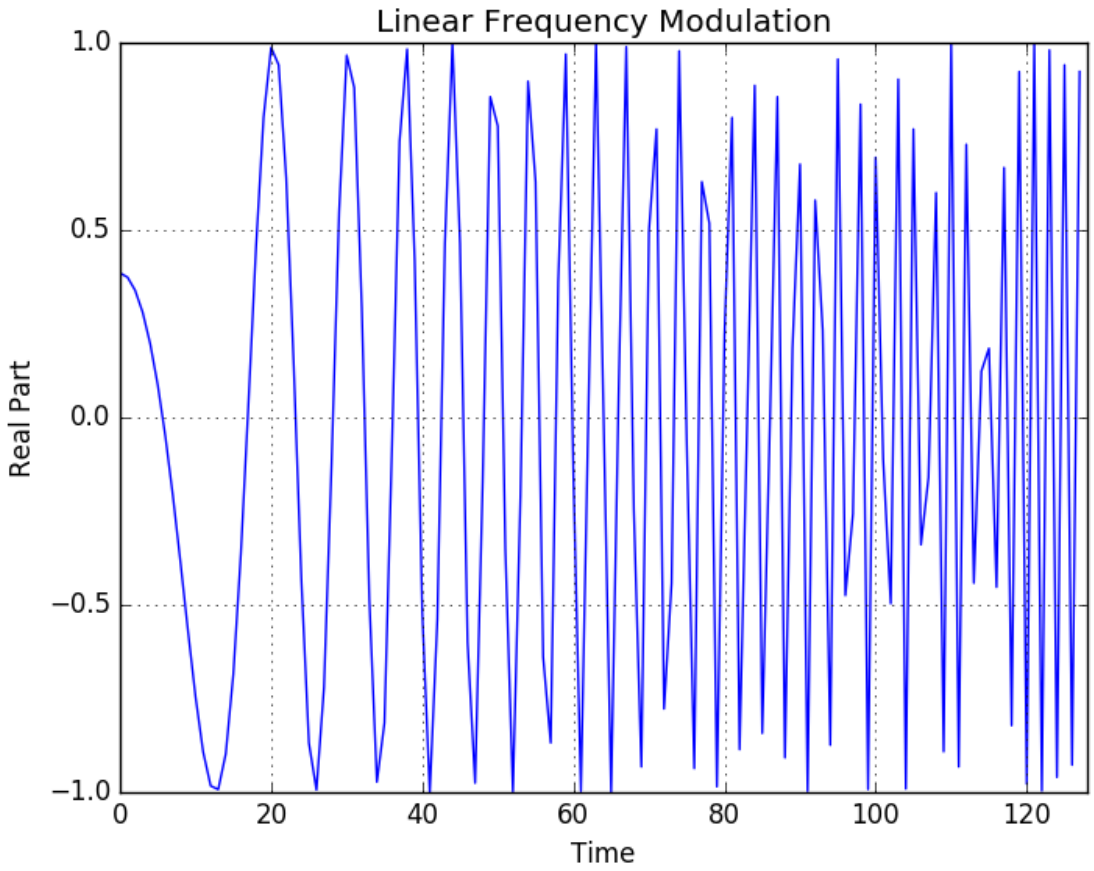

```
import numpy as np
from tftb.processing.linear import ShortTimeFourierTransform
from tftb.generators import fmlin, amgauss
import matplotlib.pyplot as plt

x = np.real(amgauss(128) * fmlin(128)[0])
window = np.ones((128,))
stft = ShortTimeFourierTransform(x, n_fbins=128, fwindow=window)
stft.run()
stft.plot(show_tf=True, cmap=plt.cm.gray)
```

Total running time of the script: (0 minutes 0.251 seconds)

3.1.18 Linear Frequency Modulation

This example shows how PyTFTB is used to generate a signal with linear frequency modulation. Such a signal is also called a chirp.

Figure 1.1 from the tutorial.



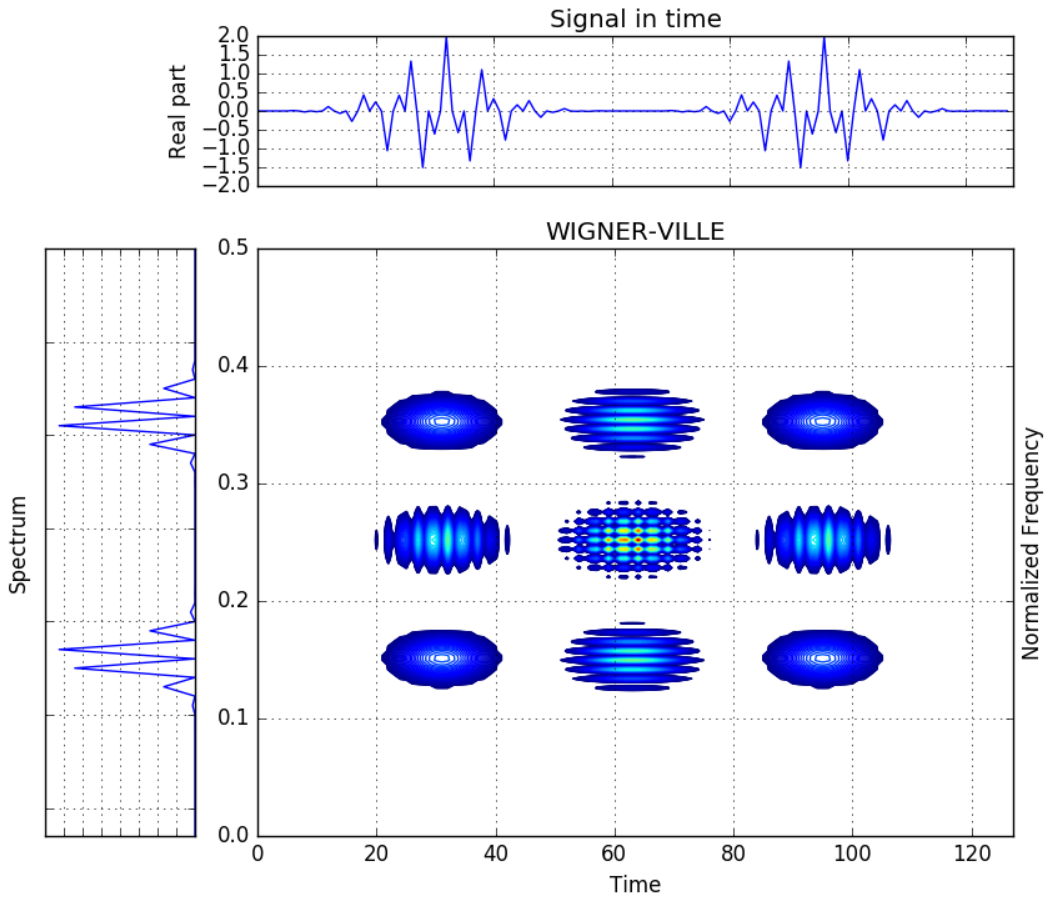

```
from tftb.generators import fmlin
import matplotlib.pyplot as plt
import numpy as np

# Generate a chirp signal

n_points = 128
fmin, fmax = 0.0, 0.5

signal, _ = fmlin(n_points, fmin, fmax)
plt.plot(np.real(signal))
plt.xlim(0, n_points)
plt.title('Linear Frequency Modulation')
plt.ylabel('Real Part')
plt.xlabel('Time')
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.167 seconds)

3.1.19 Wigner-Ville Distribution of Gaussian Atoms

This example shows the WV distribution of four Gaussian atoms, each localized at the corner of a rectangle in the time-frequency plane. The distribution does show the four signal terms, as well as six interference terms.

Figure 4.4 from the tutorial.




```
import numpy as np
from tftb.generators import atoms
from tftb.processing import WignerVilleDistribution

x = np.array([[32, .15, 20, 1],
              [96, .15, 20, 1],
              [32, .35, 20, 1],
              [96, .35, 20, 1]])
g = atoms(128, x)
spec = WignerVilleDistribution(g)
spec.run()
spec.plot(kind="contour", show_tf=True, scale="log")
```

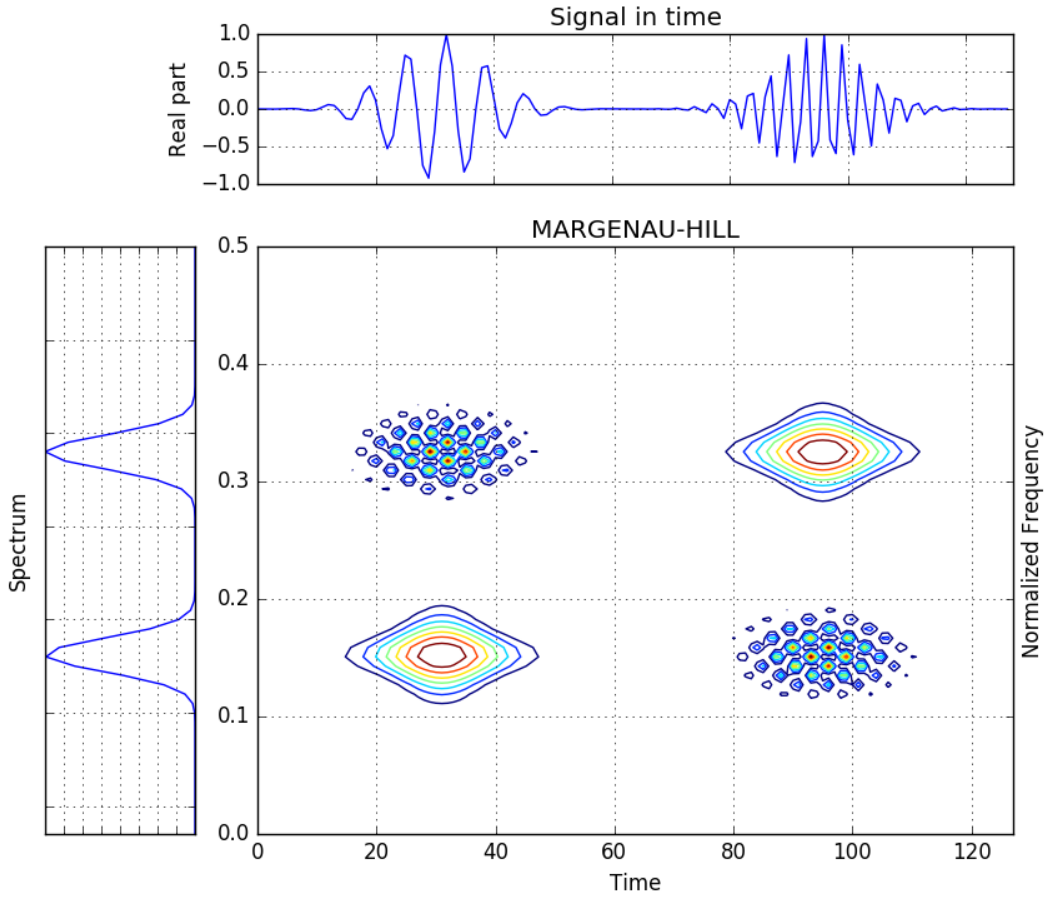
Total running time of the script: (0 minutes 0.450 seconds)

3.1.20 Margenau-Hill Representation of Chirps with Different Slopes

This example demonstrates the Margenau-Hill distribution of a signal composed of two chirps with Gaussian amplitude modulation but havind linear frequency modulations with different slopes. This distribution too, like the Wigner-Ville distribution, spearates the signal terms, but produces interferences such that they appear diagonally opposite their

corresponding signals.

Figure 4.14 from the tutorial.



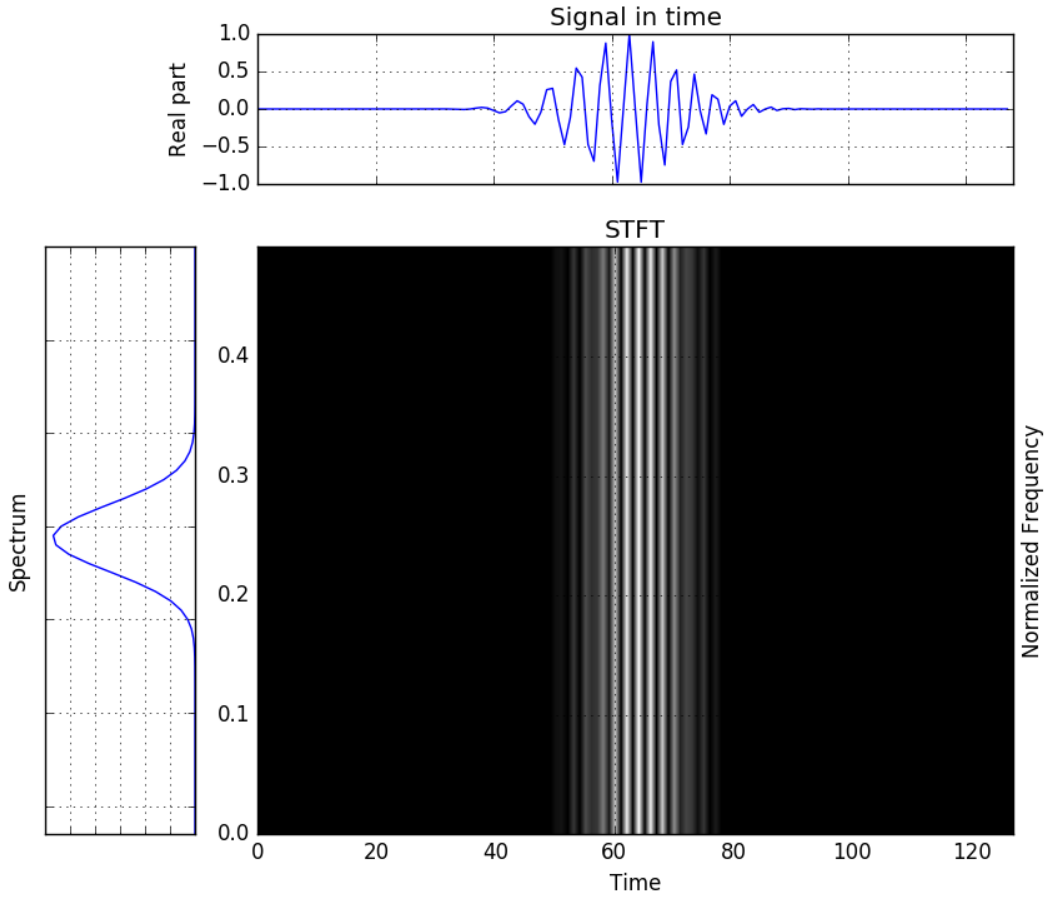

```
import numpy as np
from tftb.generators import atoms
from tftb.processing import MargenauHillDistribution


sig = atoms(128, np.array([[32, 0.15, 20, 1], [96, 0.32, 20, 1]]))
tfr = MargenauHillDistribution(sig)
tfr.run()
tfr.plot(show_tf=True, kind='contour', sqmod=False, threshold=0,
         contour_y=np.linspace(0, 0.5, tfr.tfr.shape[0] / 2))
```

Total running time of the script: (0 minutes 0.399 seconds)

3.1.21 Ideal time resolution

This example demonstrates that only the shortest possible window can provide ideal resolution in time.

Figure 3.5 from the tutorial.

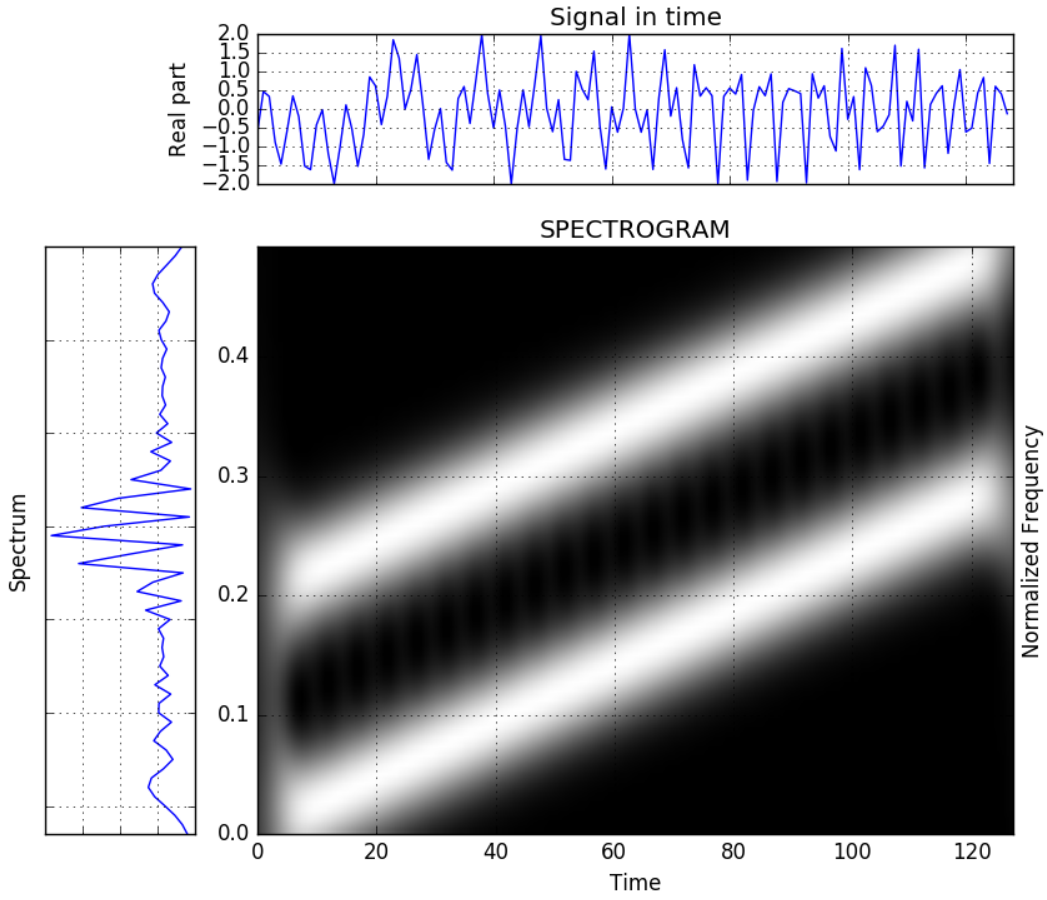



```
import numpy as np
from tftb.processing.linear import ShortTimeFourierTransform
from tftb.generators import fmlin, amgauss
from matplotlib.pyplot import cm

x = np.real(amgauss(128) * fmlin(128)[0])
window = np.array([1])
stft = ShortTimeFourierTransform(x, n_fbins=128, fwindow=window)
tfr, _, _ = stft.run()

stft.plot(show_tf=True, cmap=cm.gray)
```

Total running time of the script: (0 minutes 0.217 seconds)

3.1.22 Distant Chirps with a Short Gaussian Analysis Window

Figure 3.17 from the tutorial.



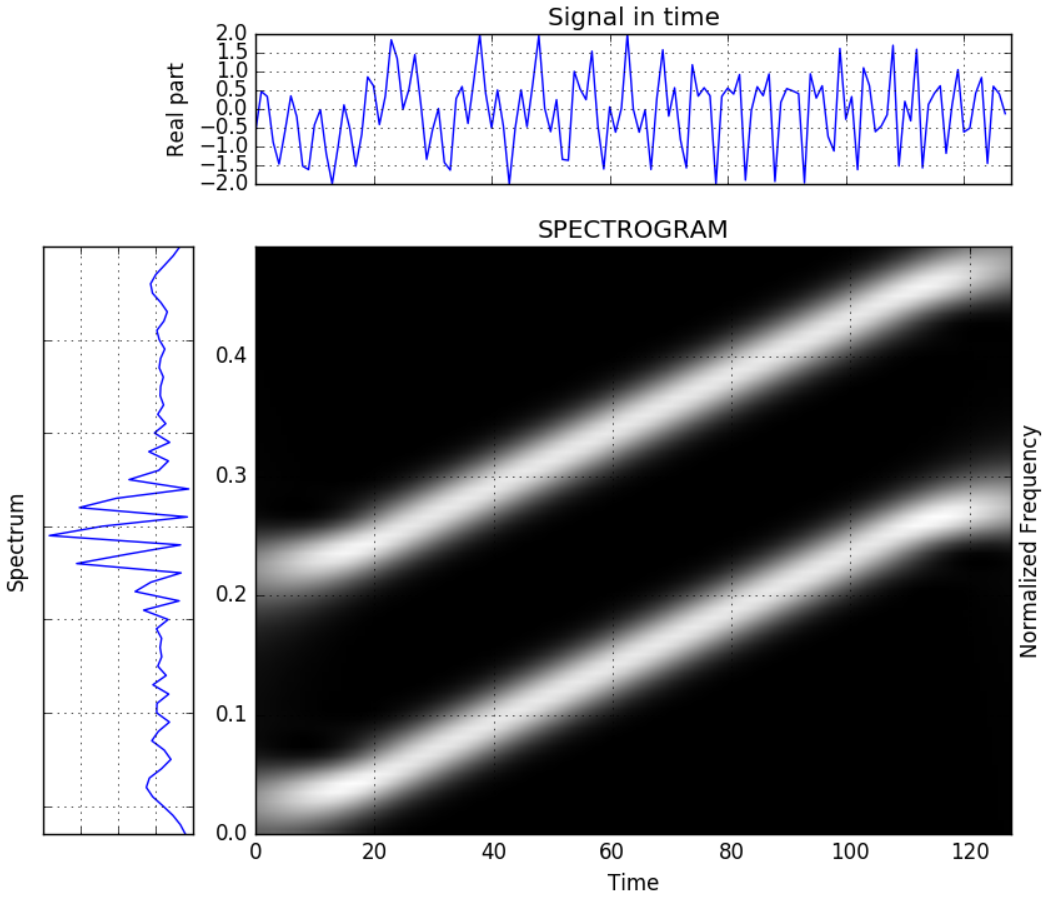

```
from tftb.generators import fmlin
from tftb.processing.cohen import Spectrogram
import numpy as np
import matplotlib.pyplot as plt

sig = fmlin(128, 0, 0.3)[0] + fmlin(128, 0.2, 0.5)[0]
window = np.exp(np.log(0.005) * np.linspace(-1, 1, 23) ** 2)
spec = Spectrogram(sig, fwindow=window, n_fbins=128)
spec.run()
spec.plot(show_tf=True, cmap=plt.cm.gray)
```

Total running time of the script: (0 minutes 0.327 seconds)

3.1.23 Distant Chirps with a Long Gaussian Analysis Window

This example visualizes a spectrogram of two chirp signals which are well separated in frequency ranges. A longer Gaussian analysis window suffices here to see the separation of frequencies, since the variation in frequencies is relatively slow.

Figure 3.18 from the tutorial.



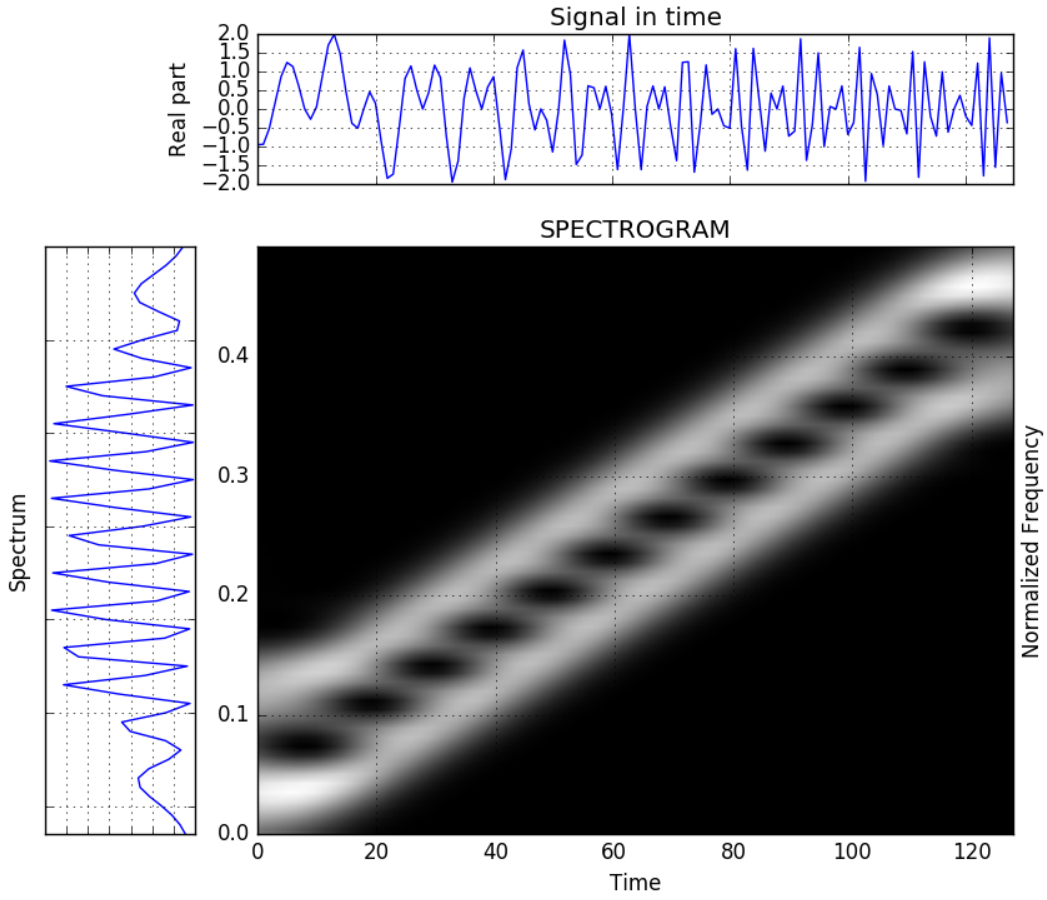

```
from tftb.generators import fmlin
from tftb.processing.cohen import Spectrogram
import numpy as np
import matplotlib.pyplot as plt

sig = fmlin(128, 0, 0.3)[0] + fmlin(128, 0.2, 0.5)[0]
window = np.exp(np.log(0.005) * np.linspace(-1, 1, 63) ** 2)
spec = Spectrogram(sig, fwindow=window, n_fbins=128)
spec.run()
spec.plot(show_tf=True, cmap=plt.cm.gray)
```

Total running time of the script: (0 minutes 0.209 seconds)

3.1.24 Spectrogram of Parallel Chirps with a Long Gaussian Analysis Window

This example visualizes the spectrogram of two "parallel" chirps, using a Gaussian window function that has a long length, relative to the length of a signal. The two chirps can be made out, but interference can also be seen along the time axis, since time resolution is compromised.

Figure 3.16 from the tutorial.



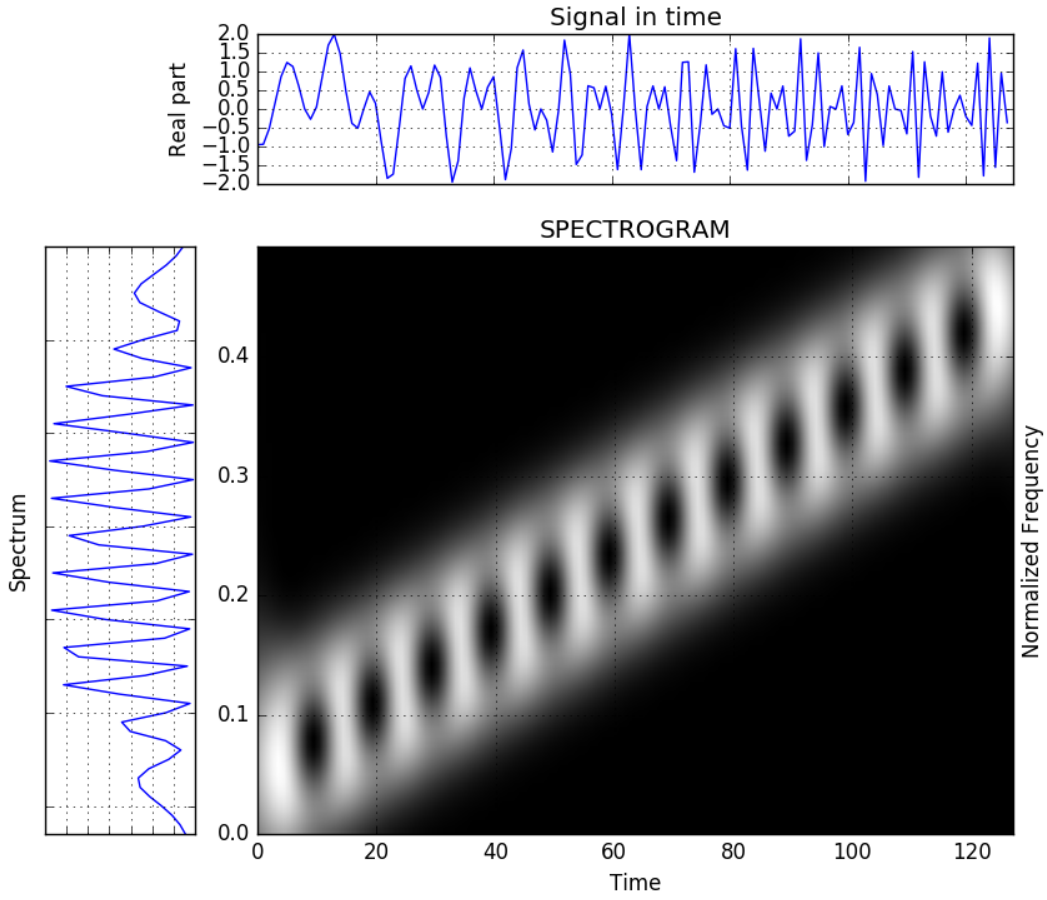

```
from tftb.generators import fmlin
from tftb.processing.cohen import Spectrogram
import numpy as np
import matplotlib.pyplot as plt

sig = fmlin(128, 0, 0.4)[0] + fmlin(128, 0.1, 0.5)[0]
window = np.exp(np.log(0.005) * np.linspace(-1, 1, 63) ** 2)
spec = Spectrogram(sig, fwindow=window, n_fbins=128)
spec.run()
spec.plot(show_tf=True, cmap=plt.cm.gray)
```

Total running time of the script: (0 minutes 0.239 seconds)

3.1.25 Spectrogram of Parallel Chirps with Short Gaussian Analysis Window

This example visualizes the spectrogram of two “parallel” chirps, using a Gaussian window function that has a short length, relative to the length of a signal. The two chirps can be made out, but interference can also be seen along the frequency axis, since frequency resolution is compromised.

Figure 3.15 from the tutorial.



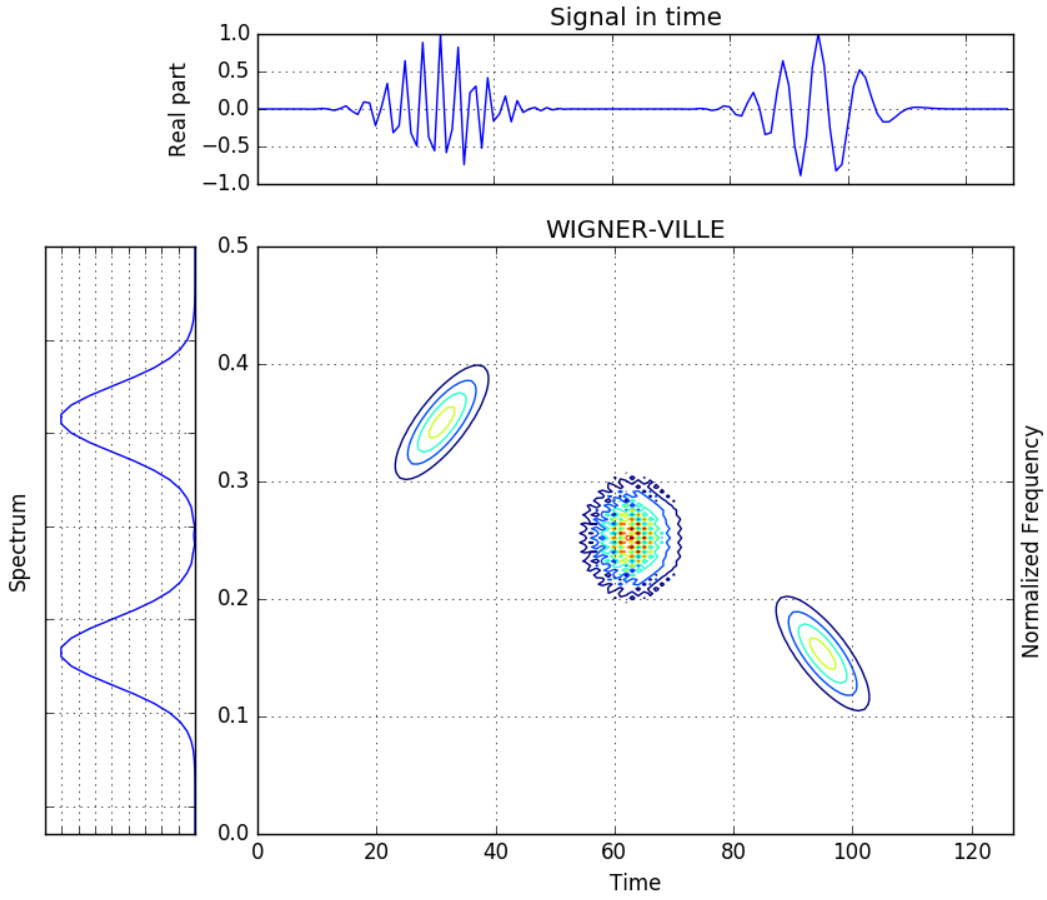

```
from tftb.generators import fmlin
from tftb.processing.cohen import Spectrogram
import numpy as np
import matplotlib.pyplot as plt

sig = fmlin(128, 0, 0.4)[0] + fmlin(128, 0.1, 0.5)[0]
window = np.exp(np.log(0.005) * np.linspace(-1, 1, 23) ** 2)
spec = Spectrogram(sig, fwindow=window, n_fbins=128)
spec.run()
spec.plot(show_tf=True, cmap=plt.cm.gray)
```

Total running time of the script: (0 minutes 0.259 seconds)

3.1.26 Wigner-Ville Distribution of Chirps with Different Slopes

This example demonstrates the Wigner-Ville distribution of a signal composed of two chirps with Gaussian amplitude modulation but havind linear frequency modulations with different slopes. Note that the AF interference terms are located away from the origin. We can see the two distint signal terms, but there is some interference around the middle.

Figure 4.12 from the tutorial.



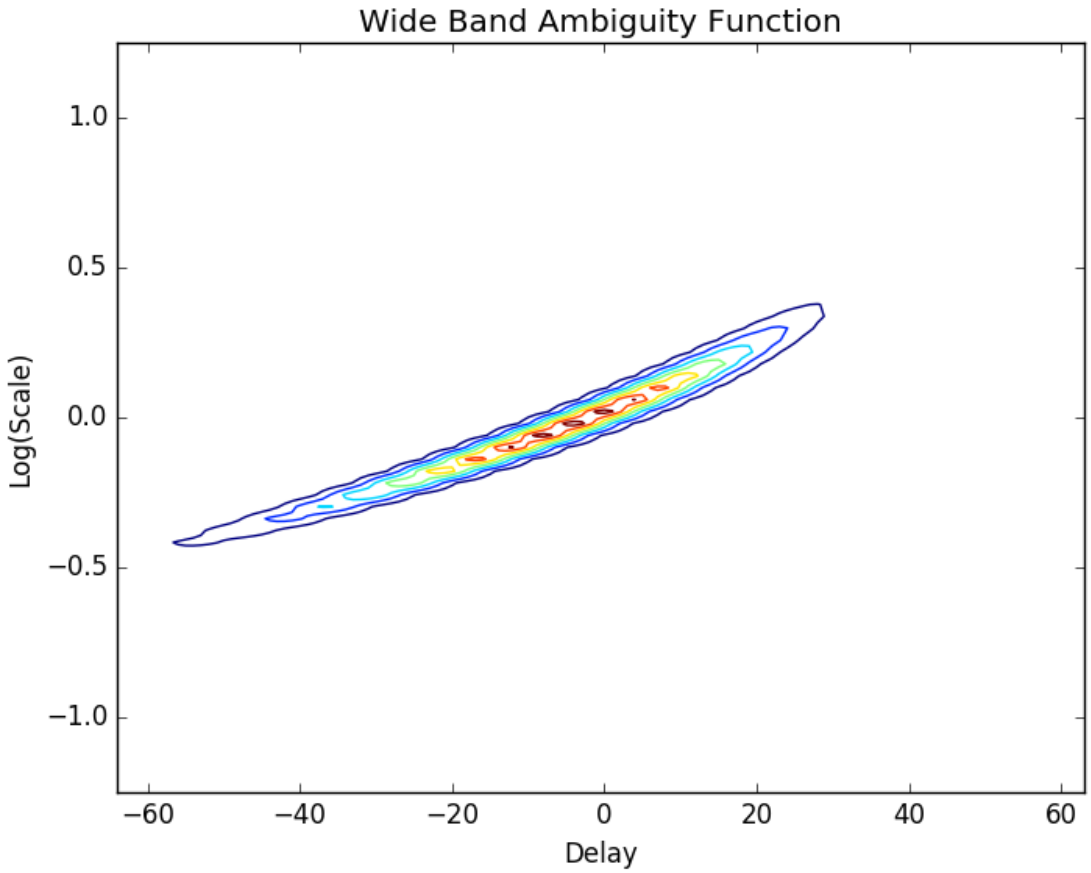

```
from tftb.generators import fmlin, amgauss
from tftb.processing import WignerVilleDistribution
import numpy as np

n_points = 64
sig1 = fmlin(n_points, 0.2, 0.5)[0] * amgauss(n_points)
sig2 = fmlin(n_points, 0.3, 0)[0] * amgauss(n_points)
sig = np.hstack((sig1, sig2))

tfr = WignerVilleDistribution(sig)
tfr.run()
tfr.plot(kind='contour', show_tf=True)
```

Total running time of the script: (0 minutes 0.300 seconds)

3.1.27 Wideband Ambiguity Function of an Altes Signal

For wideband signals, the narrow band ambiguity function is not appropriate for wideband signals. So we consider a wide band ambiguity function. This function is equivalent to the wavelet transform of a signal whose mother wavelet is the signal itself.

Figure 4.25 from the tutorial.



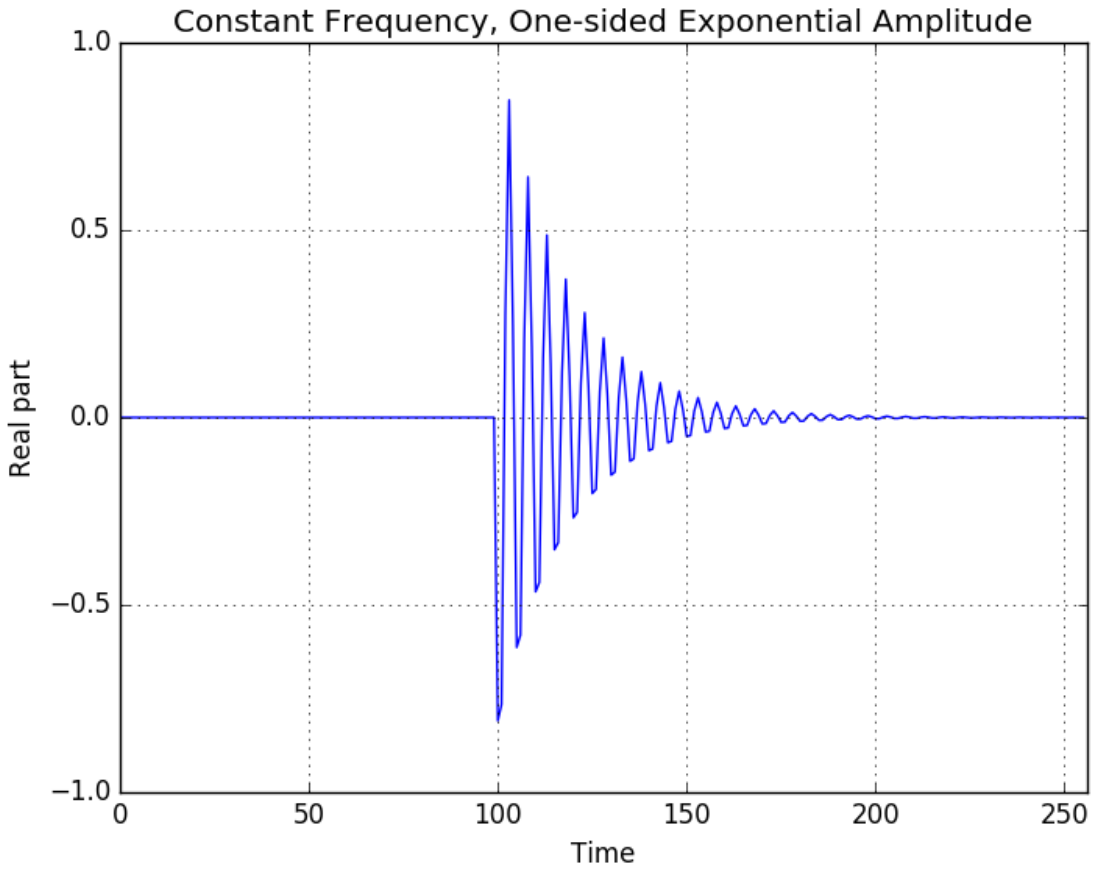

```
from tftb.generators import altes
from tftb.processing.ambiguity import wide_band
import matplotlib.pyplot as plt
import numpy as np

signal = altes(128, 0.1, 0.45)
waf, tau, theta = wide_band(signal, 0.1, 0.35, 64)
plt.contour(tau, theta, np.abs(waf) ** 2)
plt.xlabel("Delay")
plt.ylabel("Log(Scale)")
plt.title("Wide Band Ambiguity Function")
plt.show()
```

Total running time of the script: (0 minutes 0.082 seconds)

3.1.28 Monocomponent Nonstationary Signal with Constant Frequency Modulation and One-Sided Exponential Amplitude Modulation

Generate a monocomponent nonstationary signal with constant frequency modulation and one-sided exponential amplitude modulation.

Figure 2.7 from the tutorial.



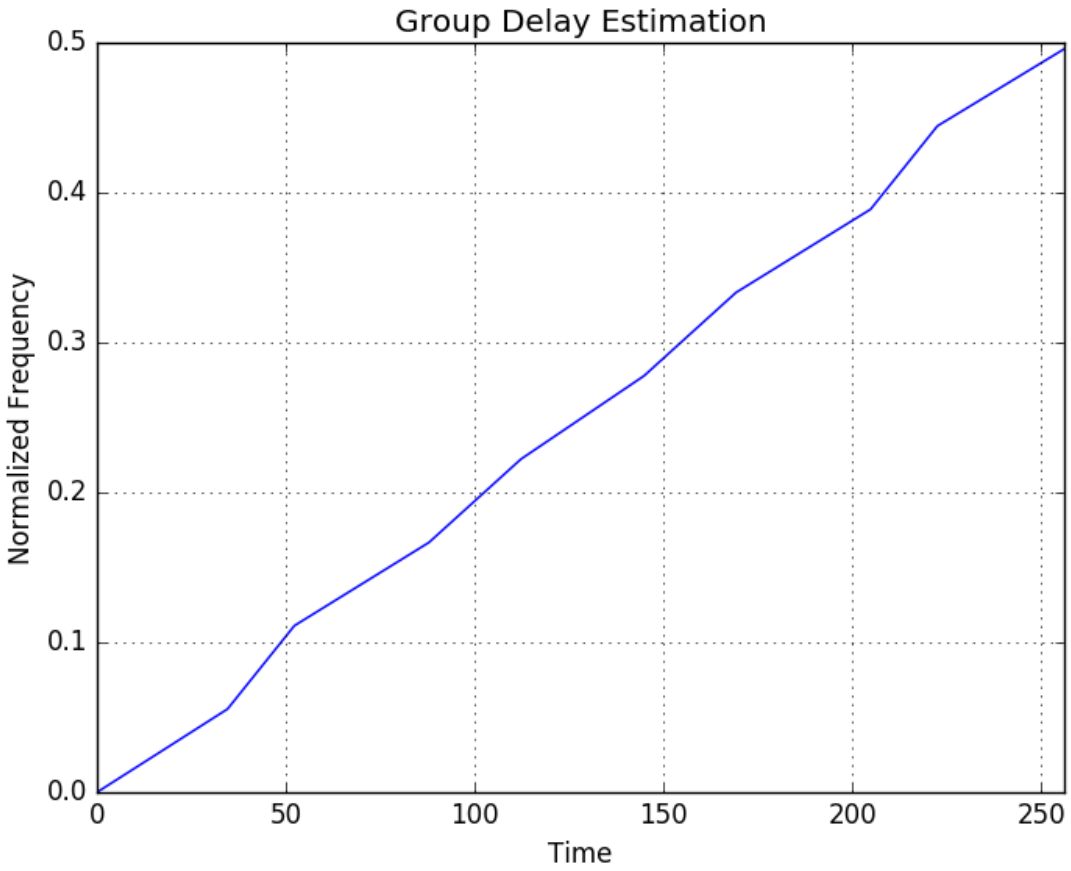

```
from tftb.generators import fmconst, amexpos
import matplotlib.pyplot as plt
from numpy import real

fm, _ = fmconst(256, 0.2)
am = amexpos(256, 100, kind='unilateral')
signal = am * fm

plt.plot(real(signal))
plt.xlabel('Time')
plt.ylabel('Real part')
plt.title('Constant Frequency, One-sided Exponential Amplitude')
plt.xlim(0, 256)
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.071 seconds)

3.1.29 Group Delay Estimation of a Chirp

Constuct a chirp and estimates its group delay.

Figure 2.4 from the tutorial.



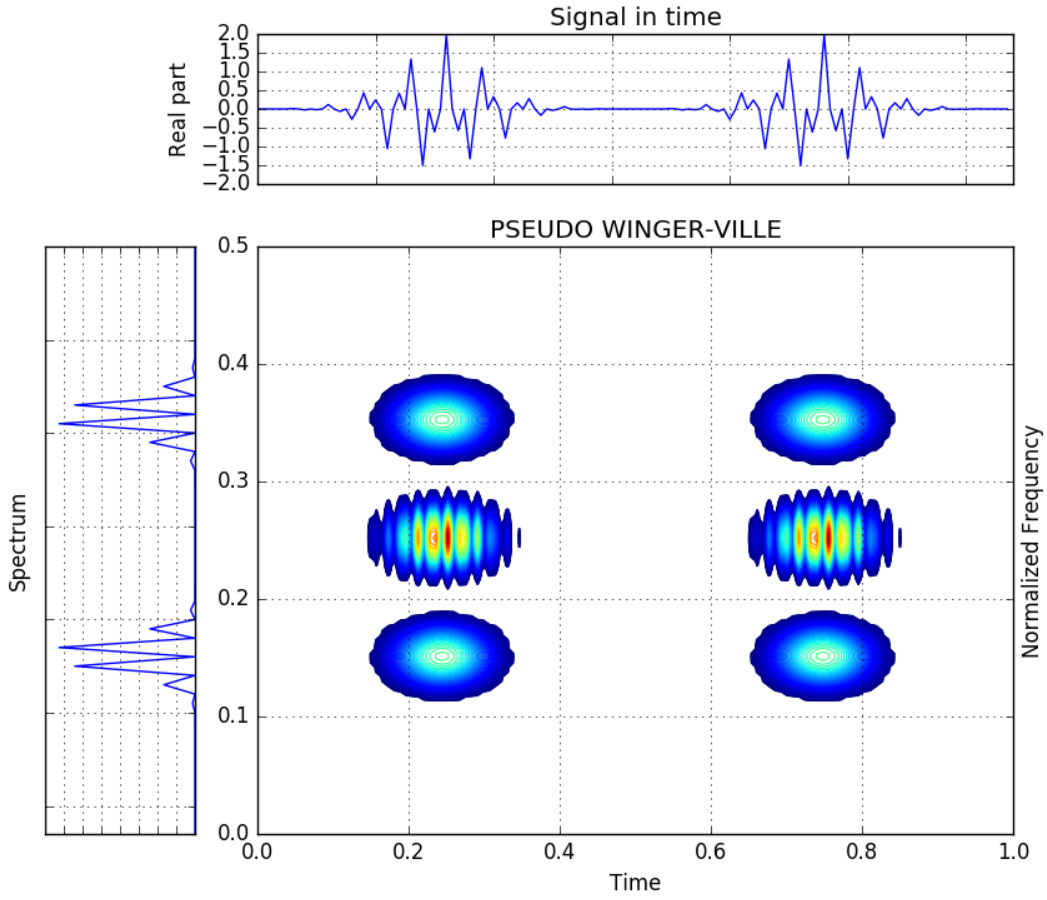

```
from tftb.generators import fmlin
from tftb.processing import group_delay
import numpy as np
import matplotlib.pyplot as plt

signal, _ = fmlin(256)
fnorm = np.linspace(0, .5, 10)
gd = group_delay(signal, fnorm)
plt.plot(gd, fnorm)
plt.grid(True)
plt.xlim(0, 256)
plt.xlabel('Time')
plt.ylabel('Normalized Frequency')
plt.title('Group Delay Estimation')
plt.show()
```

Total running time of the script: (0 minutes 0.060 seconds)

3.1.30 Pseudo Wigner-Ville Distribution of Gaussian Atoms

This example shows the Pseudo Wigner-Ville distribution of four Gaussian atoms located at the corners of a rectangle in the time-frequency plane. The *PseudoWignerVilleDistribution* class uses frequency smoothing, which attenuates the interferences oscillating along the time axis.

Figure 4.5 from the tutorial.



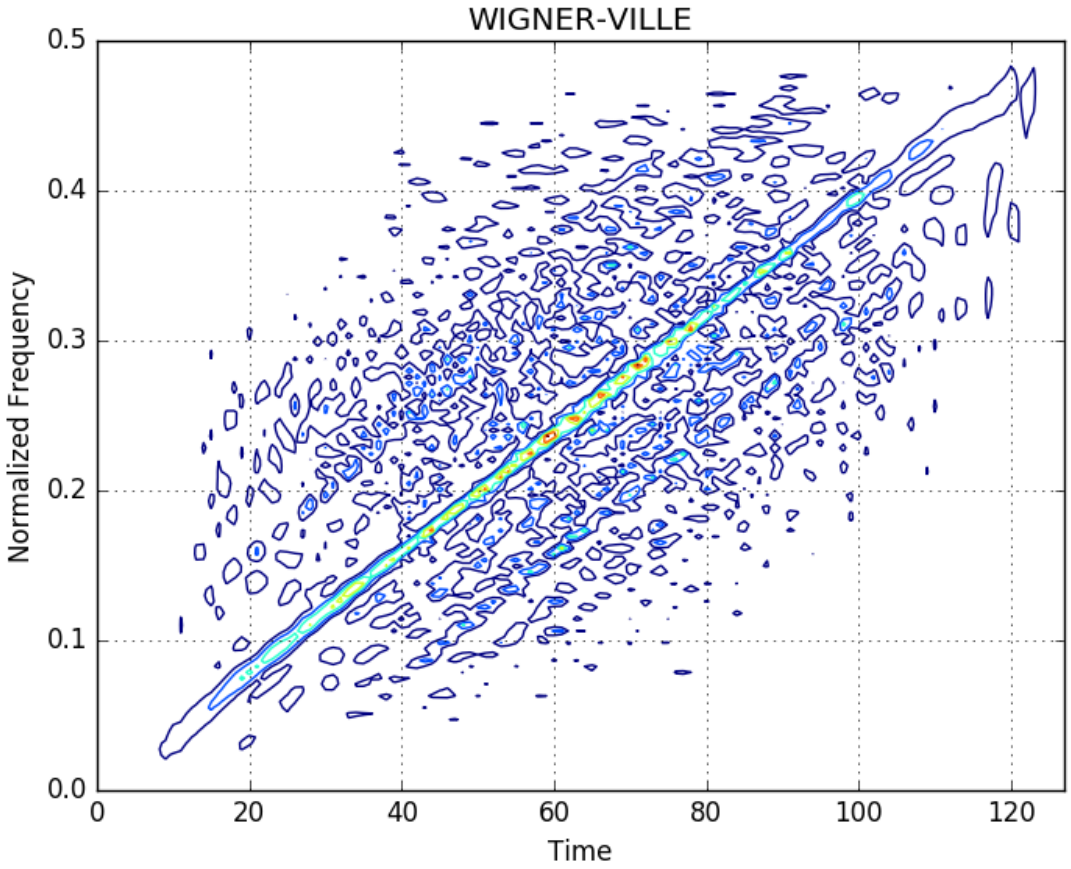

```
import numpy as np
from tftb.generators import atoms
from tftb.processing import PseudoWignerVilleDistribution

x = np.array([[32, .15, 20, 1],
              [96, .15, 20, 1],
              [32, .35, 20, 1],
              [96, .35, 20, 1]])
g = atoms(128, x)
t = np.linspace(0, 1, 128)
spec = PseudoWignerVilleDistribution(g, timestamps=t)
spec.run()
spec.plot(kind="contour", scale="log", show_tf=True)
```

Total running time of the script: (0 minutes 0.372 seconds)

3.1.31 Wigner-Ville Distribution of a Noisy Chirp

Generate a noisy chirp and visualize its Wigner-Ville spectrum.

Figure 1.6 from the tutorial.



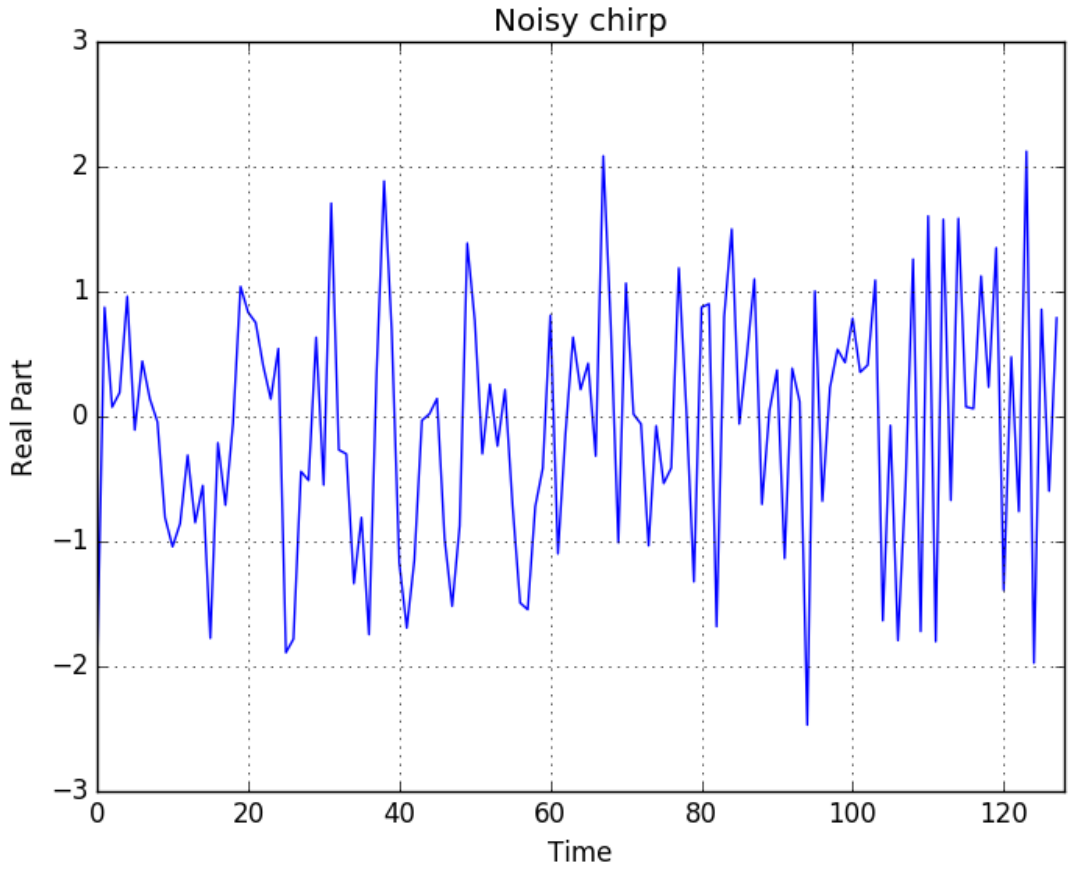

```
from tftb.generators import fmlin, sigmerge, noisecg
from tftb.processing.cohen import WignerVilleDistribution

# Generate a chirp signal

n_points = 128
fmin, fmax = 0.0, 0.5

signal, _ = fmlin(n_points, fmin, fmax)

# Noisy chirp

noisy_signal = sigmerge(signal, noisecg(128), 0)


# Wigner-Ville spectrum of noisy chirp.

wvd = WignerVilleDistribution(noisy_signal)
wvd.run()
wvd.plot(kind='contour')
```

Total running time of the script: (0 minutes 0.107 seconds)

3.1.32 Generating a Noisy Chirp

This example shows how to generate a chirp signal, with some analytical gaussian noise, and the usage of the sigmerge function to combine them.

Figure 1.4 from the tutorial.



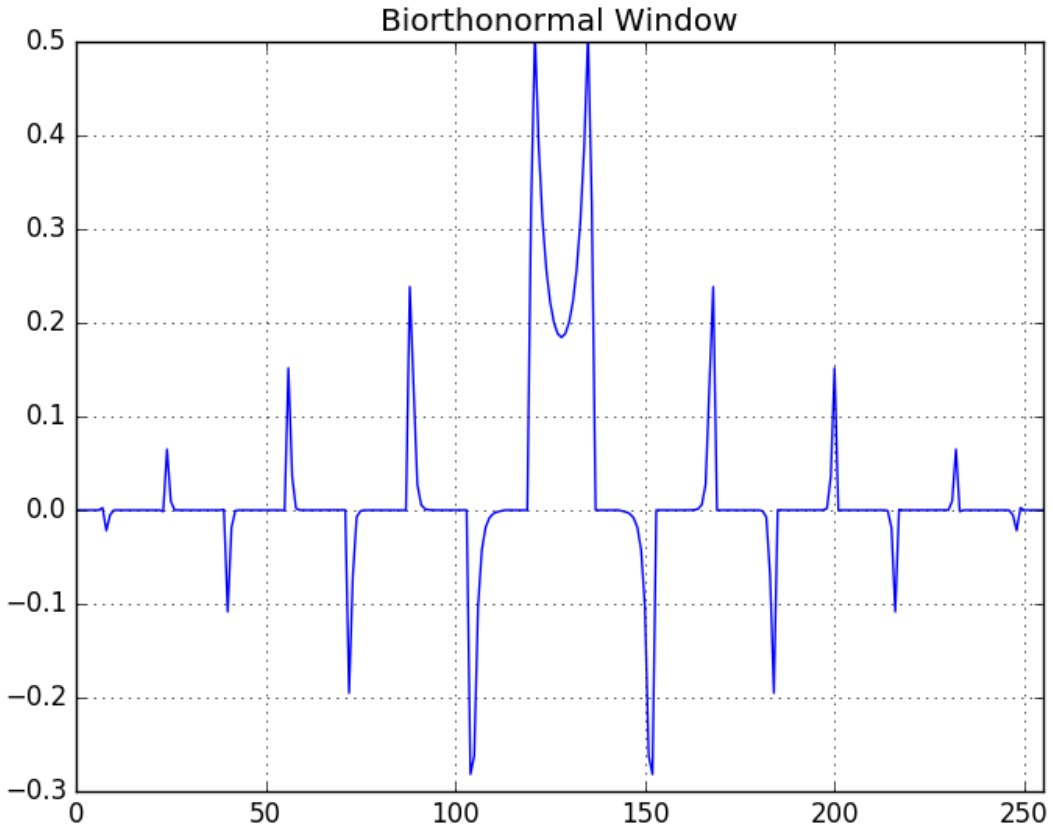

```
from tftb.generators import fmlin, sigmerge, noisecg
import matplotlib.pyplot as plt
import numpy as np

# Generate a chirp signal

n_points = 128
fmin, fmax = 0.0, 0.5

signal, _ = fmlin(n_points, fmin, fmax)

# Noisy chirp

noisy_signal = sigmerge(signal, noisecg(128), 0)
plt.plot(np.real(noisy_signal))
plt.xlim(0, 128)
plt.title('Noisy chirp')
plt.ylabel('Real Part')
plt.xlabel('Time')
```

```
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.061 seconds)

3.1.33 Biorthonormal Window Function

Figure 3.10 from the tutorial.




```
from tftb.generators import fmlin
from tftb.processing.linear import gabor
import matplotlib.pyplot as plt
import numpy as np

N1 = 256
Ng = 33
Q = 1
sig = fmlin(N1)[0]
window = np.exp(np.log(0.005) * np.linspace(-1, 1, Ng) ** 2)
window = window / np.linalg.norm(window)
tfr, dgr, h = gabor(sig, 16, Q, window)
plt.plot(h)
plt.ylim(top=0.5)
plt.xlim(right=255)
```

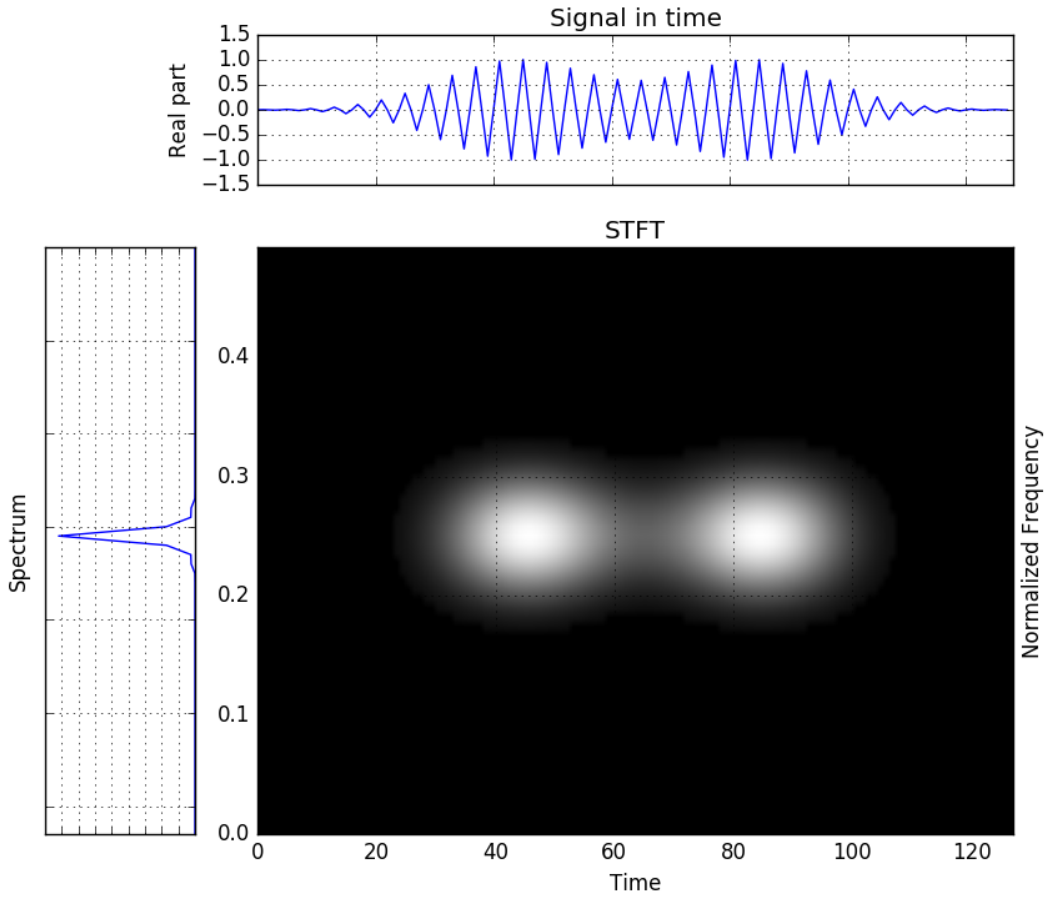
```
plt.title('Biorthonormal Window')
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.061 seconds)

3.1.34 Time-frequency Resolution: Short Analysis Window

This example shows the effect of an analysis window which is short in time on the time-frequency resolution. Specifically, smaller windows have good time resolutions but poor frequency resolutions.

Figure 3.8 from the tutorial.




```
import numpy as np
import matplotlib.pyplot as plt
from tftb.generators import atoms
from scipy.signal import hamming
from tftb.processing.linear import ShortTimeFourierTransform

coords = np.array([[45, .25, 32, 1], [85, .25, 32, 1]])
sig = atoms(128, coords)
x = np.real(sig)
```

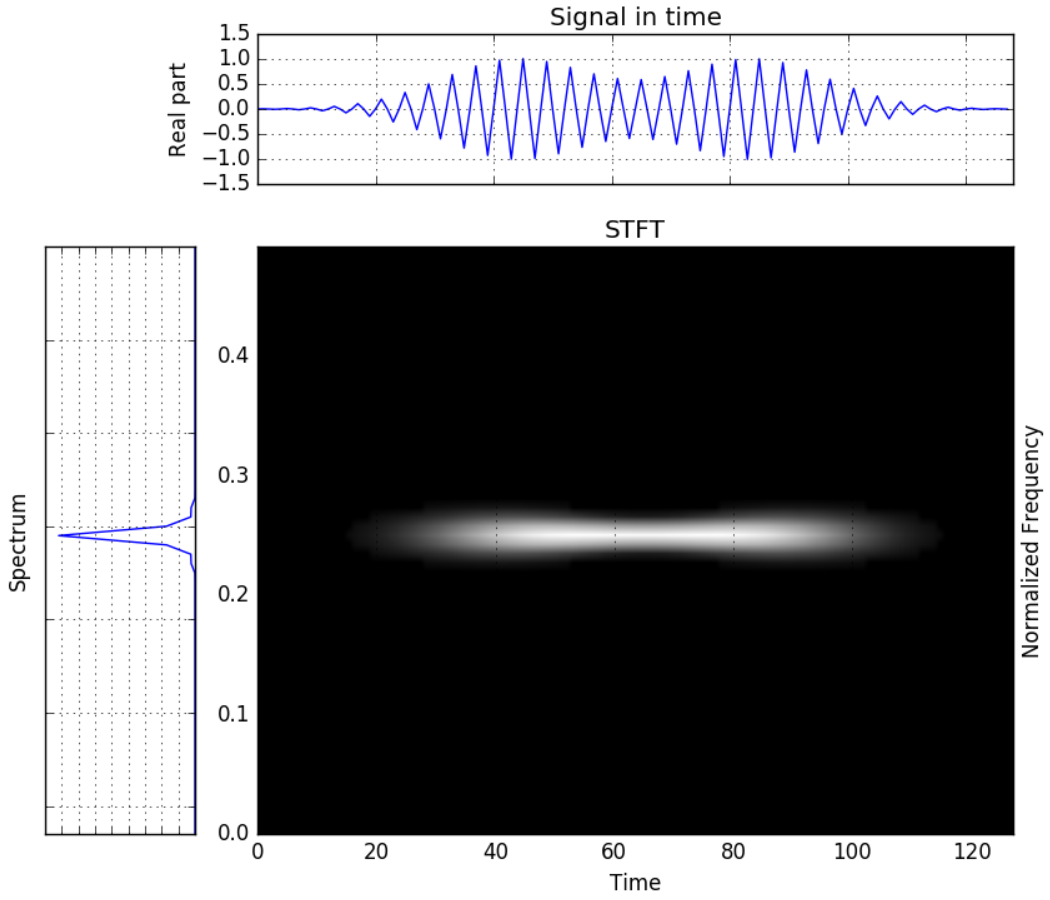
```
window = hamming(17)
stft = ShortTimeFourierTransform(sig, n_fbins=128, fwindow=window)
stft.run()
stft.plot(show_tf=True, cmap=plt.cm.gray)
```

Total running time of the script: (0 minutes 0.319 seconds)

3.1.35 STFT of Gaussian Wave Packets with a Hamming Analysis Window

This example demonstrates the construction of a signal containing two transient components, having the same Gaussian amplitude modulation and the same frequency, but different time centers. It also shows the effect of a Hamming window function when used with th STFT.

Figure 3.7 from the tutorial.




```
import numpy as np
import matplotlib.pyplot as plt
from tftb.generators import atoms
from scipy.signal import hamming
from tftb.processing.linear import ShortTimeFourierTransform

coords = np.array([[45, .25, 32, 1], [85, .25, 32, 1]])
```

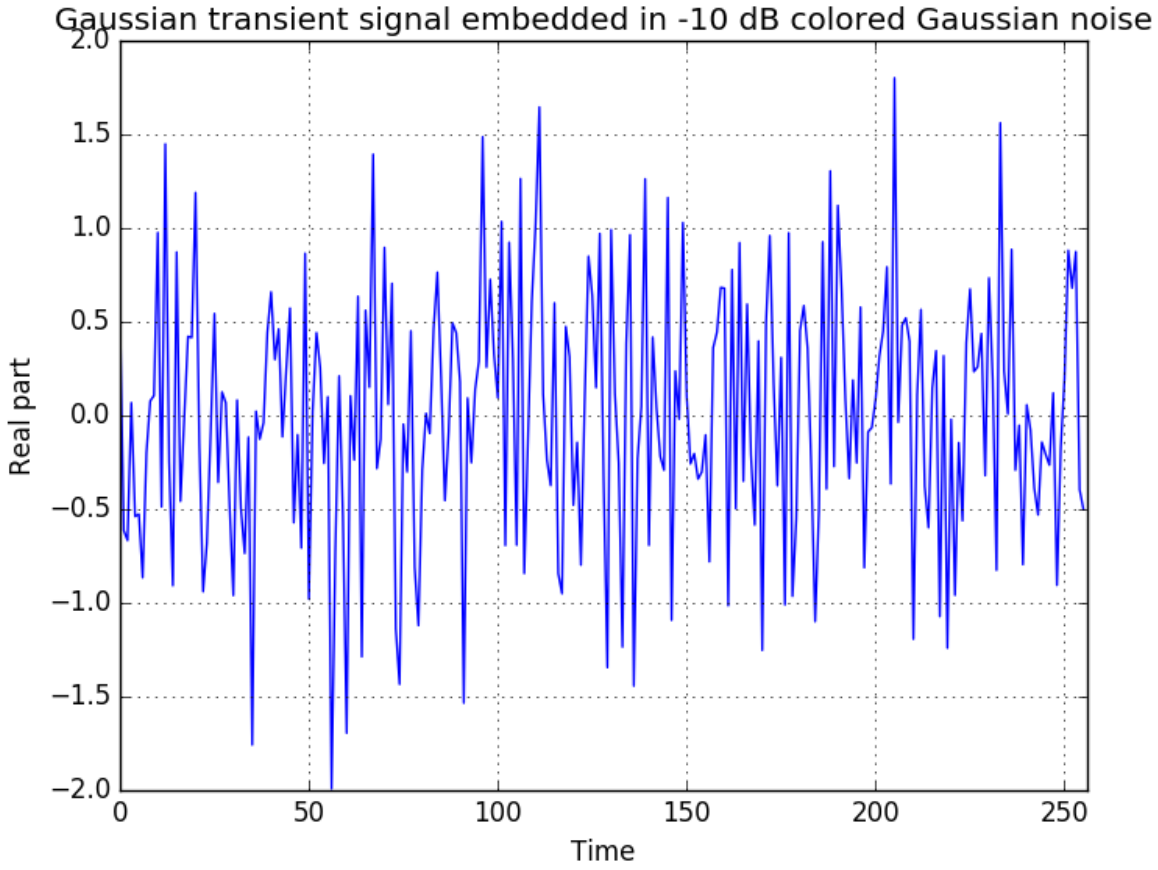
```
sig = atoms(128, coords)
x = np.real(sig)
window = hamming(65)
stft = ShortTimeFourierTransform(sig, n_fbins=128, fwindow=window)
stft.run()
stft.plot(show_tf=True, cmap=plt.cm.gray)
```

Total running time of the script: (0 minutes 0.216 seconds)

3.1.36 Noisy Monocomponent Chirp

This example demonstrates the construction of a monocomponent signal with linear frequency modulation and colored Gaussian noise.

Figure 2.9 from the tutorial.




```
from tftb.generators import fmlin, amgauss, noisecg, sigmerge
from numpy import real
import matplotlib.pyplot as plt

fm, _ = fmlin(256)
am = amgauss(256)
signal = fm * am
```

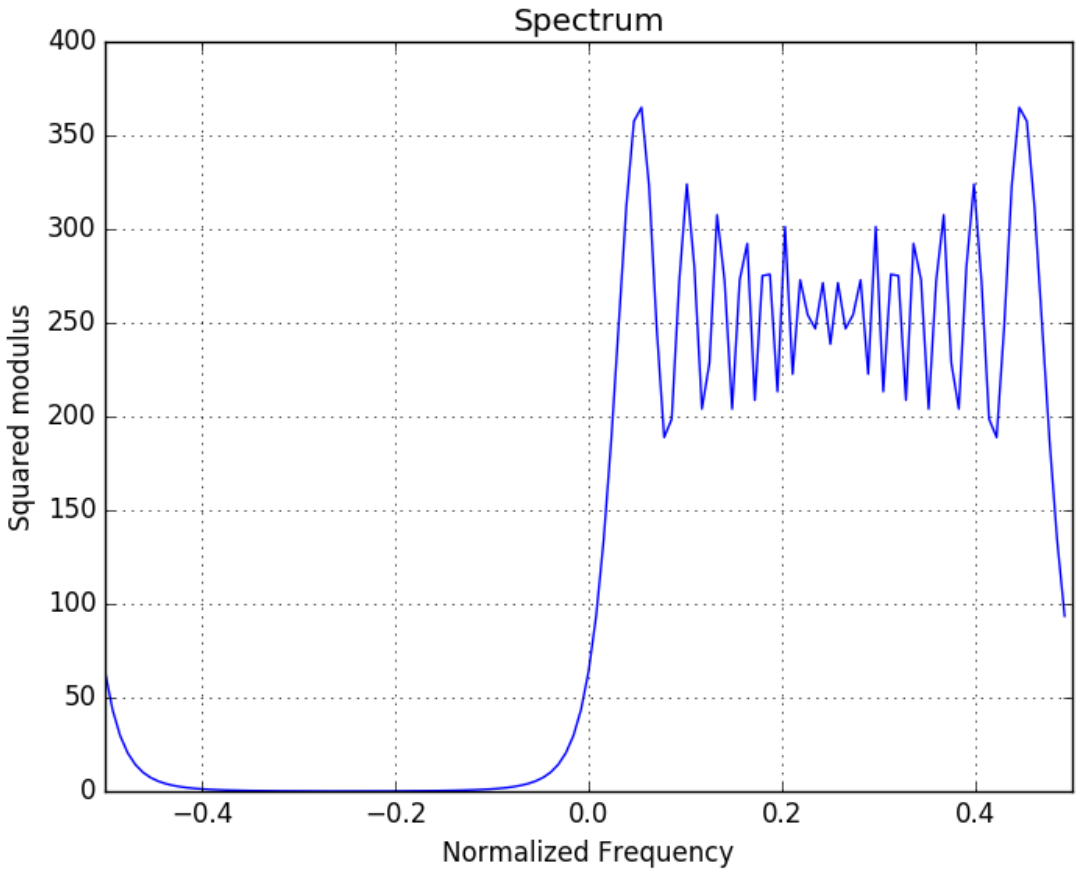
```
noise = noisecg(256, .8)
sign = sigmerge(signal, noise, -10)

plt.plot(real(sign))
plt.xlabel('Time')
plt.ylabel('Real part')
plt.title('Gaussian transient signal embedded in -10 dB colored Gaussian noise')
plt.xlim(0, 256)
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.064 seconds)

3.1.37 Energy Spectral Density of a Chirp

Construct a chirp and plot its energy spectral density.

Figure 1.2 from the tutorial.




```
from tftb.generators import fmlin
import matplotlib.pyplot as plt
import numpy as np

n_points = 128
```

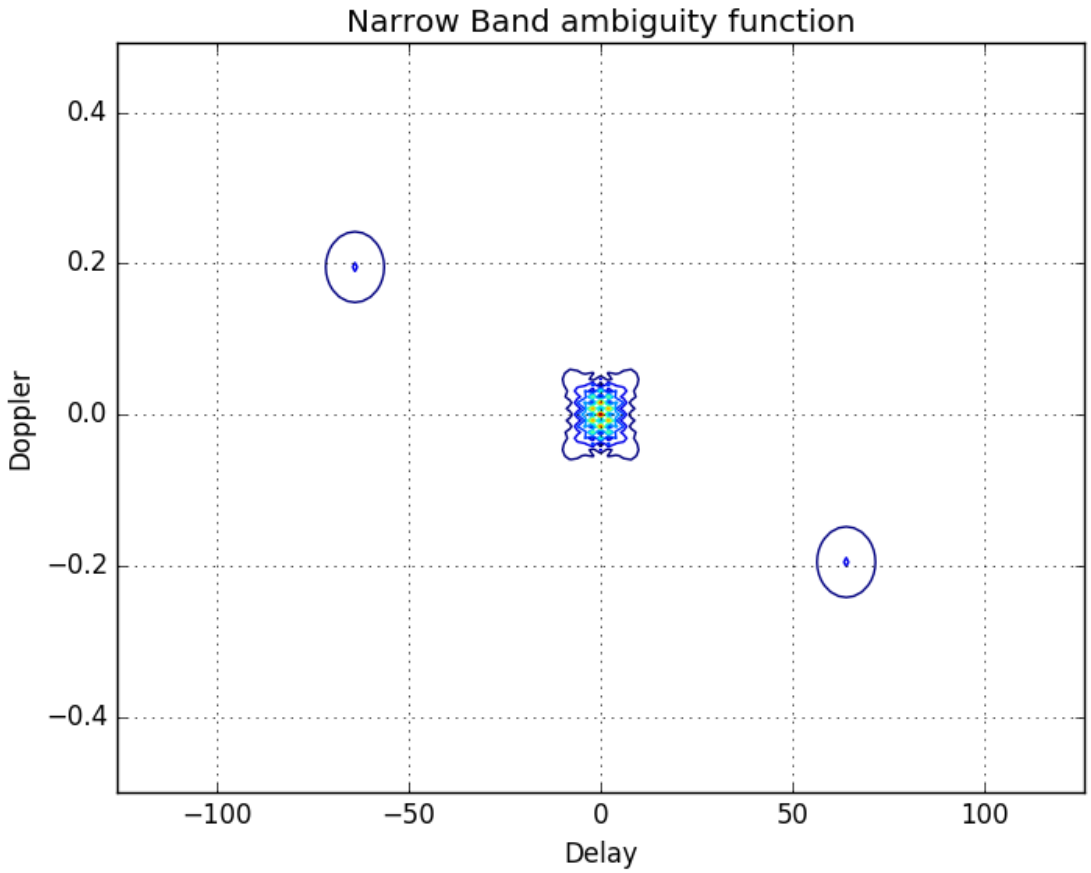
```
fmin, fmax = 0.0, 0.5
signal, _ = fmlin(n_points, fmin, fmax)

# Plot the energy spectrum of the chirp

dsp1 = np.fft.fftshift(np.abs(np.fft.fft(signal)) ** 2)
plt.plot(np.arange(-64, 64, dtype=float) / 128.0, dsp1)
plt.xlim(-0.5, 0.5)
plt.title('Spectrum')
plt.ylabel('Squared modulus')
plt.xlabel('Normalized Frequency')
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.069 seconds)

3.1.38 Narrow Band Ambiguity Function of Chirps with Different Slopes

This example demonstrates the narrow band ambiguity function (AF) of a signal composed of two chirps with Gaussian amplitude modulation but havind linear frequency modulations with different slopes. Note that the AF interference terms are located away from the origin.

Figure 4.13 from the tutorial.

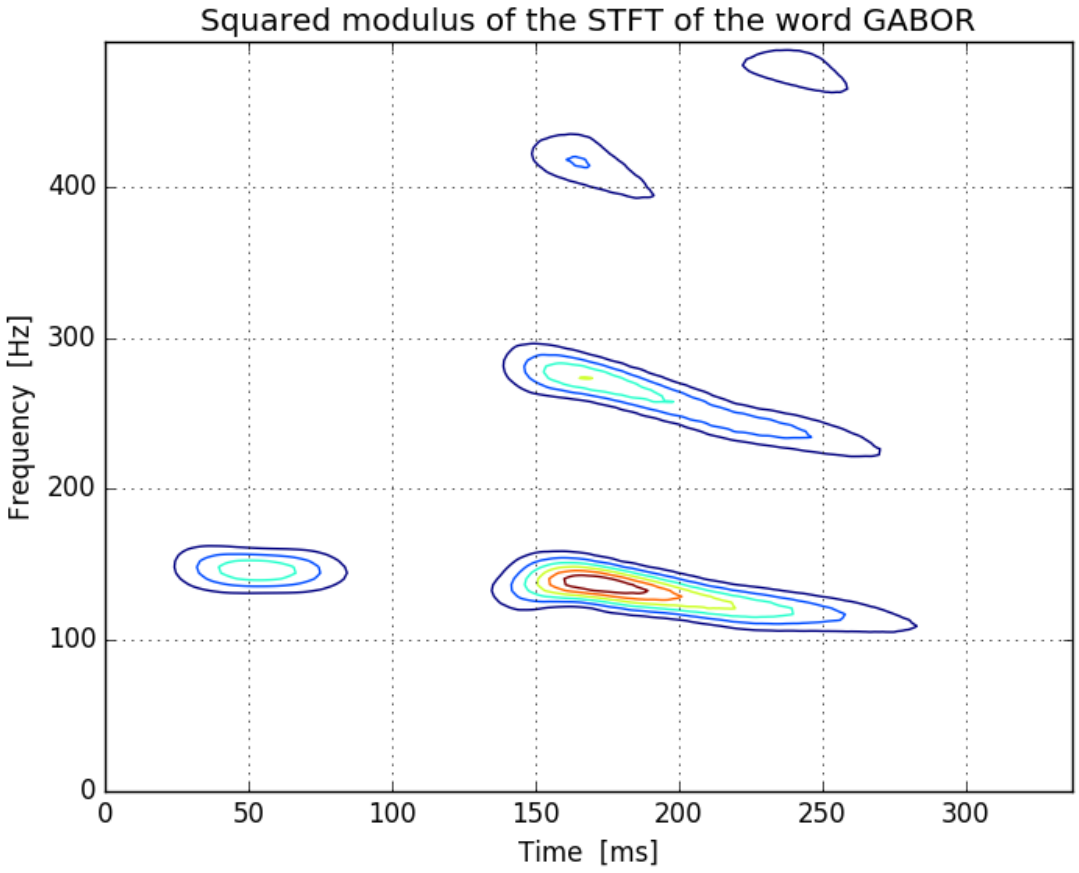


```
from tftb.generators import fmlin, amgauss
from tftb.processing.ambiguity import narrow_band
import numpy as np
import matplotlib.pyplot as plt

n_points = 64
sig1 = fmlin(n_points, 0.2, 0.5)[0] * amgauss(n_points)
sig2 = fmlin(n_points, 0.3, 0)[0] * amgauss(n_points)
sig = np.hstack((sig1, sig2))

tfr, x, y = narrow_band(sig)
plt.contour(2 * x, y, np.abs(tfr) ** 2, 16)
plt.title('Narrow Band ambiguity function')
plt.xlabel('Delay')
plt.ylabel('Doppler')
plt.grid(True)
plt.show()
```

Total running time of the script: (0 minutes 0.091 seconds)

3.1.39 STFT of an Audio Signal

Figure 3.4 from the tutorial.



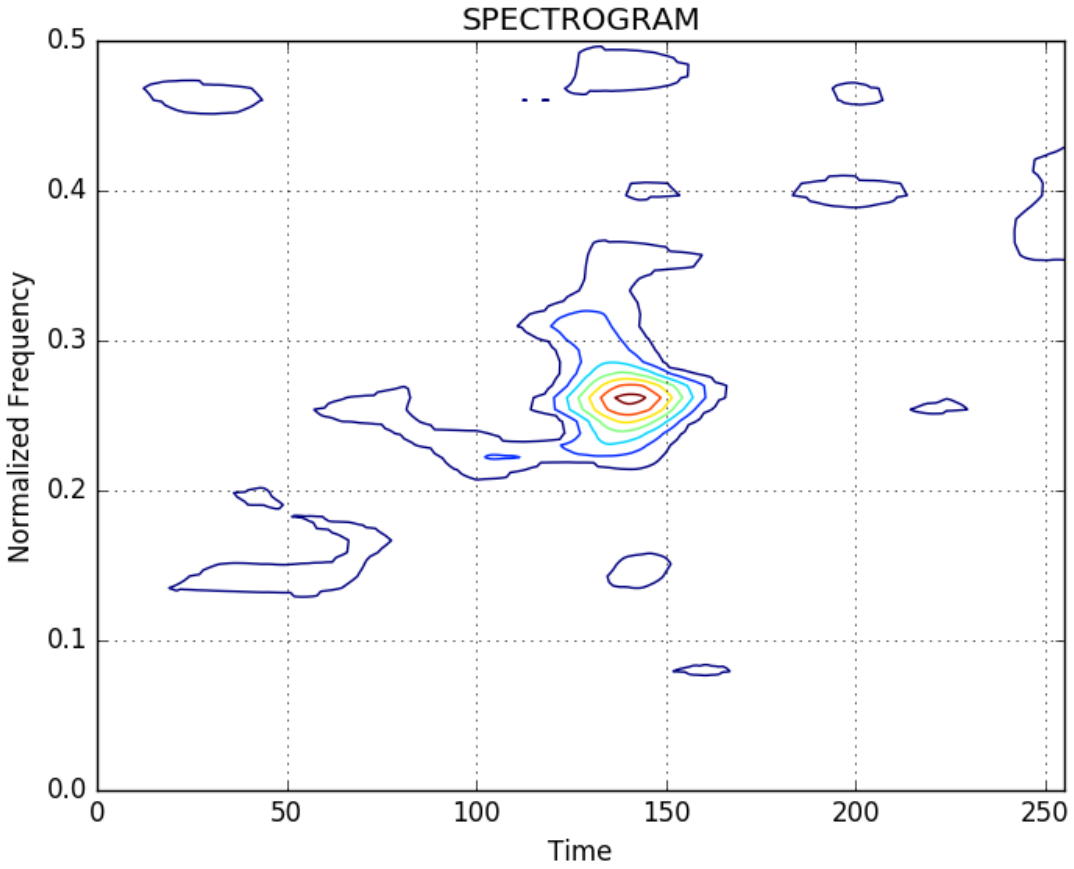
```
from os.path import dirname, abspath, join
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt

DATA_PATH = join(abspath(dirname("__file__")), "data", "gabor.mat")
signal = loadmat(DATA_PATH)['gabor'].ravel()
tfr = loadmat(DATA_PATH)['tfr']
time = np.arange(338)
freq = np.arange(128, dtype=float) / 256.0 * 1000

plt.contour(time, freq, tfr)
plt.grid(True)
plt.xlabel('Time  [ms]')
plt.ylabel('Frequency  [Hz]')
plt.title('Squared modulus of the STFT of the word GABOR')
plt.show()
```

Total running time of the script: (0 minutes 0.142 seconds)

3.1.40 Spectrogram of a Noisy Transient Signal

This example demonstrates the simple use of a Spectrogram to localize a signal in time and frequency. The transient signal appears at the normalized frequency 0.25 and between time points 125 and 160.

Figure 1.11 from the tutorial.



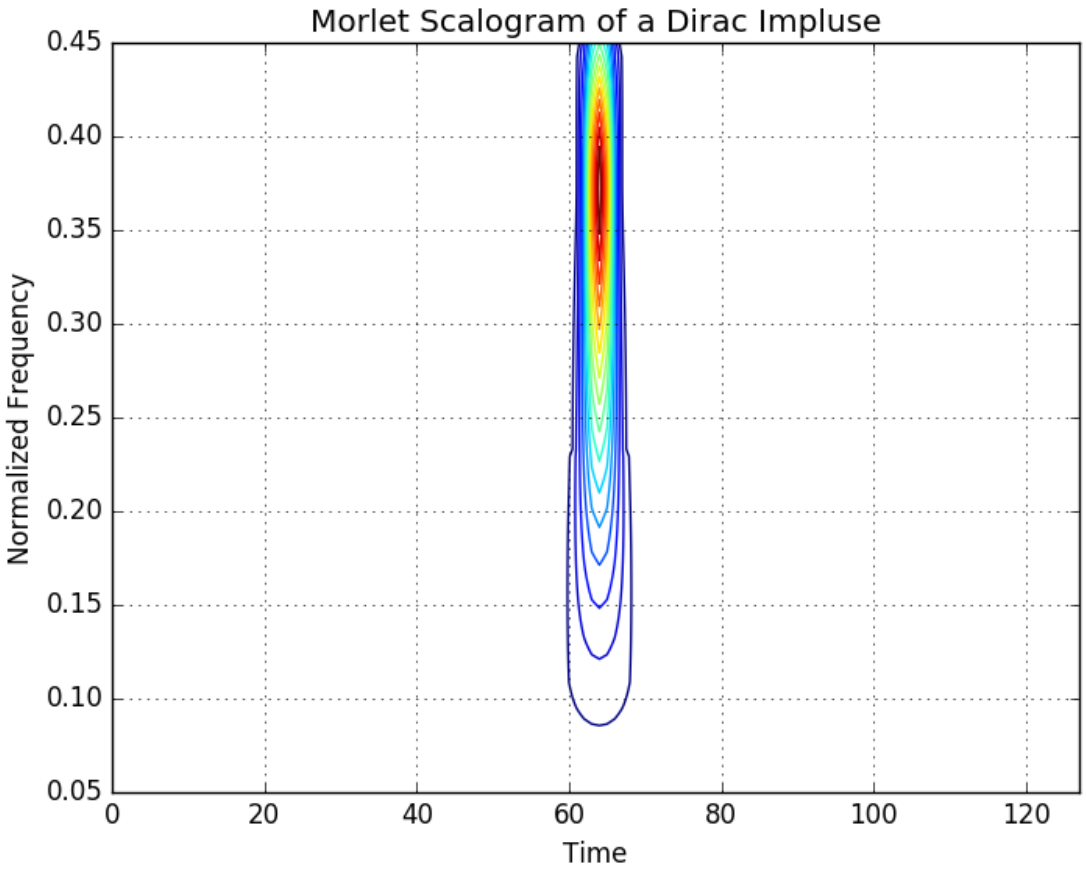

```
import numpy as np
from scipy.signal import hamming
from tftb.generators import amexpos, fmconst, sigmerge, noisecg
from tftb.processing.cohen import Spectrogram

# Generate a noisy transient signal.
transsig = amexpos(64, kind='unilateral') * fmconst(64)[0]
signal = np.hstack((np.zeros((100,)), transsig, np.zeros((92,))))
signal = sigmerge(signal, noisecg(256), -5)

fwindow = hamming(65)
spec = Spectrogram(signal, n_fbins=128, fwindow=fwindow)
spec.run()
spec.plot(kind="contour", threshold=0.1, show_tf=False)
```

Total running time of the script: (0 minutes 0.089 seconds)

3.1.41 Morlet Scalogram of a Dirac Impulse

This example plots the scalogram of a Dirac impulse functions. This shows the behaviour of the scalograms as the scale (or inversely, the frequency) changes. it is well localized for small scales (large frequencies), and less localized as the scale increases (as the frequency decreases).

Figure 3.19 from the tutorial.

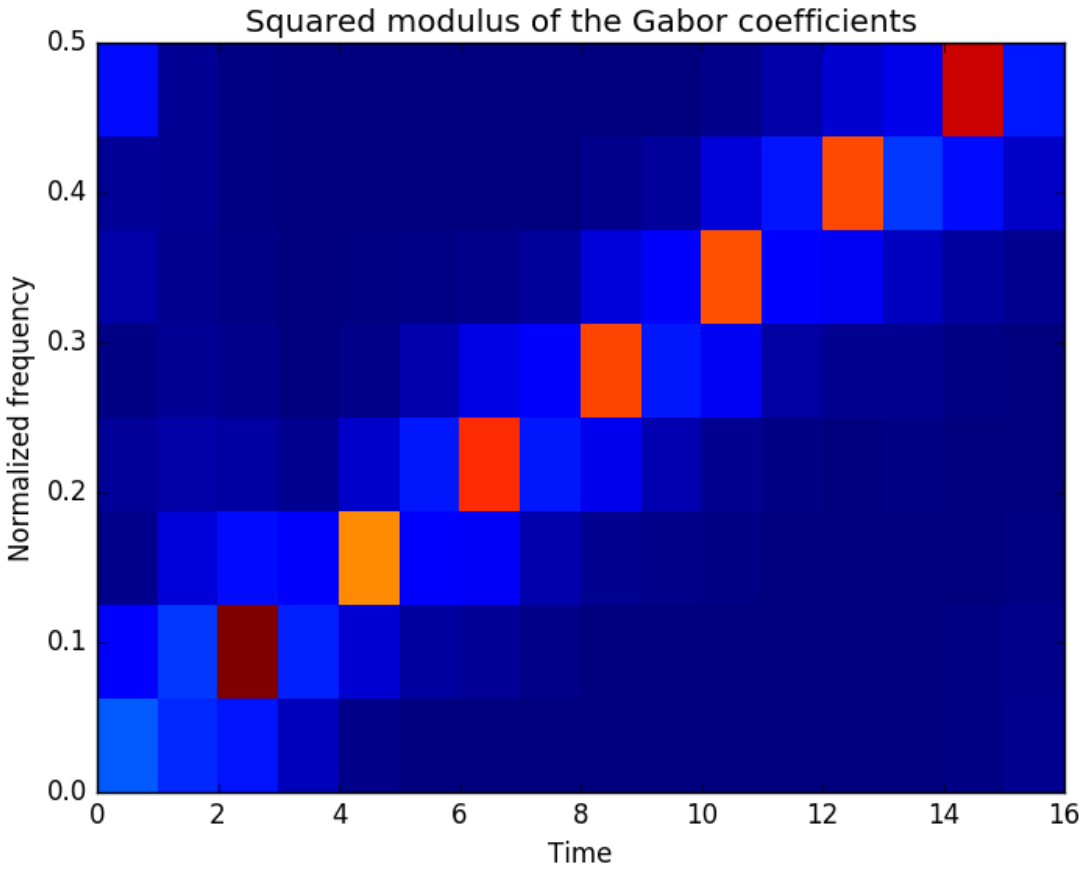



```
from tftb.generators import anapulse
from tftb.processing import Scalogram
import numpy as np
import matplotlib.pyplot as plt

sig1 = anapulse(128)
tfr, t, f, _ = Scalogram(sig1, waveparams=6, fmin=0.05, fmax=0.45,
                         n_voices=128).run()
tfr = np.abs(tfr) ** 2
threshold = np.amax(tfr) * 0.05
tfr[tfr <= threshold] = 0.0
t, f = np.meshgrid(t, f)
plt.contour(t, f, tfr, 20)
plt.grid()
plt.title('Morlet Scalogram of a Dirac Impluse')
plt.xlabel('Time')
plt.ylabel('Normalized Frequency')
plt.show()
```

Total running time of the script: (0 minutes 0.118 seconds)

3.1.42 Gabor Representation of a Chirp

Figure 3.11 from the tutorial.



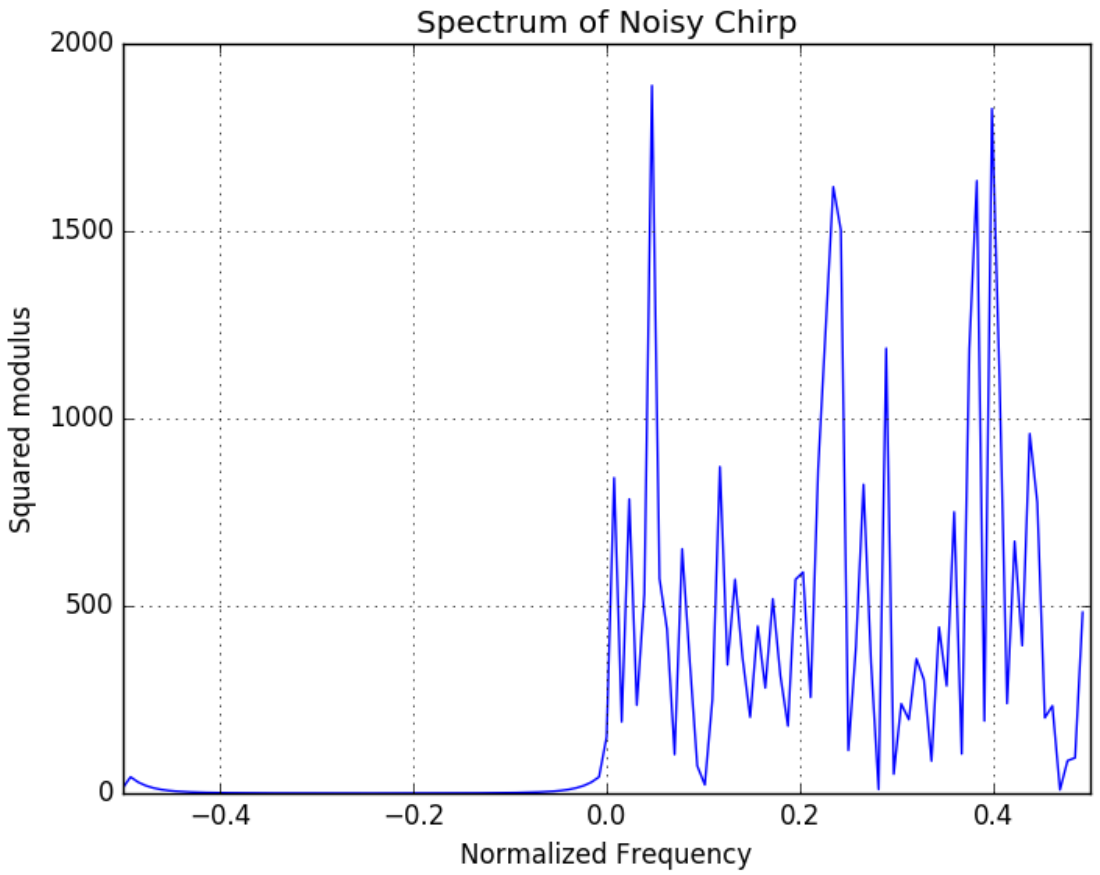

```
from tftb.generators import fmlin
from tftb.processing.linear import gabor
import matplotlib.pyplot as plt
import numpy as np

N1 = 256
Ng = 33
Q = 1
sig = fmlin(N1)[0]
window = np.exp(np.log(0.005) * np.linspace(-1, 1, Ng) ** 2)
window = window / np.linalg.norm(window)
tfr, dgr, h = gabor(sig, 16, Q, window)

plt.imshow(np.flipud(tfr)[8:, :], aspect='auto', extent=[0, 16, 0, 0.5],
           interpolation='none')
plt.xlabel('Time')
plt.ylabel('Normalized frequency')
plt.title('Squared modulus of the Gabor coefficients')
plt.show()
```

Total running time of the script: (0 minutes 0.070 seconds)

3.1.43 Energy Spectrum of a Noisy Chirp

Generate a noisy chirp and plot its energy spectrum.

Figure 1.5 from the tutorial.




```
from tftb.generators import fmlin, sigmerge, noisecg
import matplotlib.pyplot as plt
import numpy as np

# Generate a chirp signal

n_points = 128
fmin, fmax = 0.0, 0.5

signal, _ = fmlin(n_points, fmin, fmax)

# Noisy chirp

noisy_signal = sigmerge(signal, noisecg(128), 0)

# Enery spectrum of the noisy chirp.

dsp1 = np.fft.fftshift(np.abs(np.fft.fft(noisy_signal)) ** 2)
plt.plot(np.arange(-64, 64, dtype=float) / 128.0, dsp1)
plt.xlim(-0.5, 0.5)
```

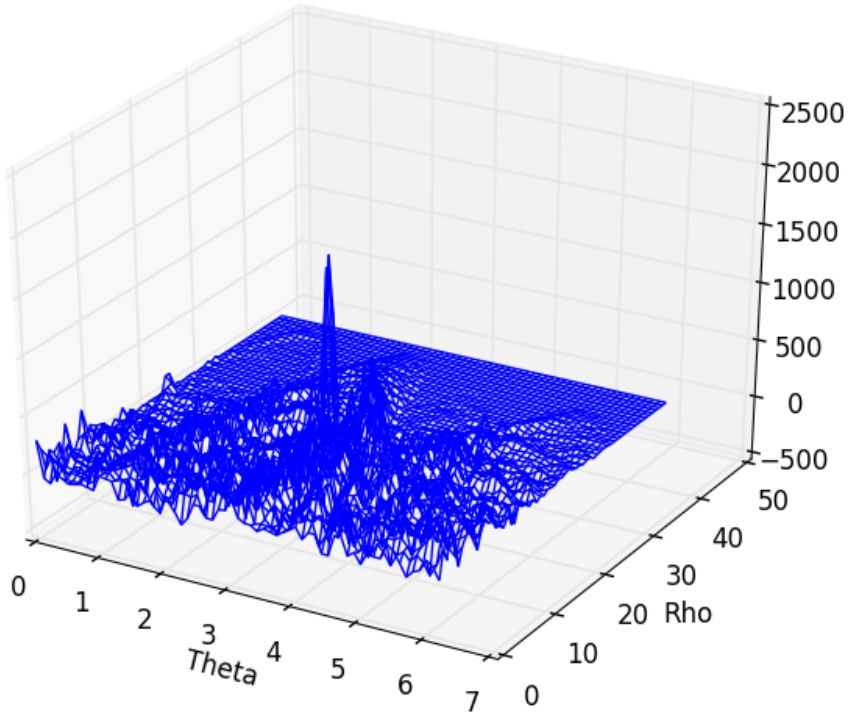
```
plt.title('Spectrum of Noisy Chirp')
plt.ylabel('Squared modulus')
plt.xlabel('Normalized Frequency')
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.210 seconds)

3.1.44 Wigner-Hough Transform of a Chirp

This example demonstrates the use of the Hough transform to extract the estimates of a chirp signal from its Wigner Ville distribution.

Figure 5.4 from the tutorial.




```
import numpy as np
from tftb.generators import noisecg, sigmerge, fmlin
from tftb.processing.cohen import WignerVilleDistribution
from tftb.processing.postprocessing import hough_transform
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

N = 64
sig = sigmerge(fmlin(N, 0, 0.3)[0], noisecg(N), 1)
```

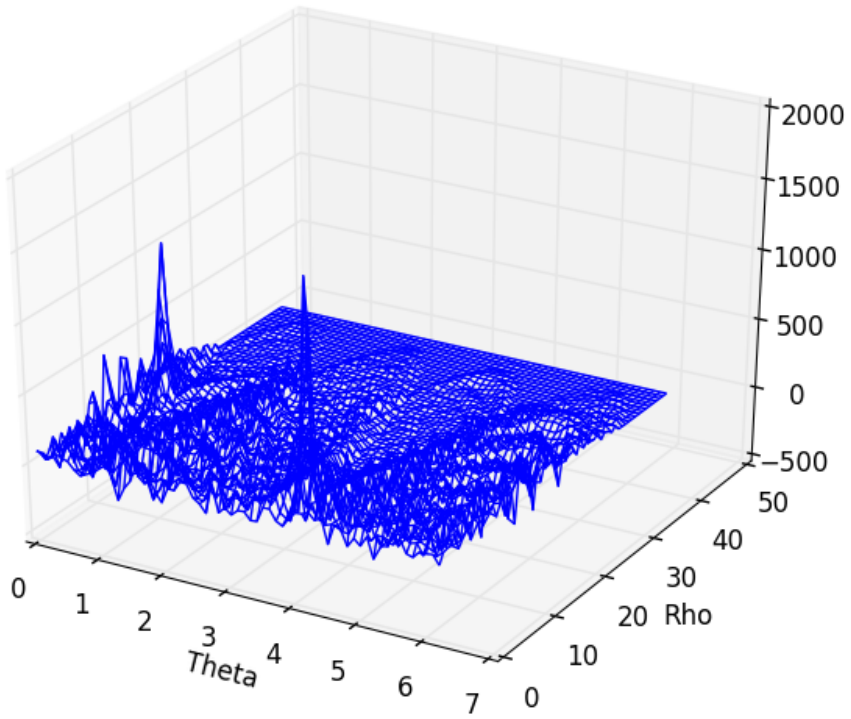
```
tfr, _, _ = WignerVilleDistribution(sig).run()

ht, rho, theta = hough_transform(tfr, N, N)
theta, rho = np.meshgrid(theta, rho)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_wireframe(theta, rho, ht)
ax.set_xlabel('Theta')
ax.set_ylabel('Rho')
plt.show()
```

Total running time of the script: (0 minutes 1.161 seconds)

3.1.45 Hough-Wigner Transform of Two Simultaneous Chirps

Compute the Hough transform of the Wigner-Ville distribution of a signal composed of two chirps. Two peaks corresponding to the two chirps can be seen.

Figure 5.6 from the tutorial.



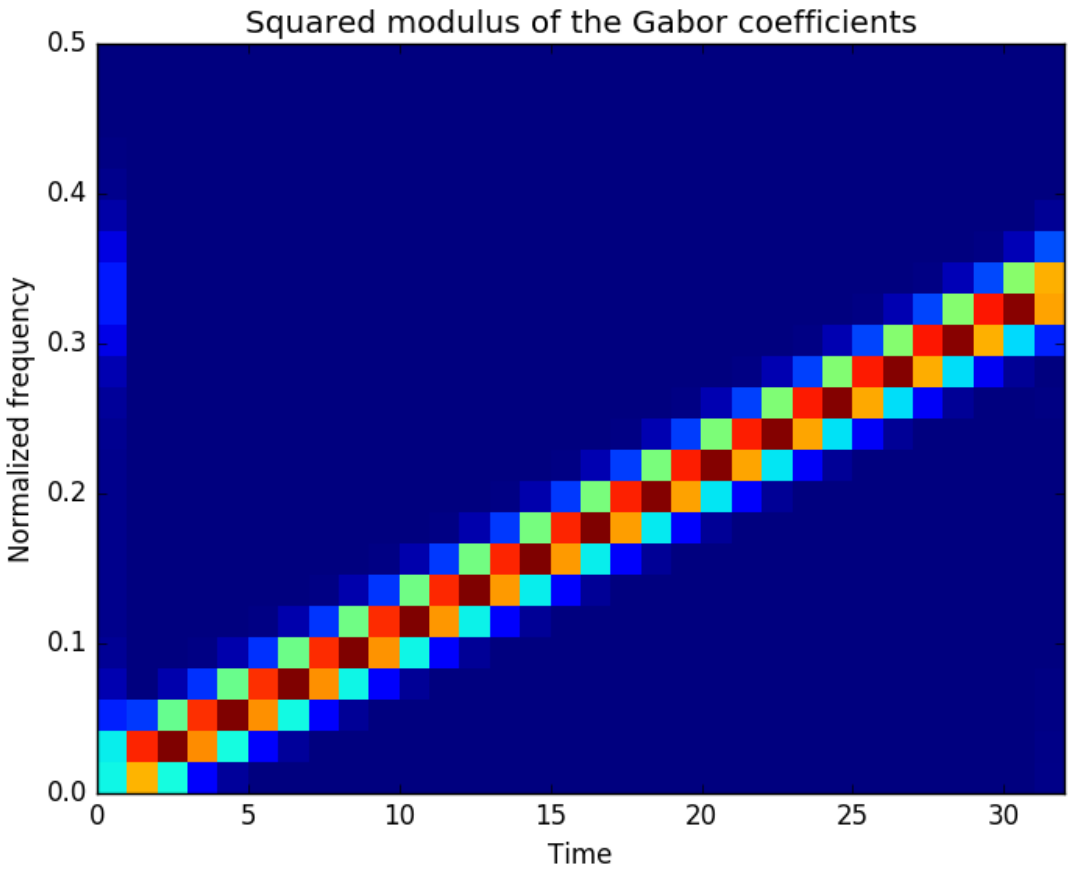

```
from tftb.generators import fmlin, sigmerge
from tftb.processing.cohen import WignerVilleDistribution
from tftb.processing.postprocessing import hough_transform
import numpy as np
```

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

N = 64
sig = sigmerge(fmlin(N, 0, 0.4)[0], fmlin(N, 0.3, 0.5)[0], 1)
tfr, _, _ = WignerVilleDistribution(sig).run()

ht, rho, theta = hough_transform(tfr, N, N)
theta, rho = np.meshgrid(theta, rho)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_wireframe(theta, rho, ht)
ax.set_xlabel('Theta')
ax.set_ylabel('Rho')
plt.show()
```

Total running time of the script: (0 minutes 1.232 seconds)

3.1.46 Gabor Representation of a Chirp with Oversampling

Figure 3.13 from the tutorial.



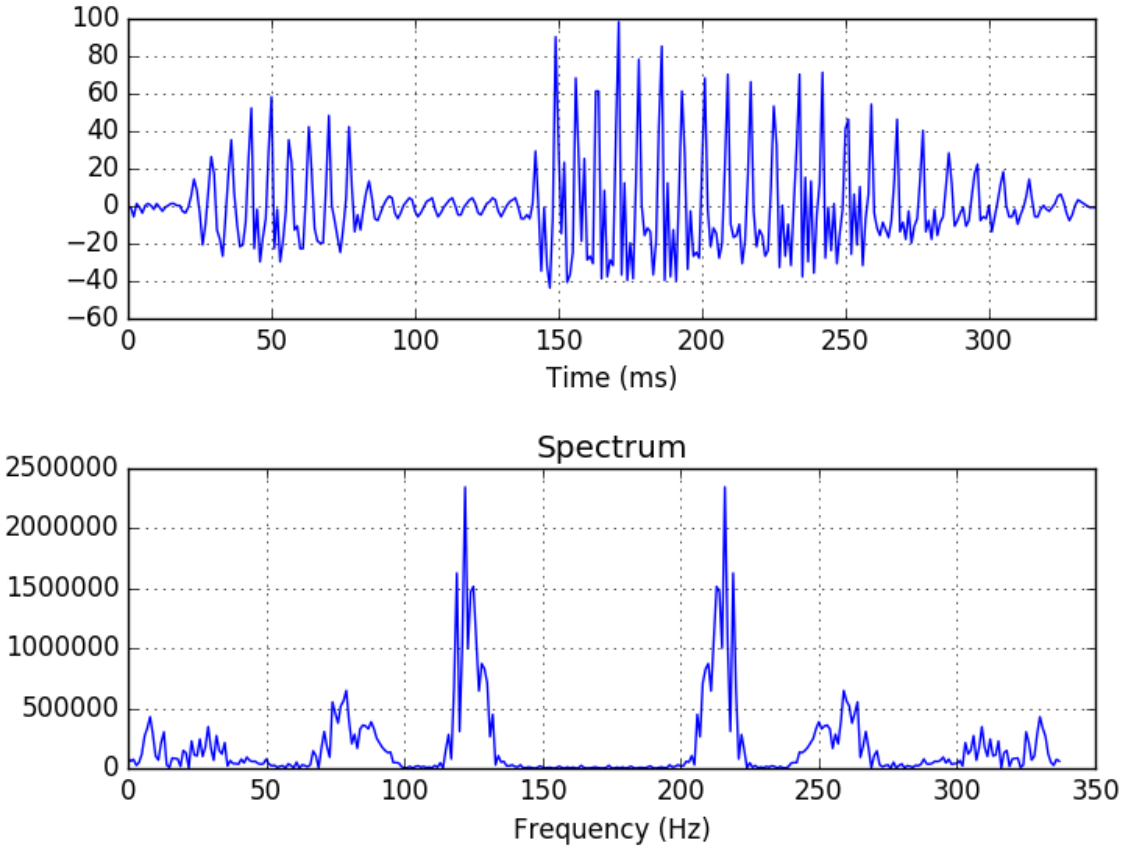

```
from tftb.generators import fmlin
from tftb.processing.linear import gabor
```

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

N1 = 256
Ng = 33
Q = 4
sig = fmlin(N1)[0]
window = np.exp(np.log(0.005) * np.linspace(-1, 1, Ng) ** 2)
window = window / np.linalg.norm(window)
tfr, dgr, h = gabor(sig, 32, Q, window)
time = np.arange(256)
freq = np.linspace(0, 0.5, 128)
plt.imshow(np.flipud(tfr)[8:, :], aspect='auto', extent=[0, 32, 0, 0.5],
           interpolation='none')
plt.xlabel('Time')
plt.ylabel('Normalized frequency')
plt.title('Squared modulus of the Gabor coefficients')
plt.show()
```

Total running time of the script: (0 minutes 0.068 seconds)

3.1.47 Energy Spectrum of an Audio Signal

Figure 3.3 from the tutorial.



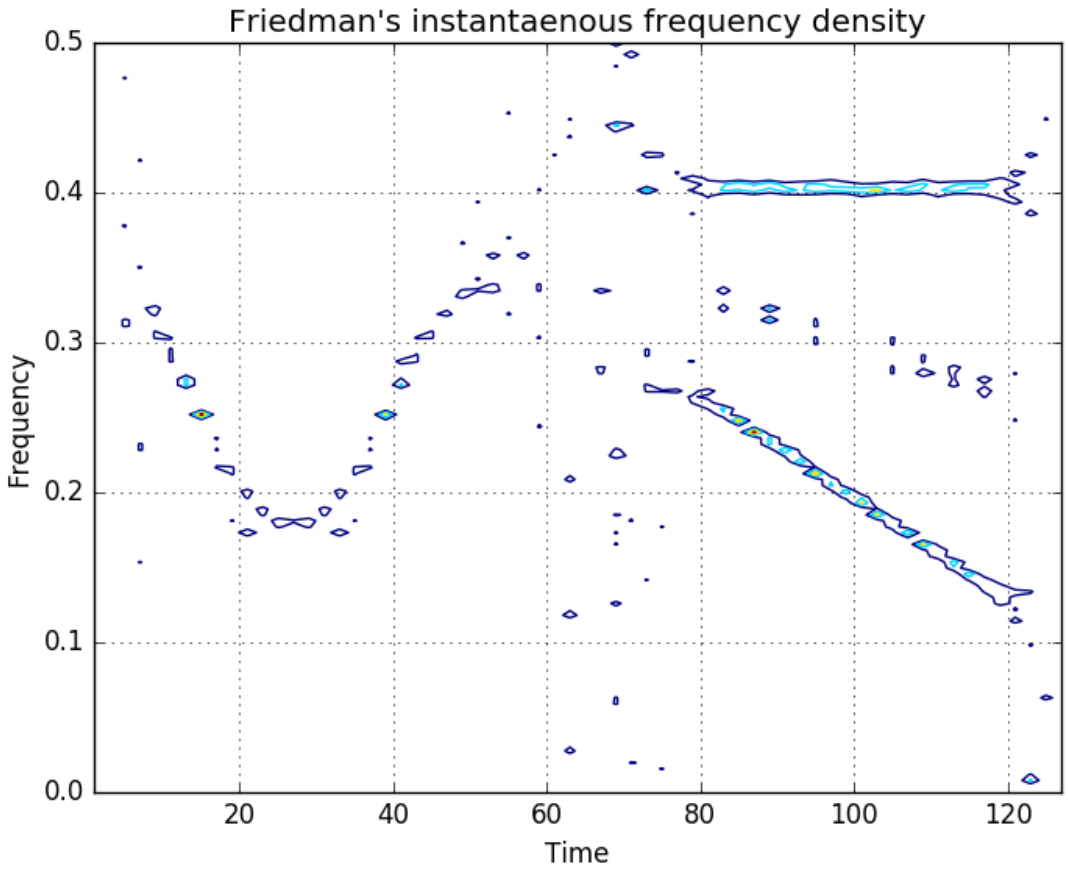
```
from os.path import dirname, abspath, join
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt

DATA_PATH = join(abspath(dirname("__file__")), "data", "gabor.mat")
signal = loadmat(DATA_PATH)['gabor'].ravel()
time = np.arange(338)
dsp = np.fft.fftshift(np.abs(np.fft.fft(signal)) ** 2)
freq = np.arange(-169, 169, dtype=float) / 338 * 1000

plt.subplot(211)
plt.plot(time, signal)
plt.grid(True)
plt.xlim(0, time.max())
plt.xlabel('Time (ms)')

plt.subplot(212)
plt.plot(dsp)
plt.grid(True)
plt.title('Spectrum')
plt.xlabel('Frequency (Hz)')

plt.subplots_adjust(hspace=0.5)

plt.show()
```

Total running time of the script: (0 minutes 0.125 seconds)

3.1.48 Friedman's Instantaneous Frequency Density Calculation

This example uses Friedman's method to calculate the instantaneous frequency density of a hybrid signal. The method consists of computing the histograms of frequency displacements of the spectrogram of the signal.

Figure 4.38 from the tutorial.




```
import numpy as np
import matplotlib.pyplot as plt
from tftb.generators import fmlin, fmsin, fmconst
from tftb.processing.reassigned import pseudo_wigner_ville
from tftb.processing.postprocessing import friedman_density

sig1, if1 = fmsin(60, 0.16, 0.35, 50, 1, 0.35, 1)
sig2, if2 = fmlin(60, 0.3, 0.1)
sig3, if3 = fmconst(60, 0.4)
sig = np.hstack((sig1, np.zeros((8,)), sig2 + sig3))

t = np.arange(1, 128, step=2)
tfr, rtfr, hat = pseudo_wigner_ville(sig, timestamps=t)
tifd = friedman_density(tfr, hat, t)
f = np.linspace(0, 0.5, tifd.shape[0])

plt.contour(t, f, tifd, 4)
plt.grid(True)
plt.title("Friedman's instantaenous frequency density")
plt.xlabel('Time')
plt.ylabel('Frequency')
plt.show()
```

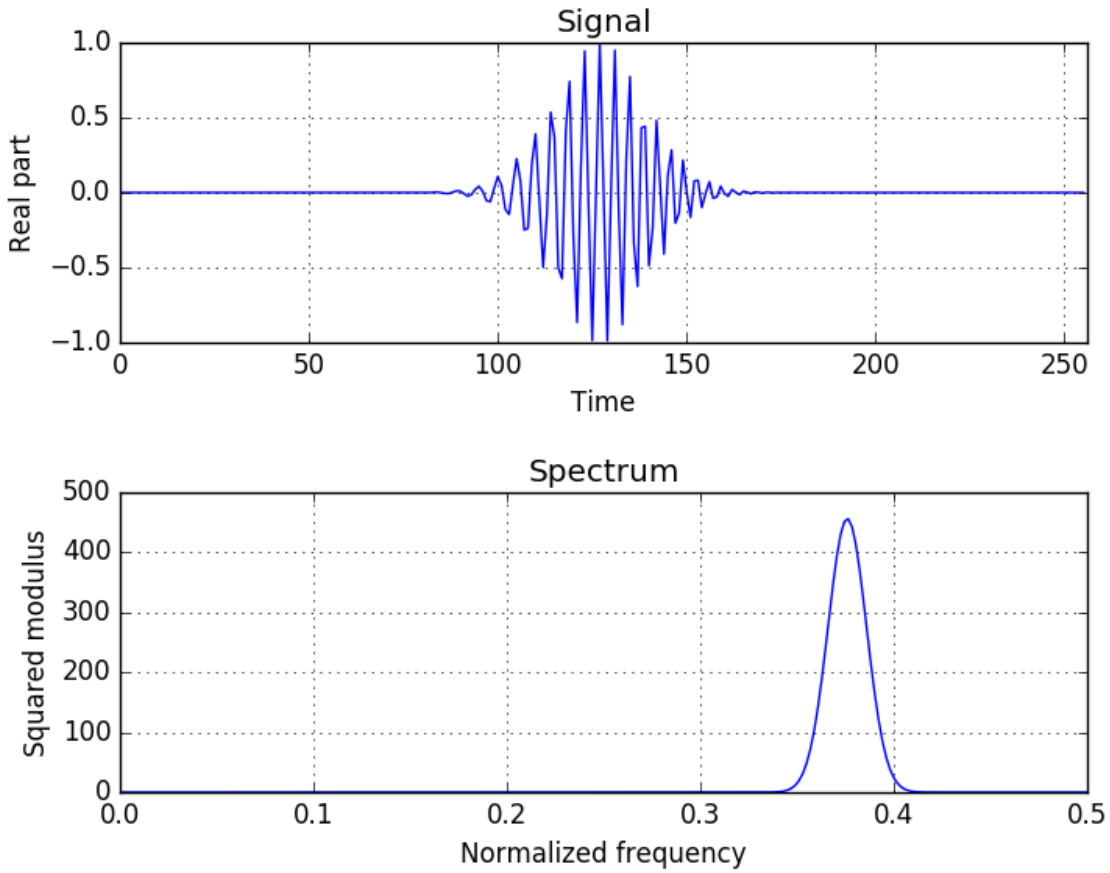
Total running time of the script: (0 minutes 0.191 seconds)

3.1.49 Time and Frequency Localization Characteristics

Generate a signal that has localized characteristics in both time and frequency and compute the following estimates:

- time center
- time duration
- frequency center
- frequency spreading

Example 2.1 from the tutorial.



Out:

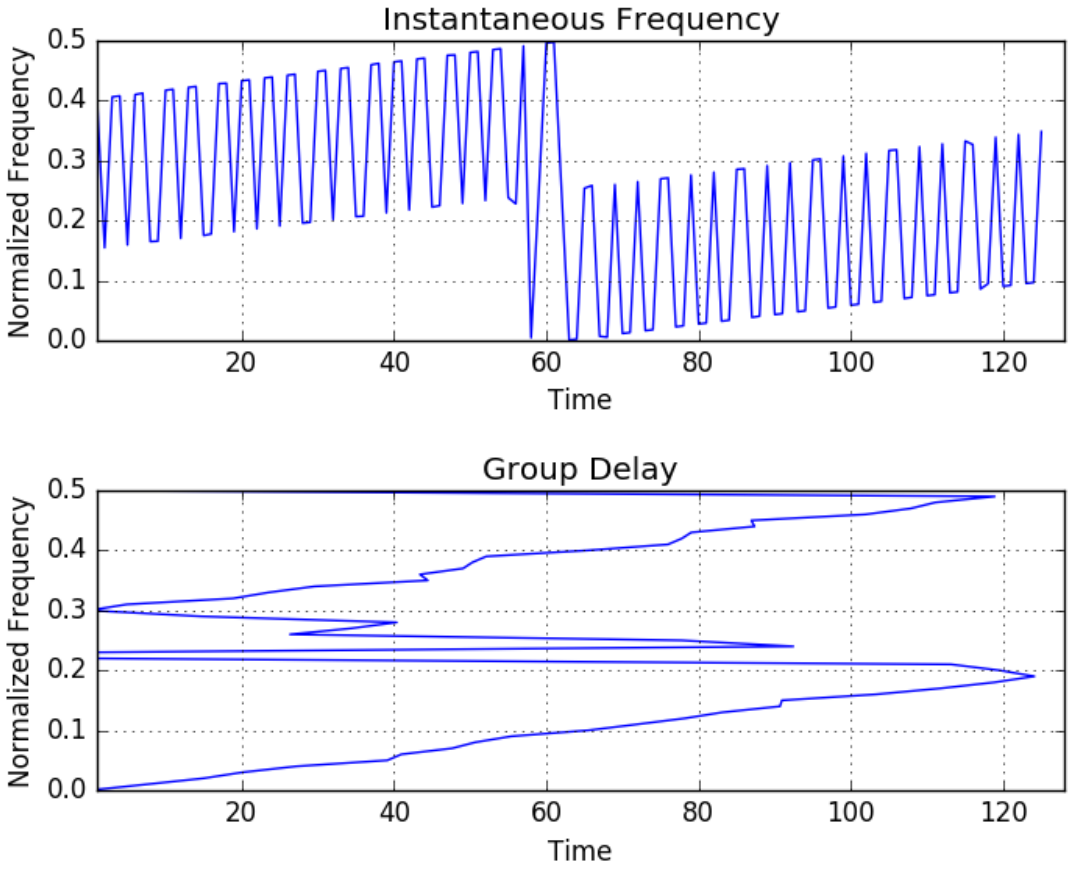
```
Time Center: 127.0
Time Duration: 32.0
Frequency Center: 0.249019607843
Frequency Spreading: 0.0700964323482
```

```
from tftb.generators import fmlin, amgauss
from tftb.processing import loctime, locfreq
import numpy as np
import matplotlib.pyplot as plt

# generate signal
signal = fmlin(256)[0] * amgauss(256)
plt.subplot(211), plt.plot(np.real(signal))
plt.xlim(0, 256)
plt.xlabel('Time')
plt.ylabel('Real part')
plt.title('Signal')
plt.grid()
fsig = np.fft.fftshift(np.abs(np.fft.fft(signal)) ** 2)
plt.subplot(212), plt.plot(np.linspace(0, 0.5, 256), fsig)
plt.xlabel('Normalized frequency')
plt.ylabel('Squared modulus')
plt.title('Spectrum')
plt.grid()
plt.subplots_adjust(hspace=0.5)
plt.show()


tm, T = loctime(signal)
print("Time Center: {}".format(tm))
print("Time Duration: {}".format(T))
num, B = locfreq(signal)
print("Frequency Center: {}".format(num))
print("Frequency Spreading: {}".format(B))
```

Total running time of the script: (0 minutes 0.194 seconds)

3.1.50 Instantatneous Frequency and Group Delay Estimation of a Multi-Component Nonstationary Signal

Figure 2.10 from the tutorial.




```
#  N=128; x1=fmlin(N,0,0.2); x2=fmlin(N,0.3,0.5);
#  x=x1+x2;
#  ifr=instfreq(x); subplot(211); plot(ifr);
#  fn=0:0.01:0.5; gd=sgrpdlay(x,fn);
#  subplot(212); plot(gd,fn);
from tftb.generators import fmlin
from tftb.processing import inst_freq, group_delay
import matplotlib.pyplot as plt
import numpy as np

N = 128
x1, _ = fmlin(N, 0, 0.2)
x2, _ = fmlin(N, 0.3, 0.5)
x = x1 + x2
ifr = inst_freq(x)[0]
fn = np.arange(0.51, step=0.01)
gd = group_delay(x, fn)

plt.subplot(211)
plt.plot(ifr)
plt.xlim(1, N)
plt.grid(True)
plt.title('Instantaneous Frequency')
plt.xlabel('Time')
plt.ylabel('Normalized Frequency')
```

```
plt.subplot(212)
plt.plot(gd, fn)
plt.xlim(1, N)
plt.grid(True)
plt.title('Group Delay')
plt.xlabel('Time')
plt.ylabel('Normalized Frequency')

plt.subplots_adjust(hspace=0.5)

plt.show()
```

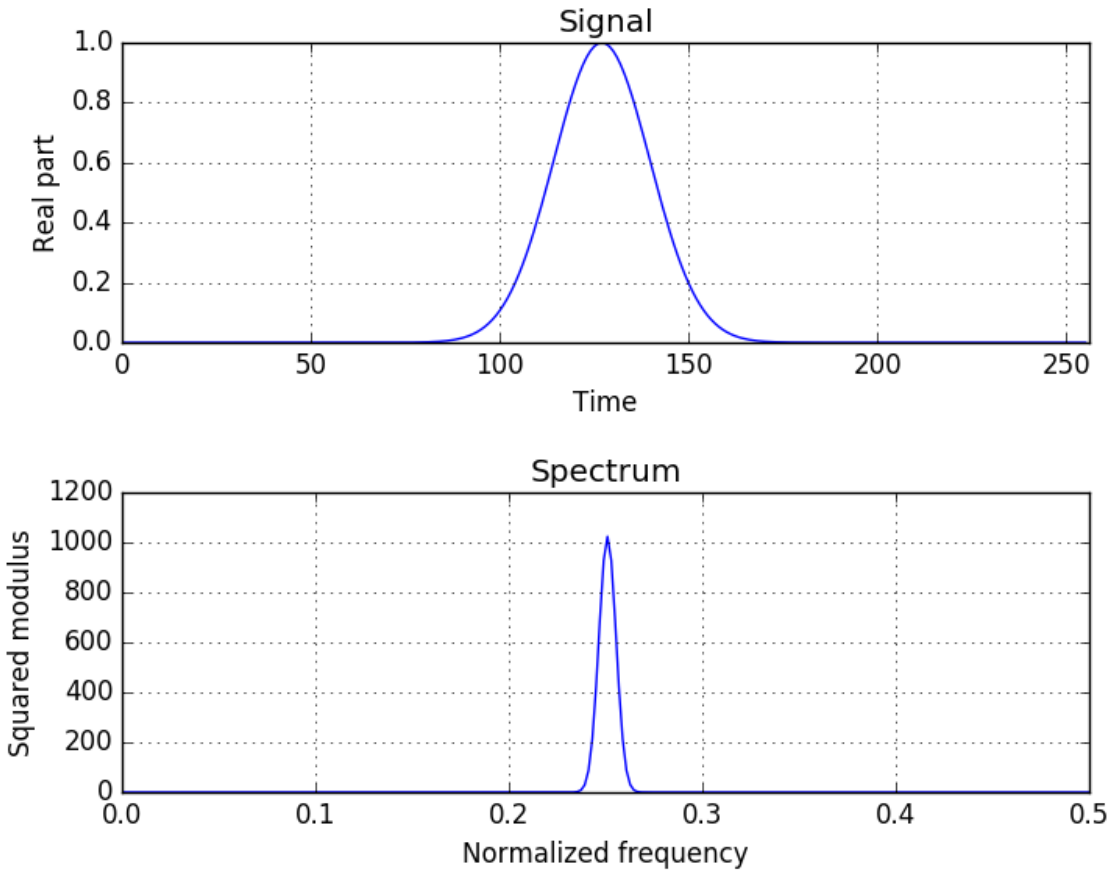
Total running time of the script: (0 minutes 0.123 seconds)

3.1.51 Heisenbeg-Gabor Inequality

This example demonstrates the Heisenberg-Gabor inequality.

Simply put, the inequality states that the time-bandwidth product of a signal is lower bound by some constant (in this case normalized to unity). This means that a signal cannot have arbitrarily high precision in time and frequency simultaneously.

Figure 2.2 from the tutorial.



Out:

```
Time Center: 127.0
Time Duration: 32.0
Frequency Center: 5.20417042793e-18
Frequency Spreading: 0.03125
Time-bandwidth product: 1.0
```

```
from tftb.generators import amgauss
from tftb.processing import loctime, locfreq
import numpy as np
import matplotlib.pyplot as plt

# generate signal
signal = amgauss(256)
plt.subplot(211), plt.plot(np.real(signal))
plt.xlim(0, 256)
plt.xlabel('Time')
plt.ylabel('Real part')
plt.title('Signal')
plt.grid()
fsig = np.fft.fftshift(np.abs(np.fft.fft(signal)) ** 2)
plt.subplot(212), plt.plot(np.linspace(0, 0.5, 256), fsig)
plt.xlabel('Normalized frequency')
plt.ylabel('Squared modulus')
plt.title('Spectrum')
plt.grid()
plt.subplots_adjust(hspace=0.5)
plt.show()

tm, T = loctime(signal)
print("Time Center: {}".format(tm))
print("Time Duration: {}".format(T))
fm, B = locfreq(signal)
print("Frequency Center: {}".format(fm))
print("Frequency Spreading: {}".format(B))
print("Time-bandwidth product: {}".format(T * B))
```

Total running time of the script: (0 minutes 0.161 seconds)

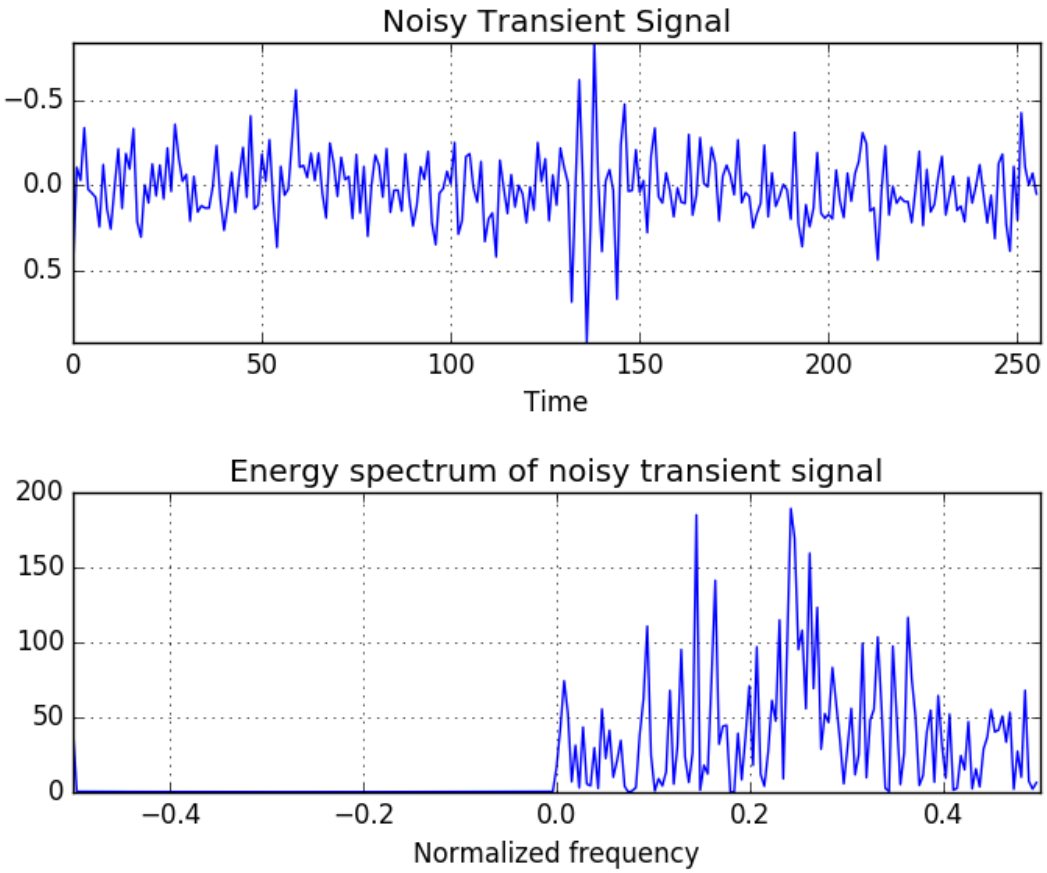
3.1.52 Spectrum of a Noisy Transient Signal

This example shows how to generate a noisy transient signal with the following characteristics:

- One-sided exponential amplitude modulation (See amexpos)
- Constant frequency modulation (See fmconst)
- -5 dB complex gaussian noise (See noisecg and sigmerge)

And how to plot its energy spectrum.

Figure 1.10 of the tutorial.




```
import numpy as np
import matplotlib.pyplot as plt
from tftb.generators import amexpos, fmconst, sigmerge, noisecg

# Generate a noisy transient signal.
transsig = amexpos(64, kind='unilateral') * fmconst(64)[0]
signal = np.hstack((np.zeros((100,)), transsig, np.zeros((92,))))
signal = sigmerge(signal, noisecg(256), -5)
fig, ax = plt.subplots(2, 1)
ax1, ax2 = ax
ax1.plot(np.real(signal))
ax1.grid()
ax1.set_title('Noisy Transient Signal')
ax1.set_xlabel('Time')
ax1.set_xlim((0, 256))
ax1.set_ylim((np.real(signal).max(), np.real(signal.min())))

# Energy spectrum of the signal
dsp = np.fft.fftshift(np.abs(np.fft.fft(signal)) ** 2)
ax2.plot(np.arange(-128, 128, dtype=float) / 256, dsp)
ax2.set_title('Energy spectrum of noisy transient signal')
ax2.set_xlabel('Normalized frequency')
ax2.grid()
ax2.set_xlim(-0.5, 0.5)

plt.subplots_adjust(hspace=0.5)
```

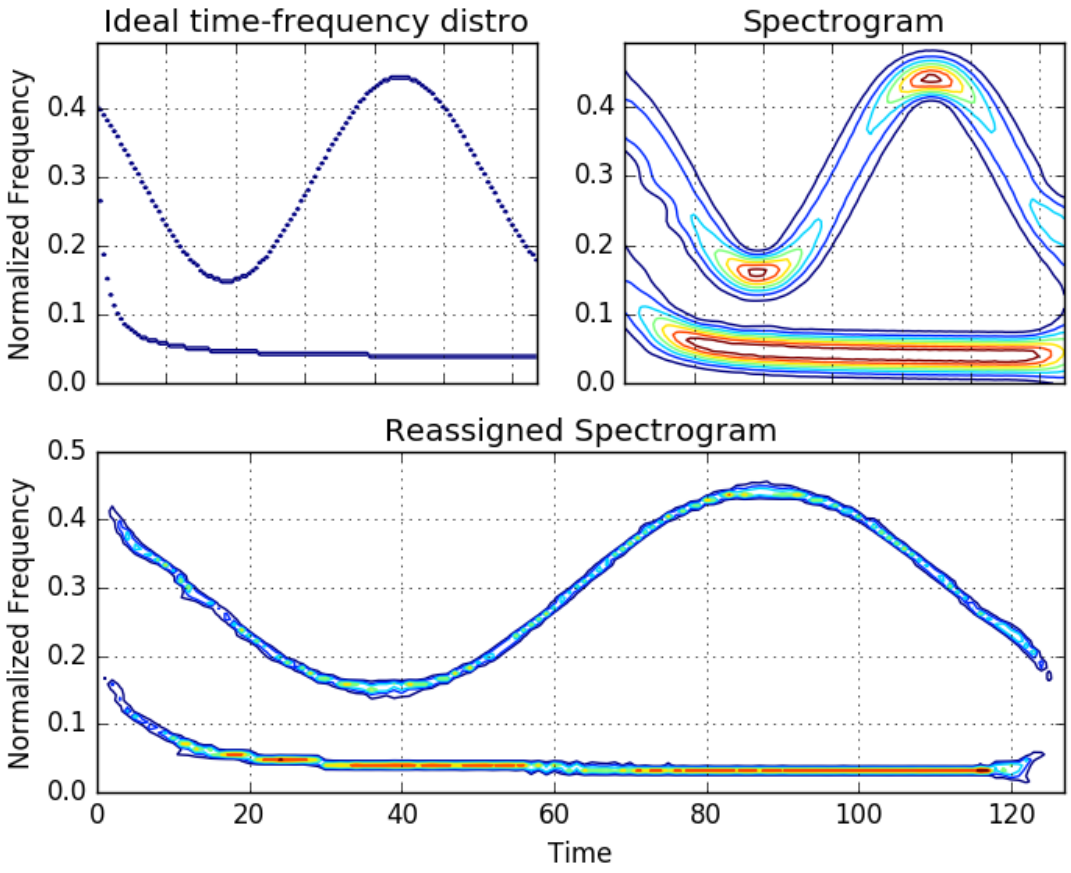
```
plt.show()
```

Total running time of the script: (0 minutes 0.133 seconds)

3.1.53 Comparison of a Spectrogram and a Reassigned Spectrogram

This example compares the spectrogram and the reassigned spectrogram of a hybrid signal (containing sinusoidal, constant and linear frequency modulations), against its ideal time-frequency characteristics.

Figure 4.34 from the tutorial.




```
from tftb.generators import fmsin, fmhyp
from tftb.processing import ideal_tfr, reassigned_spectrogram, Spectrogram
import numpy as np
import matplotlib.pyplot as plt

n_points = 128
sig1, if1 = fmsin(n_points, 0.15, 0.45, 100, 1, 0.4, -1)
sig2, if2 = fmhyp(n_points, [1, .5], [32, 0.05])
sig = sig1 + sig2
ideal, t, f = ideal_tfr(np.vstack((if1, if2)))
_, re_spec, _ = reassigned_spectrogram(sig)
spec, t3, f3 = Spectrogram(sig).run()
```

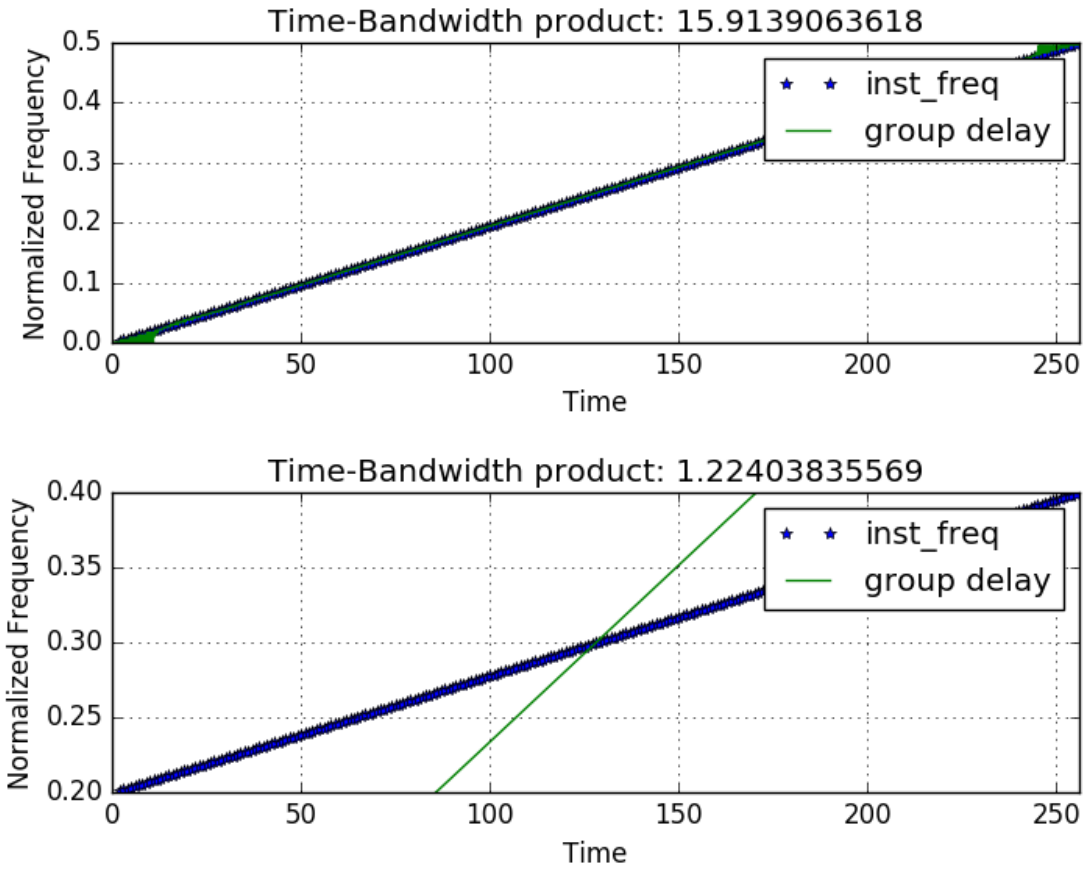
```
# Ideal tfr
plt.subplot(221)
plt.contour(t, f, ideal, 1)
plt.grid(True)
plt.gca().set_xticklabels([])
plt.title("Ideal time-frequency distro")
plt.ylabel('Normalized Frequency')

# Spectrogram
plt.subplot(222)
plt.contour(t3, f3[:64], spec[:64, :])
plt.grid(True)
plt.gca().set_xticklabels([])
plt.title("Spectrogram")

# Reassigned Spectrogram
plt.subplot(212)
f = np.linspace(0, 0.5, 64)
plt.contour(np.arange(128), f, re_spec[:64, :])
plt.grid(True)
plt.title("Reassigned Spectrogram")
plt.xlabel('Time')
plt.ylabel('Normalized Frequency')

plt.show()
```

Total running time of the script: (0 minutes 1.198 seconds)

3.1.54 Comparison of Instantaneous Frequency and Group Delay

Instantaneous frequency and group delay are very closely related. The former is the frequency of a signal at a given instant, and the latter is the time delay of frequency components. As this example shows, they coincide with each other for a given signal when the time bandwidth product of the signal is sufficiently high.

Figure 2.5 from the tutorial.




```
import numpy as np
import matplotlib.pyplot as plt
from tftb.generators import amgauss, fmlin
from tftb.processing import loctime, locfreq, inst_freq, group_delay

time_instants = np.arange(2, 256)
sig1 = amgauss(256, 128, 90) * fmlin(256)[0]
tm, T1 = loctime(sig1)
fm, B1 = locfreq(sig1)
ifr1 = inst_freq(sig1, time_instants)[0]
f1 = np.linspace(0, 0.5 - 1.0 / 256, 256)
gd1 = group_delay(sig1, f1)

plt.subplot(211)
plt.plot(time_instants, ifr1, '*', label='inst_freq')
plt.plot(gd1, f1, '-', label='group delay')
plt.xlim(0, 256)
plt.grid(True)
plt.legend()
plt.title("Time-Bandwidth product: {0}".format(T1 * B1))
plt.xlabel('Time')
plt.ylabel('Normalized Frequency')


sig2 = amgauss(256, 128, 30) * fmlin(256, 0.2, 0.4)[0]
tm, T2 = loctime(sig2)
```

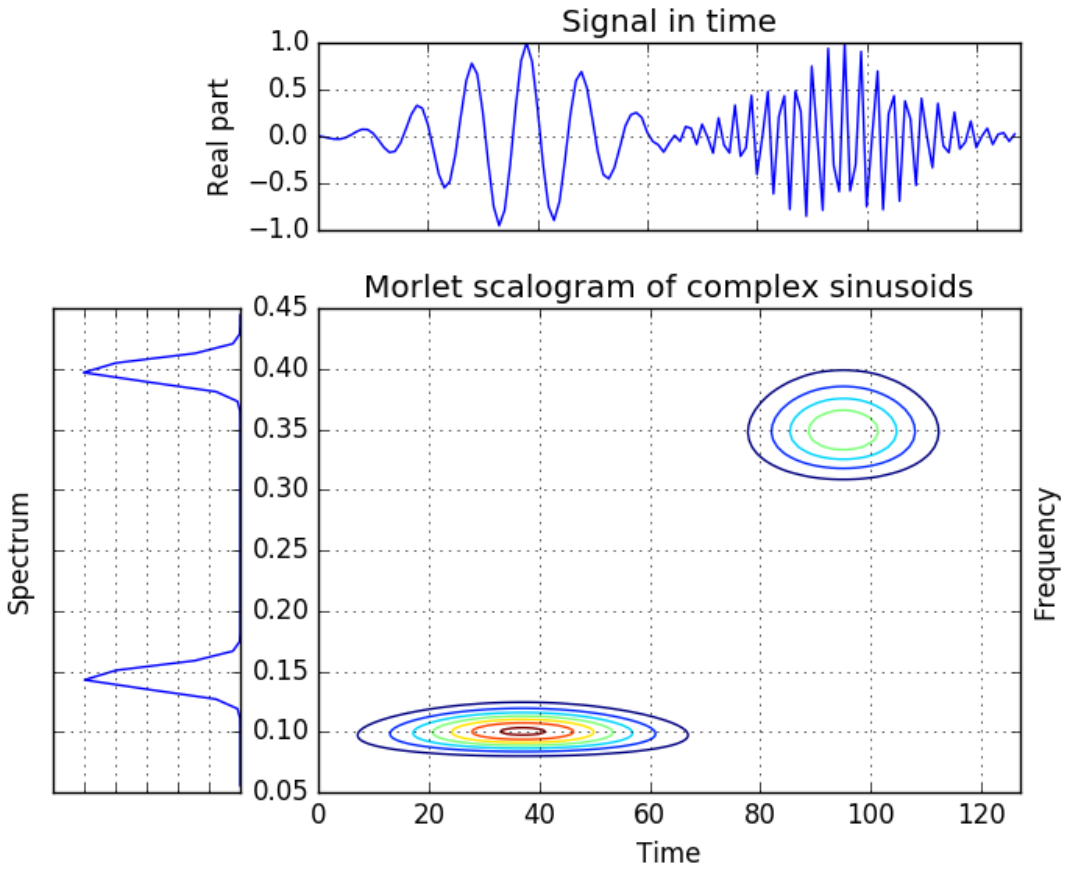
```
fm, B2 = locfreq(sig2)
ifr2 = inst_freq(sig2, time_instants)[0]
f2 = np.linspace(0.02, 0.4, 256)
gd2 = group_delay(sig2, f2)


plt.subplot(212)
plt.plot(time_instants, ifr2, '*', label='inst_freq')
plt.plot(gd2, f2, '-', label='group delay')
plt.ylim(0.2, 0.4)
plt.xlim(0, 256)
plt.grid(True)
plt.legend()
plt.title("Time-Bandwidth product: {0}".format(T2 * B2))
plt.xlabel('Time')
plt.ylabel('Normalized Frequency')

plt.subplots_adjust(hspace=0.5)

plt.show()
```

Total running time of the script: (0 minutes 0.236 seconds)

3.1.55 Morlet Scalogram of Gaussian Atoms

This example demonstrates the effect of frequency-dependent smoothing that is accomplished in a Morlet scalogram. Note that the localization at lower frequencies is much better.

Figure 4.18 from the tutorial.




```
from tftb.processing import Scalogram
from tftb.generators import atoms
import numpy as np
from mpl_toolkits.axes_grid1 import make_axes_locatable
import matplotlib.pyplot as plt

sig = atoms(128, np.array([[38, 0.1, 32, 1], [96, 0.35, 32, 1]]))
tfr, t, freqs, _ = Scalogram(sig, fmin=0.05, fmax=0.45,
                             time_instants=np.arange(1, 129)).run()
t, f = np.meshgrid(t, freqs)

fig, axContour = plt.subplots()
axContour.contour(t, f, tfr)
axContour.grid(True)
axContour.set_title("Morlet scalogram of complex sinusoids")
axContour.set_ylabel('Frequency')
axContour.yaxis.set_label_position('right')
axContour.set_xlabel('Time')

divider = make_axes_locatable(axContour)
axTime = divider.append_axes("top", 1.2, pad=0.5)
axFreq = divider.append_axes("left", 1.2, pad=0.5)
axTime.plot(np.real(sig))
axTime.set_xticklabels([])
axTime.set_xlim(0, 128)
axTime.set_ylabel('Real part')
```

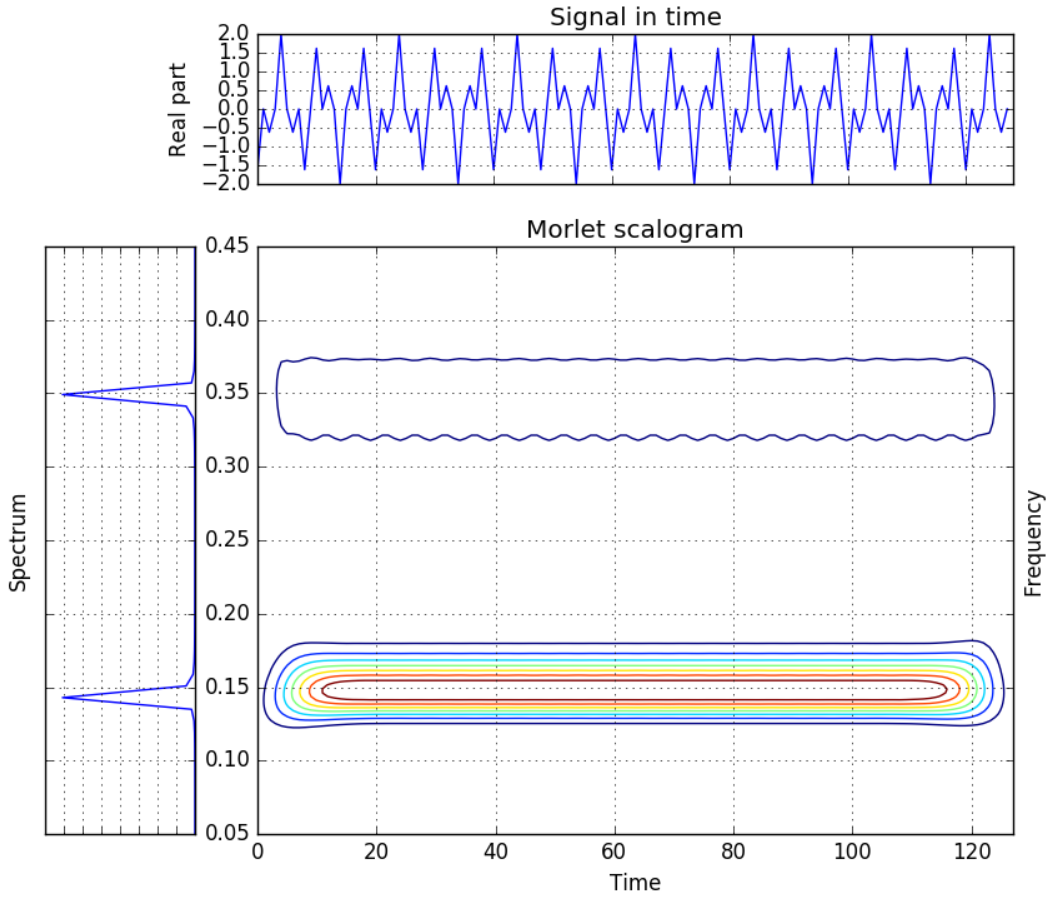
```
axTime.set_title('Signal in time')
axTime.grid(True)
freq_y = np.linspace(0, 0.5, sig.shape[0] / 2)
freq_x = (abs(np.fft.fftshift(np.fft.fft(sig))) ** 2)[::-1][:64]
ix = np.logical_and(freq_y >= 0.05, freq_y <= 0.45)
axFreq.plot(freq_x[ix], freq_y[ix])
# axFreq.set_ylim(0.05, 0.45)
axFreq.set_yticklabels([])
axFreq.set_xticklabels([])
axFreq.grid(True)
axFreq.set_ylabel('Spectrum')
axFreq.invert_xaxis()
axFreq.grid(True)
plt.show()
```

Total running time of the script: (0 minutes 0.257 seconds)

3.1.56 Morlet Scalogram of a Multicomponent Signal

This example demonstrates the visualization of the Morlet scalogram of a signal containing two complex sinusoids. In a scalogram, the frequency resolution varies on the scale of the signal. Here, the frequency resolution decreases at higher frequencies (lower scale).

Figure 3.20 from the tutorial.




```
from tftb.processing import Scalogram
from tftb.generators import fmconst
import numpy as np
from mpl_toolkits.axes_grid1 import make_axes_locatable
import matplotlib.pyplot as plt

sig2 = fmconst(128, .15)[0] + fmconst(128, .35)[0]
tfr, t, freqs, _ = Scalogram(sig2, time_instants=np.arange(1, 129), waveparams=6,
                             fmin=0.05, fmax=0.45, n_voices=128).run()
tfr = np.abs(tfr) ** 2
threshold = np.amax(tfr) * 0.05
tfr[tfr <= threshold] = 0.0
t, f = np.meshgrid(t, freqs)

fig, axContour = plt.subplots(figsize=(10, 8))
axContour.contour(t, f, tfr)
axContour.grid(True)
axContour.set_title("Morlet scalogram")
axContour.set_ylabel('Frequency')
axContour.yaxis.set_label_position('right')
axContour.set_xlabel('Time')

divider = make_axes_locatable(axContour)
```

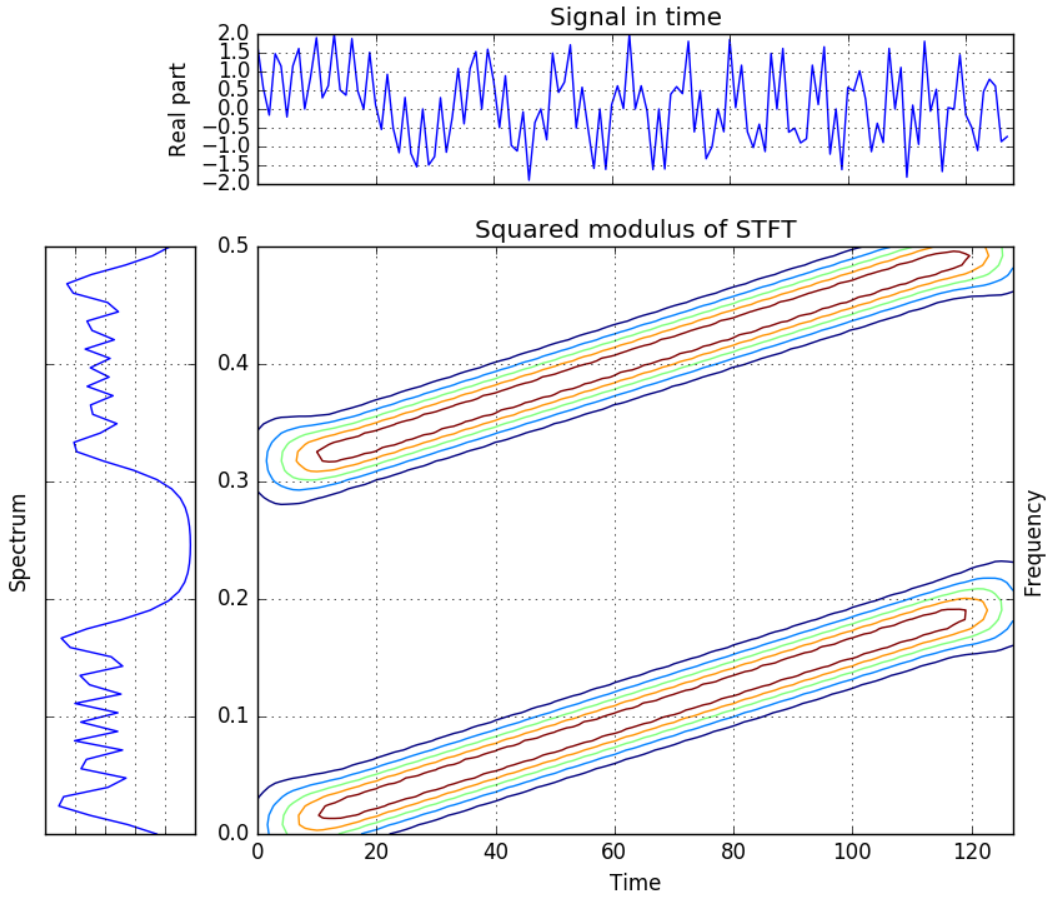
```
axTime = divider.append_axes("top", 1.2, pad=0.5)
axFreq = divider.append_axes("left", 1.2, pad=0.5)
axTime.plot(np.real(sig2))
axTime.set_xticklabels([])
axTime.set_xlim(0, 128)
axTime.set_ylabel('Real part')
axTime.set_title('Signal in time')
axTime.grid(True)
freq_y = np.linspace(0, 0.5, sig2.shape[0] / 2)
freq_x = (abs(np.fft.fftshift(np.fft.fft(sig2))) ** 2)[::-1][:64]
axFreq.plot(freq_x, freq_y)
axFreq.set_ylim(0.05, 0.45)
axFreq.set_yticklabels([])
axFreq.set_xticklabels([])
axFreq.grid(True)
axFreq.set_ylabel('Spectrum')
axFreq.invert_xaxis()
axFreq.grid(True)
plt.show()
```

Total running time of the script: (0 minutes 0.254 seconds)

3.1.57 Short time Fourier transform of a multi-component nonstationary signal

Compute and visualize the STFT of a multi component nonstationary signal.

Figure 2.11 from the tutorial.




```
from tftb.generators import fmlin
from tftb.processing.linear import ShortTimeFourierTransform
import matplotlib.pyplot as plt
from scipy.signal import hamming
import numpy as np
from mpl_toolkits.axes_grid1 import make_axes_locatable

N = 128
x1, _ = fmlin(N, 0, 0.2)
x2, _ = fmlin(N, 0.3, 0.5)
x = x1 + x2

n_fbins = 128
window = hamming(33)
tfr, _, _ = ShortTimeFourierTransform(x, timestamps=None, n_fbins=n_fbins,
                                      fwindow=window).run()
tfr = tfr[:64, :]
threshold = np.amax(np.abs(tfr)) * 0.05
tfr[np.abs(tfr) <= threshold] = 0.0 + 1j * 0.0
tfr = np.abs(tfr) ** 2
t = np.arange(tfr.shape[1])
f = np.linspace(0, 0.5, tfr.shape[0])
```

```
T, F = np.meshgrid(t, f)

fig, axScatter = plt.subplots(figsize=(10, 8))
axScatter.contour(T, F, tfr, 5)
axScatter.grid(True)
axScatter.set_title('Squared modulus of STFT')
axScatter.set_ylabel('Frequency')
axScatter.yaxis.set_label_position("right")
axScatter.set_xlabel('Time')
divider = make_axes_locatable(axScatter)
axTime = divider.append_axes("top", 1.2, pad=0.5)
axFreq = divider.append_axes("left", 1.2, pad=0.5)
axTime.plot(np.real(x))
axTime.set_xticklabels([])
axTime.set_xlim(0, N)
axTime.set_ylabel('Real part')
axTime.set_title('Signal in time')
axTime.grid(True)
axFreq.plot((abs(np.fft.fftshift(np.fft.fft(x))) ** 2)[::-1][:64], f[:64])
axFreq.set_yticklabels([])
axFreq.set_xticklabels([])
axFreq.invert_xaxis()
axFreq.set_ylabel('Spectrum')
axFreq.grid(True)
plt.show()
```

Total running time of the script: (0 minutes 0.234 seconds)

CHAPTER 4

API Reference

4.1 tftb package

4.1.1 Subpackages

tftb.generators package

Subpackages

tftb.generators.tests package

Submodules

tftb.generators.tests.test_amplitude_modulations module

tftb.generators.tests.test_analytic_signals module

tftb.generators.tests.test_frequency_modulations module

tftb.generators.tests.test_misc module

tftb.generators.tests.test_noise module

tftb.generators.tests.test_utils module

Module contents

Submodules

tftb.generators.amplitude_modulated module

tftb.generators.amplitude_modulated.**amexpos**(*n_points*, *t0=None*, *spread=None*, *kind='bilateral'*)

Exponential amplitude modulation.

amexpos generates an exponential amplitude modulation starting at time *t0* and spread proportioanl to *spread*.

Parameters

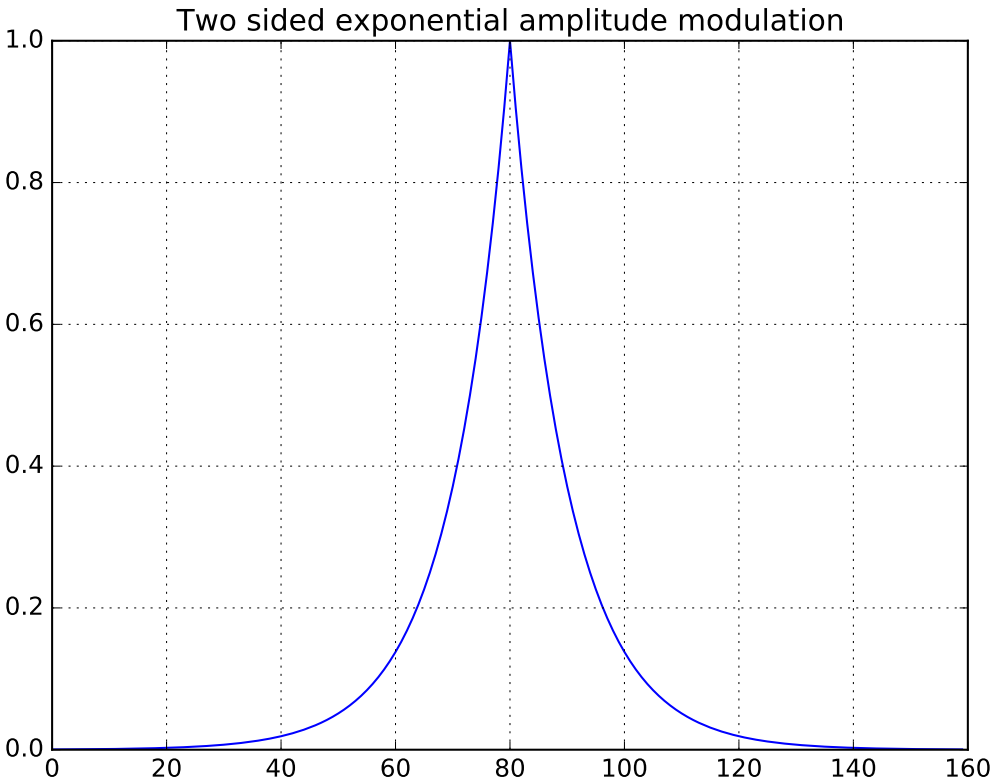
- **n_points** (*int*) – Number of points.
- **kind** (*str*) – “bilateral” (default) or “unilateral”
- **t0** (*float*) – Time center.
- **spread** (*float*) – Standard deviation.

Returns exponential function

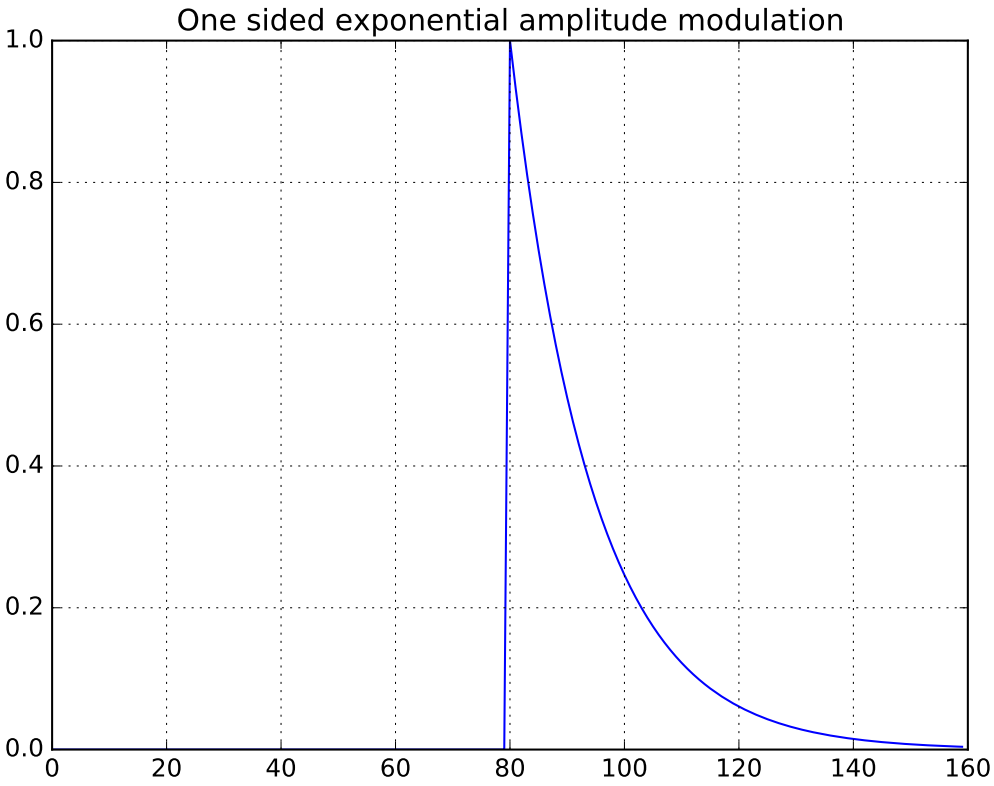
Return type numpy.ndarray

Examples

```
>>> x = amexpos(160)
>>> plot(x)
```




```
>>> x = amexpos(160, kind='unilateral')
>>> plot(x)
```



tftb.generators.amplitude_modulated.**amgauss**(*n_points*, *t0=None*, *spread=None*)

Generate a Gaussian amplitude modulated signal.

Parameters

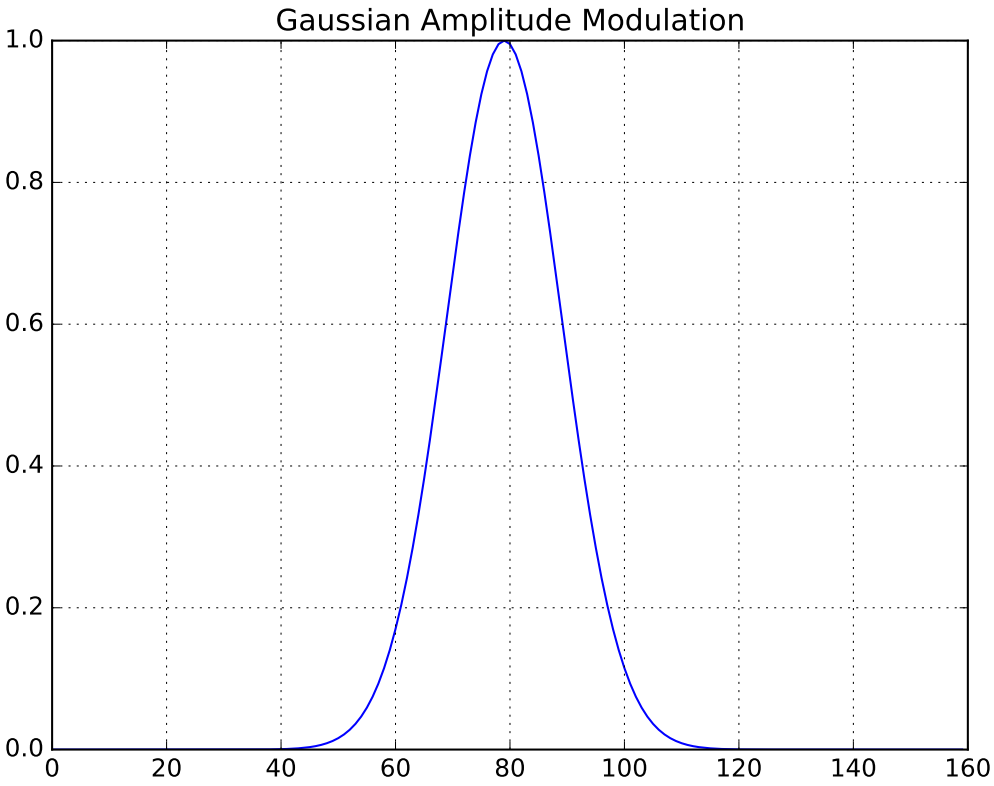
- **n_points** (*int*) – Number of points in the output.
- **t0** (*float*) – Center of the Gaussian function. (default: t0 / 2)
- **spread** (*float*) – Standard deviation of the Gaussian. (default 2 * sqrt(n_points)

Returns Gaussian function centered at time `t0`.

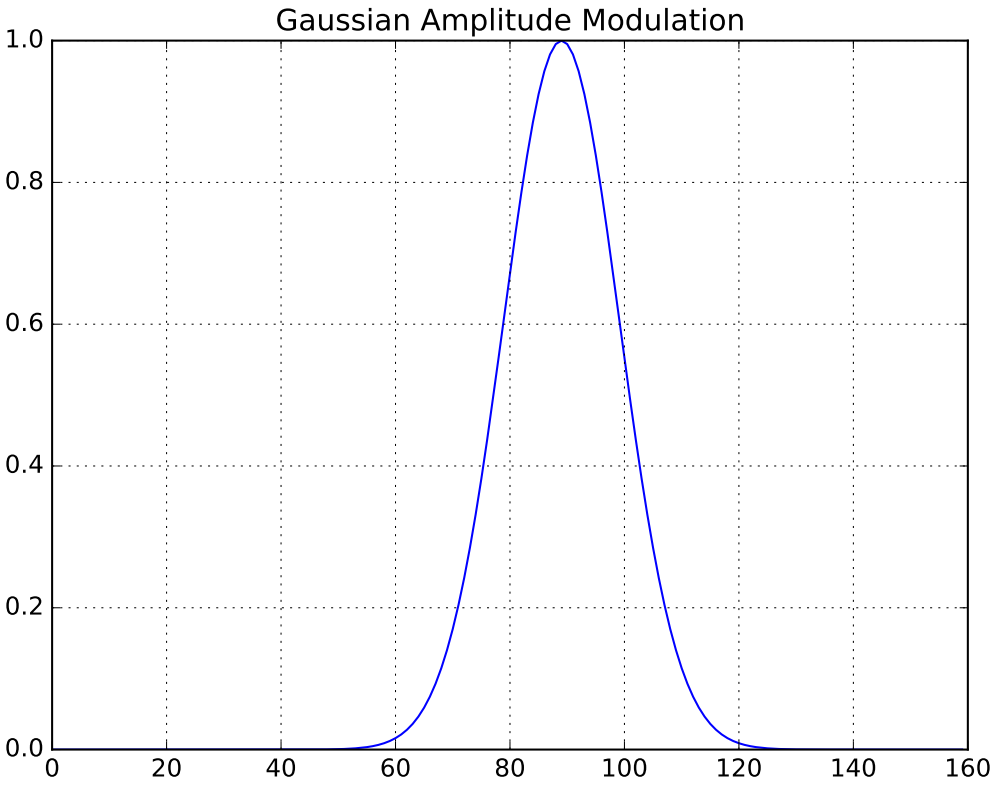
Return type numpy.ndarray

Example

```
>>> x = amgauss(160)
>>> plot(x)
```




```
>>> x = amgauss(160, 90)
>>> plot(x)
```



tftb.generators.amplitude_modulated.**amrect**(*n_points*, *t0=None*, *spread=None*)

Generate a rectangular amplitude modulation.

Parameters

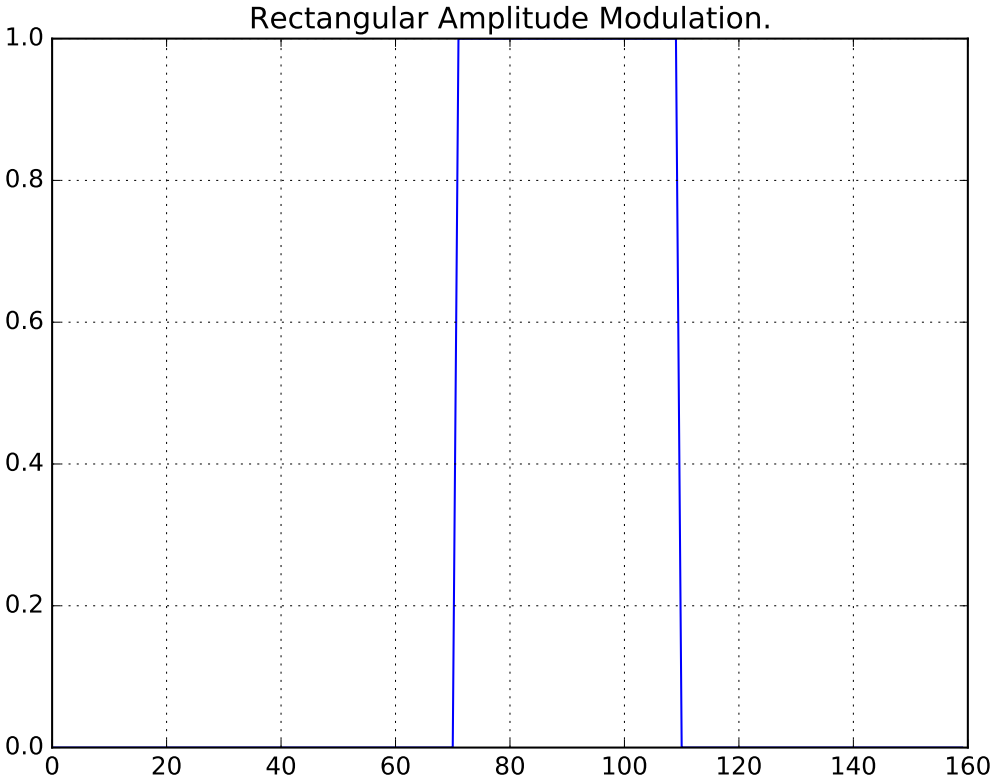
- **n_points** (*int*) – Number of points in the function.
- **t0** (*float*) – Time center
- **spread** (*float*) – standard deviation of the function.

Returns A rectangular amplitude modulator.

Return type numpy.ndarray.

Examples

```
>>> x = amrect(160, 90, 40.0)
>>> plot(x)
```



tftb.generators.amplitude_modulated.**amtriang**(*n_points*, *t0=None*, *spread=None*)

Generate a triangular amplitude modulation.

Parameters

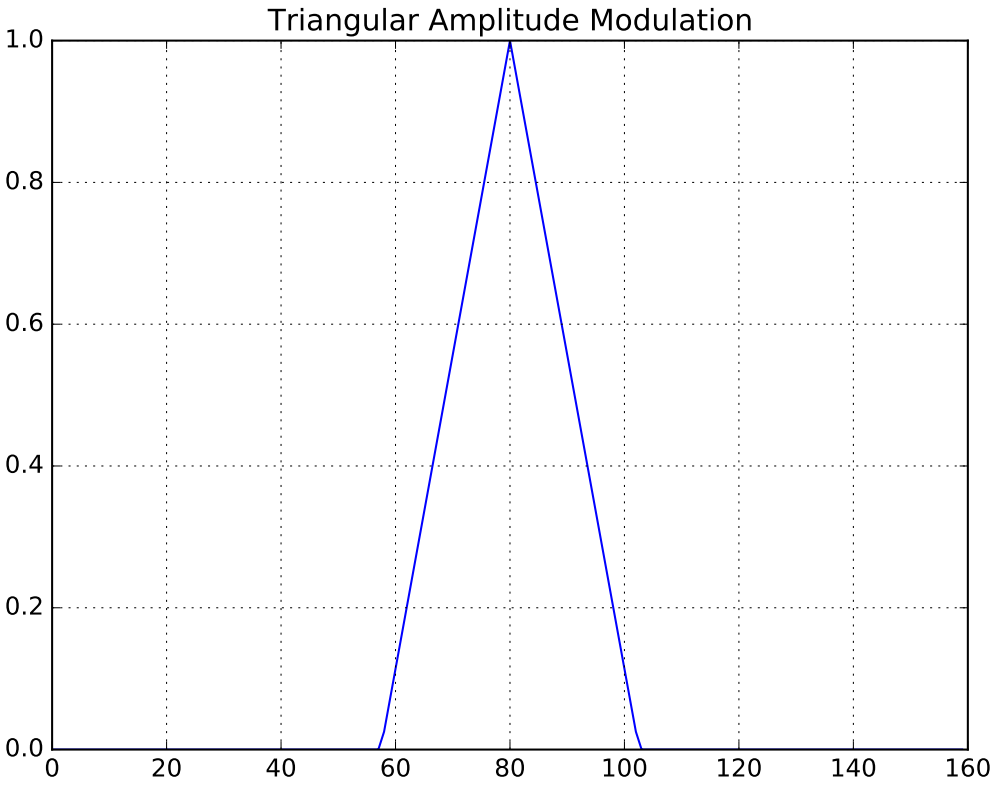
- **n_points** (*int*) – Number of points in the function.
- **t0** (*float*) – Time center
- **spread** (*float*) – standard deviation of the function.

Returns A triangular amplitude modulator.

Return type numpy.ndarray.

Examples

```
>>> x = amtriang(160)
>>> plot(x)
```



tftb.generators.analytic_signals module

tftb.generators.analytic_signals.**anaask**(*n_points*, *n_comp=None*, *f0=0.25*)

Generate an amplitude shift (ASK) keying signal.

Parameters

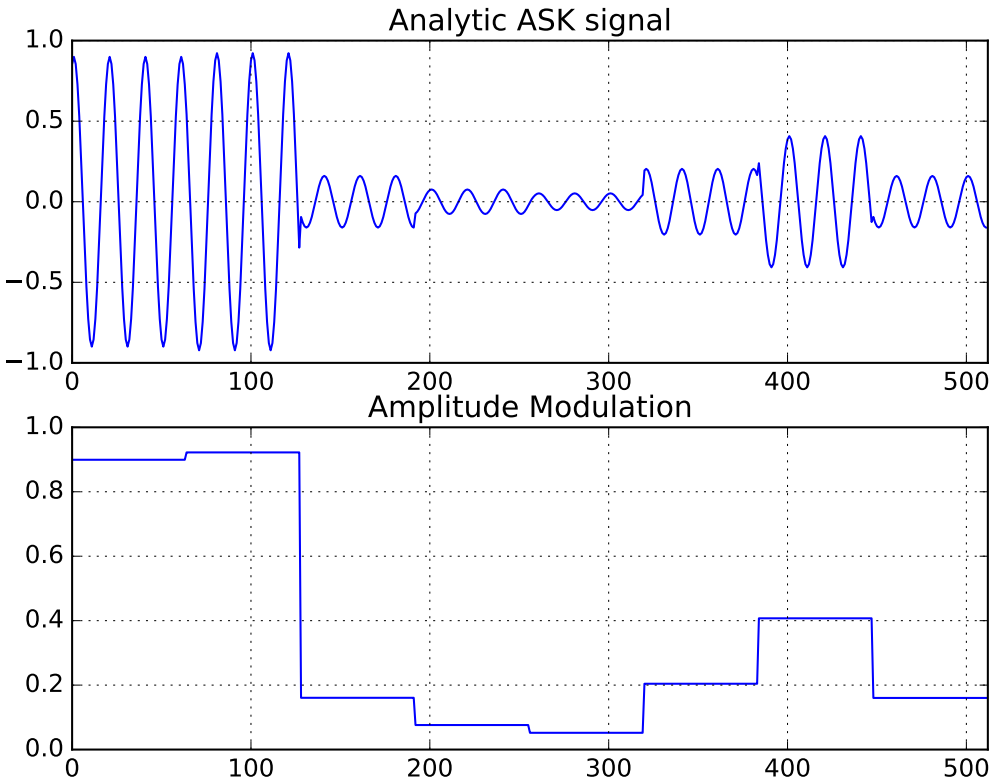
- **n_points** (*int*) – number of points.
- **n_comp** (*int*) – number of points of each component.
- **f0** (*float*) – normalized frequency.

Returns Tuple containing the modulated signal and the amplitude modulation.

Return type tuple(numpy.ndarray)

Examples

```
>>> x, am = anaask(512, 64, 0.05)
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(am)
```



tftb.generators.analytic_signals.**anabpsk**(*n_points*, *n_comp=None*, *f0=0.25*)

Binary phase shift keying (BPSK) signal.

Parameters

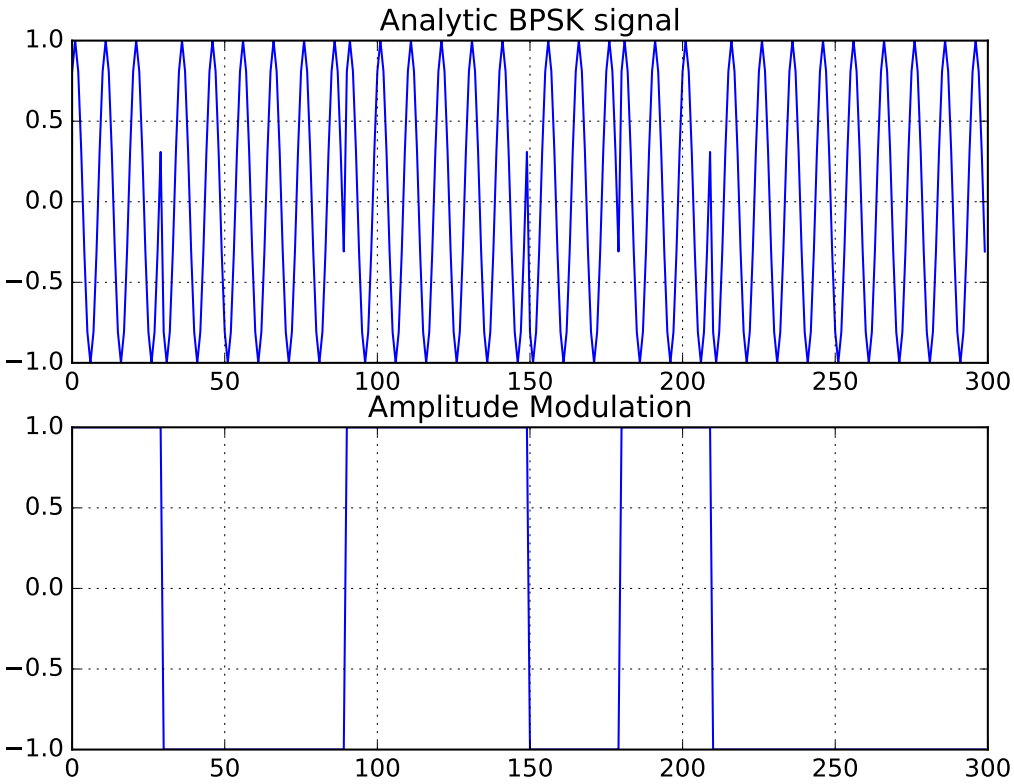
- **n_points** (*int*) – number of points.
- **n_comp** (*int*) – number of points in each component.
- **f0** (*float*) – normalized frequency.

Returns BPSK signal

Return type numpy.ndarray

Examples

```
>>> x, am = anabpsk(300, 30, 0.1)
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(am)
```



tftb.generators.analytic_signals.**anafsk**(*n_points*, *n_comp=None*, *Nbf=4*)

Frequency shift keying (FSK) signal.

Parameters

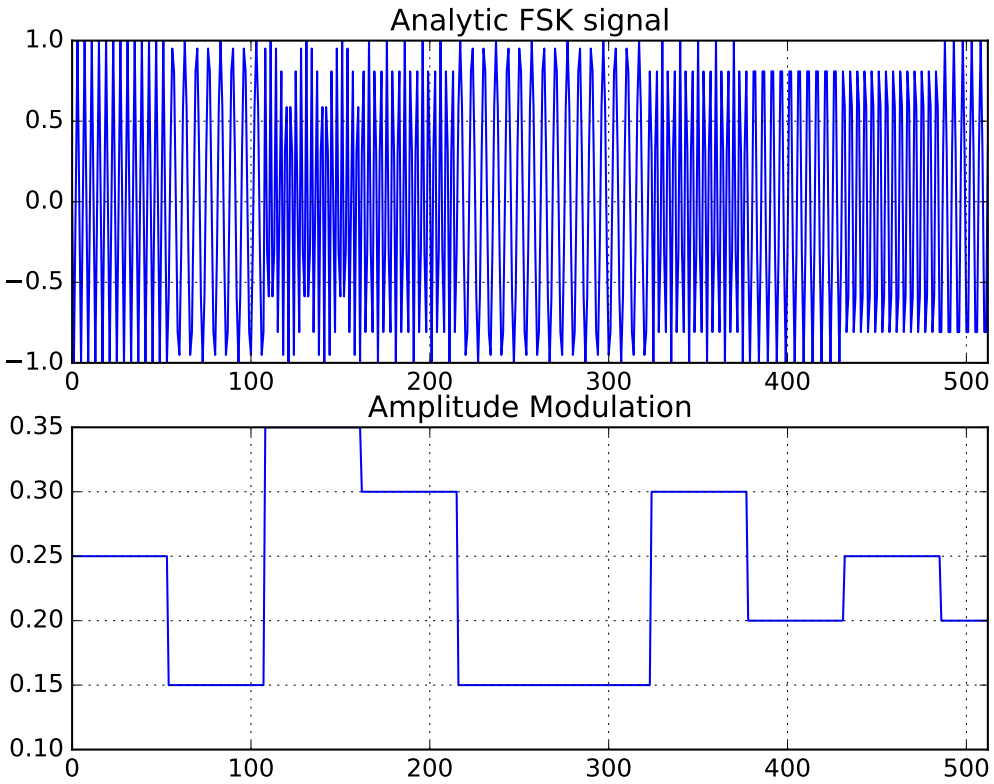
- **n_points** (*int*) – number of points.
- **n_comp** (*int*) – number of points in each components.
- **Nbf** (*int*) – number of distinct frequencies.

Returns FSK signal.

Return type numpy.ndarray

Examples

```
>>> x, am = anafsk(512, 54.0, 5.0)
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(am)
```



tftb.generators.analytic_signals.**anapulse**(*n_points*, *ti=None*)

Analytic projection of unit amplitude impulse signal.

Parameters

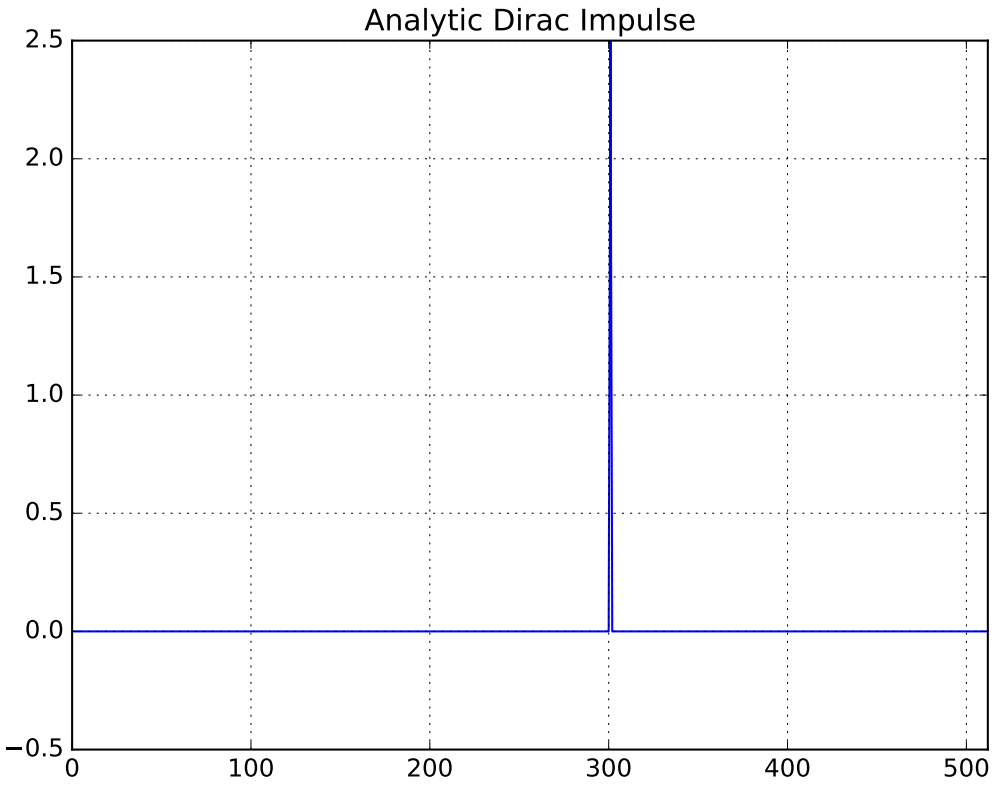
- **n_points** (*int*) – Number of points.
- **ti** (*float*) – time position of the impulse.

Returns analytic impulse signal.

Return type numpy.ndarray

Examples

```
>>> x = 2.5 * anapulse(512, 301)
>>> plot(real(x))
```



tftb.generators.analytic_signals.**anaqpsk**(*n_points*, *n_comp=None*, *f0=0.25*)

Quaternary Phase Shift Keying (QPSK) signal.

Parameters

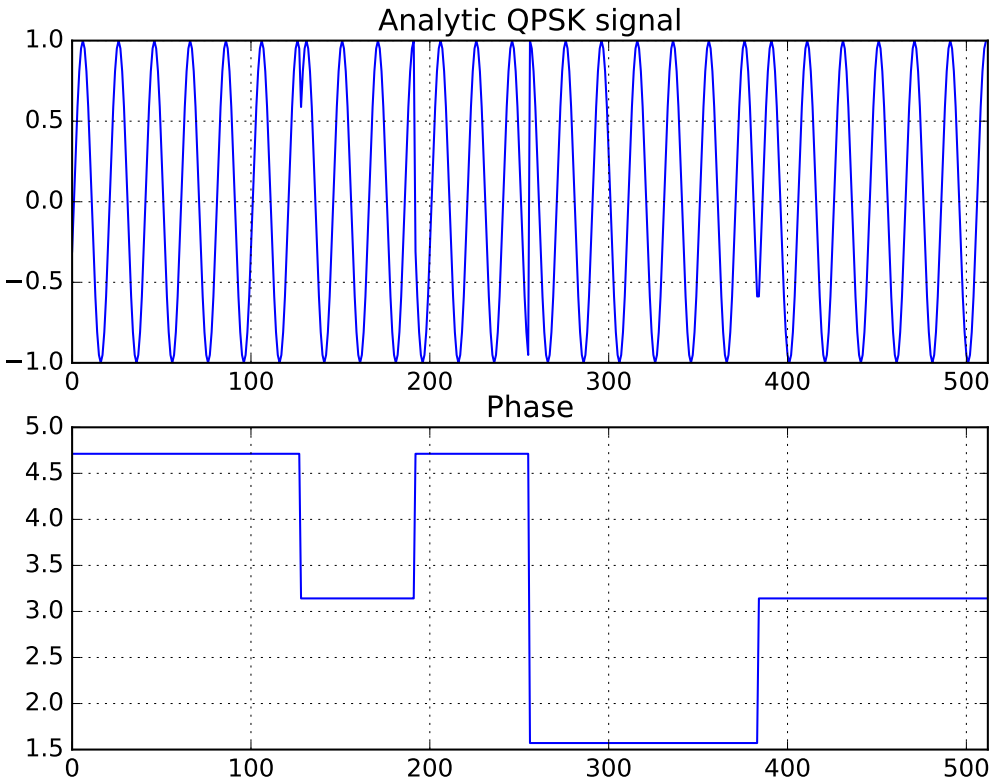
- **n_points** (*int*) – number of points.
- **n_comp** (*int*) – number of points in each component.
- **f0** (*float*) – normalized frequency

Returns complex phase modulated signal of normalized frequency f0 and initial phase.

Return type tuple

Examples

```
>>> x, phase = anaqpsk(512, 64.0, 0.05)
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(phase)
```



tftb.generators.analytic_signals.**anasing**(*n_points*, *t0=None*, *h=0.0*)

Lipschitz singularity. Refer to the wiki page on *Lipschitz condition*, good test case.

Parameters

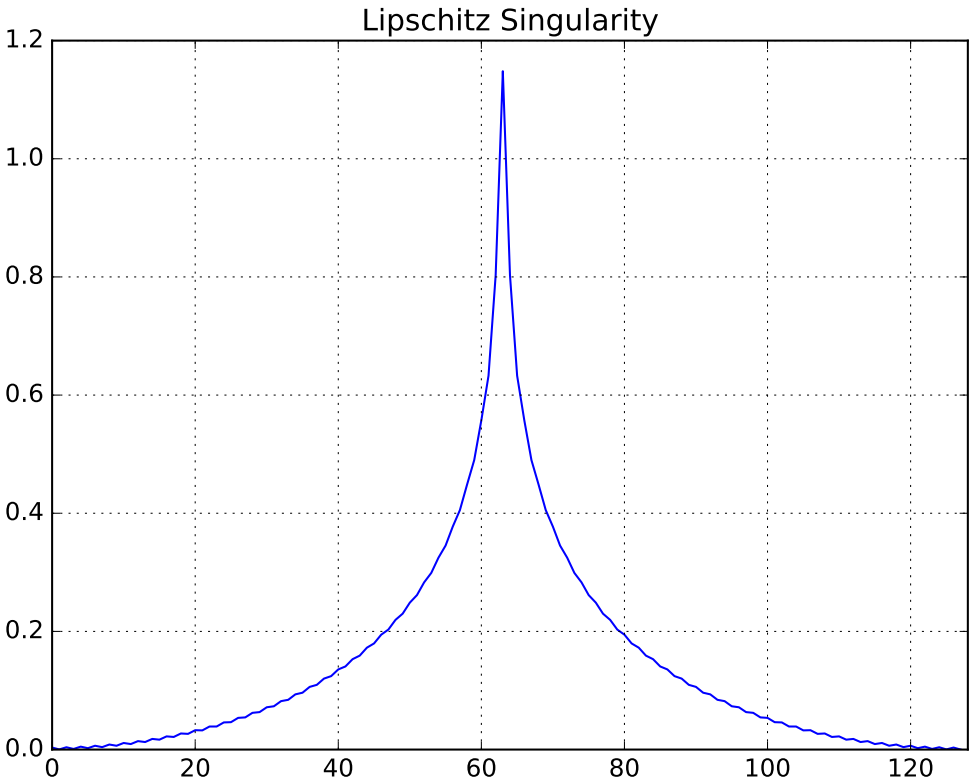
- **n_points** (*int*) – number of points in time.
- **t0** (*float*) – time localization of singularity
- **h** (*float*) – strength of the singularity

Returns N-point Lipschitz singularity centered around t0

Return type numpy.ndarray

Examples

```
>>> x = anasing(128)
>>> plot(real(x))
```



tftb.generators.analytic_signals.**anastep**(*n_points*, *ti=None*)

Analytic projection of unit step signal.

Parameters

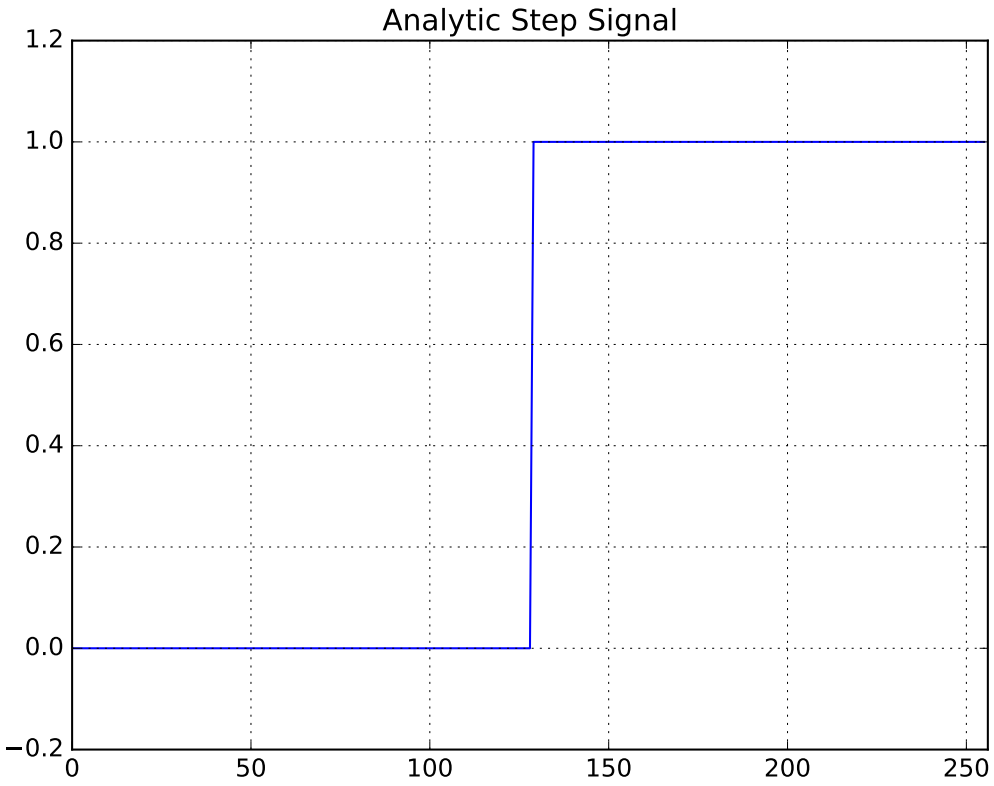
- **n_points** (*int*) – Number of points.
- **ti** (*float*) – starting position of unit step.

Returns output signal

Return type numpy.ndarray

Examples

```
>>> x = anastep(256, 128)
>>> plot(real(x))
```



tftb.generators.frequency_modulated module

tftb.generators.frequency_modulated.**fmconst**(*n_points*, *fnorm=0.25*, *t0=None*)

Generate a signal with constant frequency modulation.

Parameters

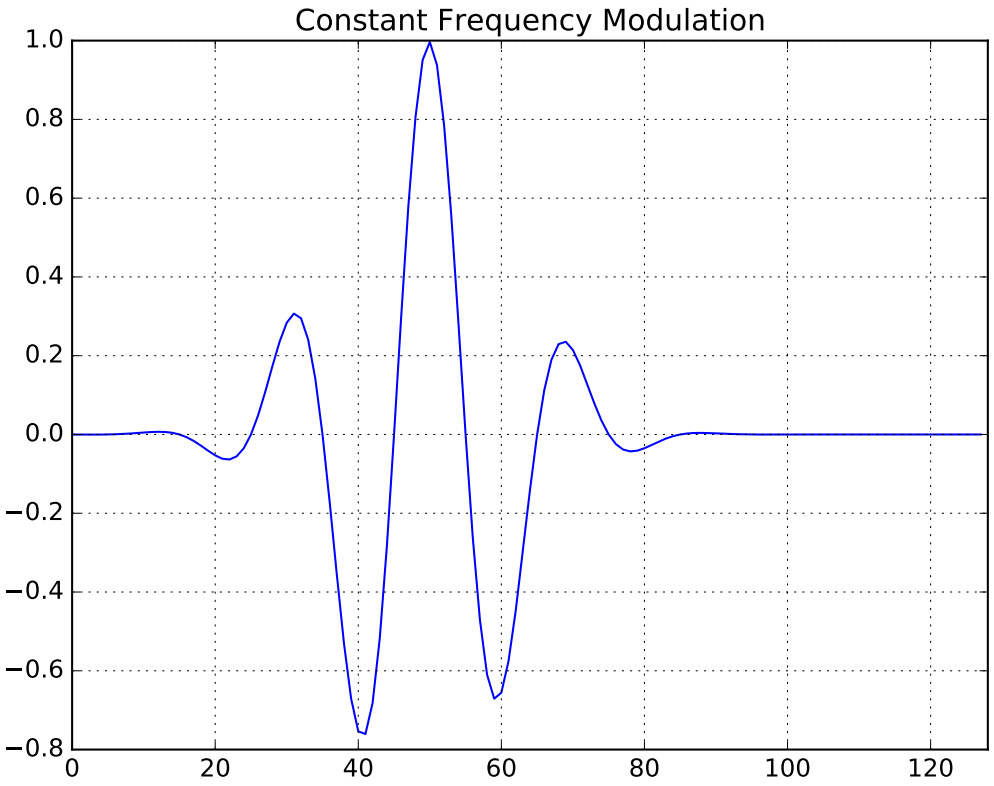
- **n_points** (*int*) – number of points
- **fnorm** (*float*) – normalized frequency
- **t0** (*float*) – time center

Returns frequency modulation signal with frequency fnorm

Return type numpy.ndarray

Examples

```
>>> from tftb.generators import amgauss
>>> z = amgauss(128, 50, 30) * fmconst(128, 0.05, 50)[0]
>>> plot(real(z))
```



tftb.generators.frequency_modulated.**fmhyp**(*n_points*, *p1*, *p2*)

Signal with hyperbolic frequency modulation.

Parameters

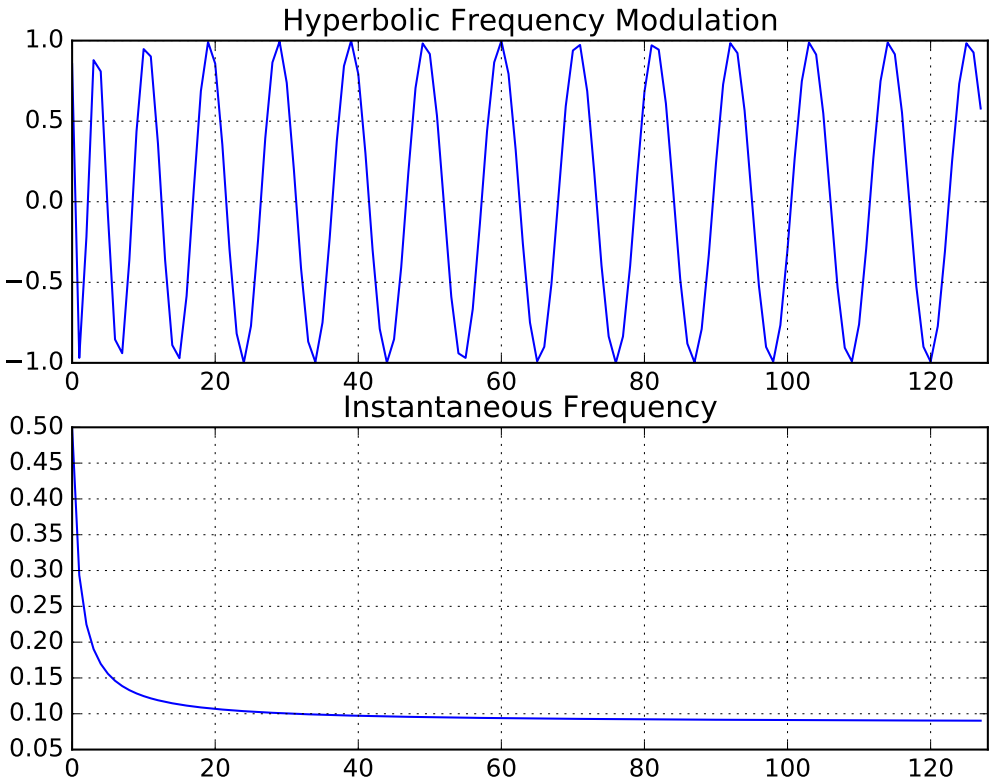
- **n_points** (*int*) – number of points.
- **p2** (*float*) – coefficients of the hyperbolic function.

Returns vector containing the modulated signal samples.

Return type numpy.ndarray

Examples

```
>>> signal, iflaw = fmhyp(128, (1, 0.5), (32, 0.1))
>>> subplot(211), plot(real(signal))
>>> subplot(212), plot(iflaw)
```



tftb.generators.frequency_modulated.**fmlin**(*n_points*, *fnormi=0.0*, *fnormf=0.5*, *t0=None*)

Generate a signal with linear frequency modulation.

Parameters

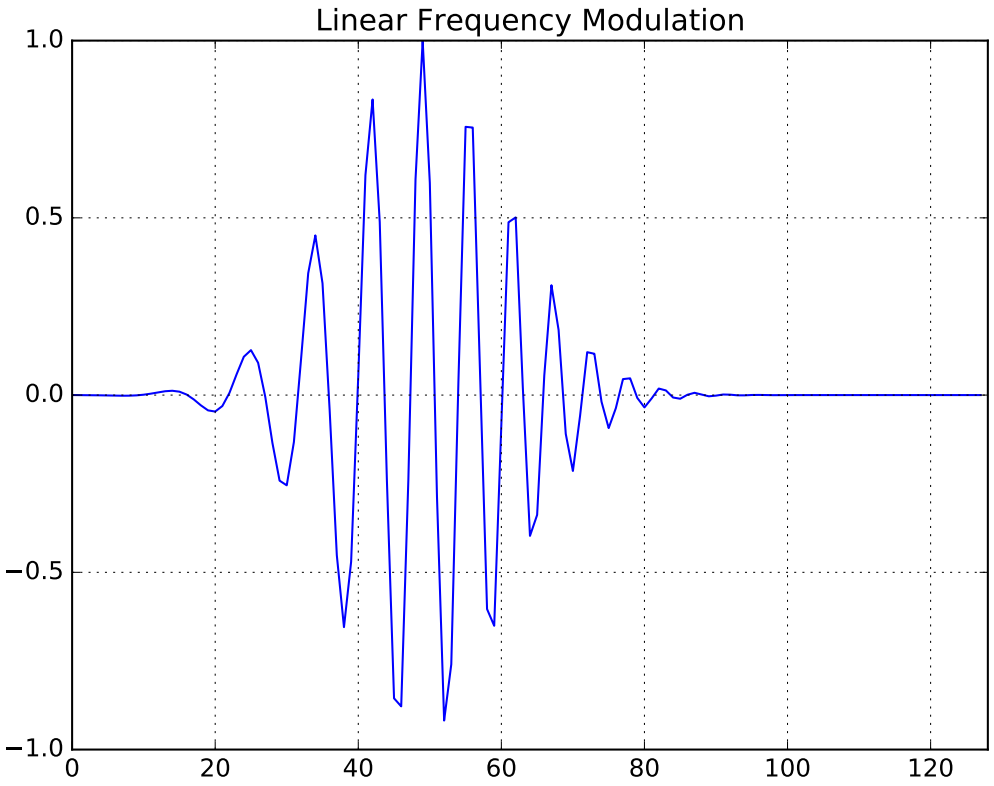
- **n_points** (*int*) – number of points
- **fnormi** (*float*) – initial normalized frequency
- **fnormf** (*float*) – final normalized frequency
- **t0** (*float*) – time center

Returns The modulated signal, and the instantaneous amplitude law.

Return type tuple(array-like)

Examples

```
>>> from tftb.generators import amgauss
>>> z = amgauss(128, 50, 40) * fmlin(128, 0.05, 0.3, 50)[0]
>>> plot(real(z))
```



tftb.generators.frequency_modulated.**fmodany**(*iflaw*, *t0=1*)

Arbitrary frequency modulation.

Parameters

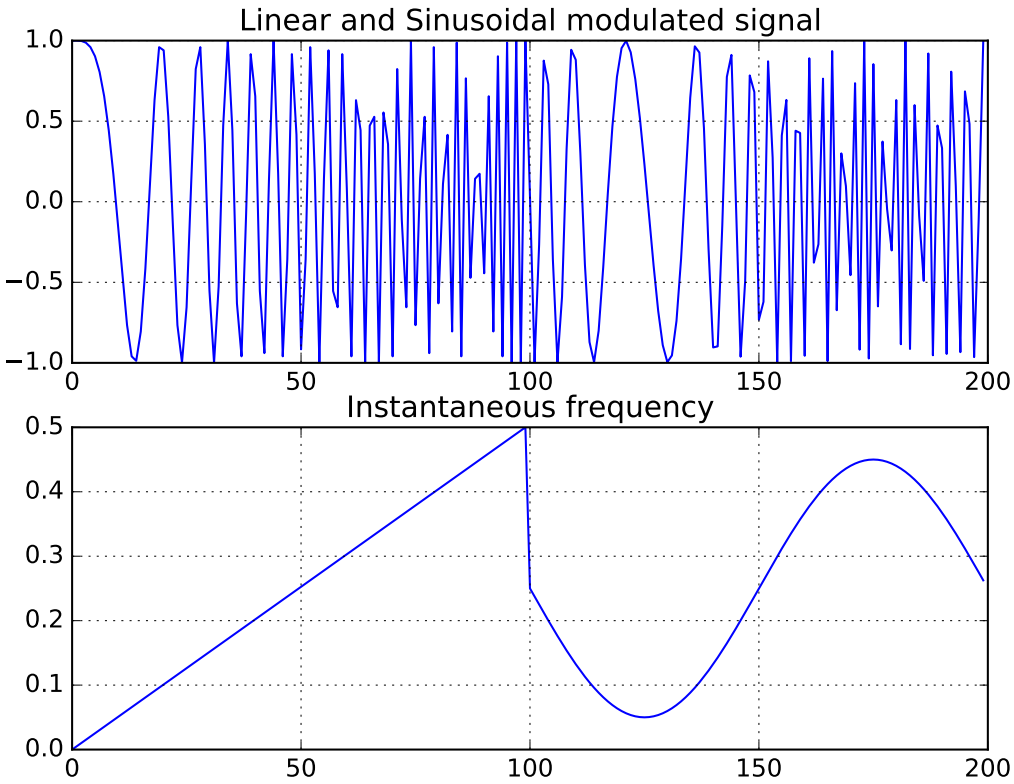
- **iflaw** (*numpy.ndarray*) – Vector of instantaneous frequency law samples.
- **t0** (*float*) – time center

Returns output signal

Return type

Examples

```
>>> from tftb.generators import fmlin
>>> import numpy as np
>>> y1, ifl1 = fmlin(100)  # A linear instantaneous frequency law.
>>> y2, ifl2 = fmsin(100)  # A sinusoidal instantaneous frequency law.
>>> iflaw = np.append(ifl1, ifl2)  # combination of the two
>>> sig = fmodany(iflaw)
>>> subplot(211), plot(real(sig))
>>> subplot(212), plot(iflaw)
```



tftb.generators.frequency_modulated.**fmpar**(*n_points*, *coefficients*)

Parabolic frequency modulated signal.

Parameters

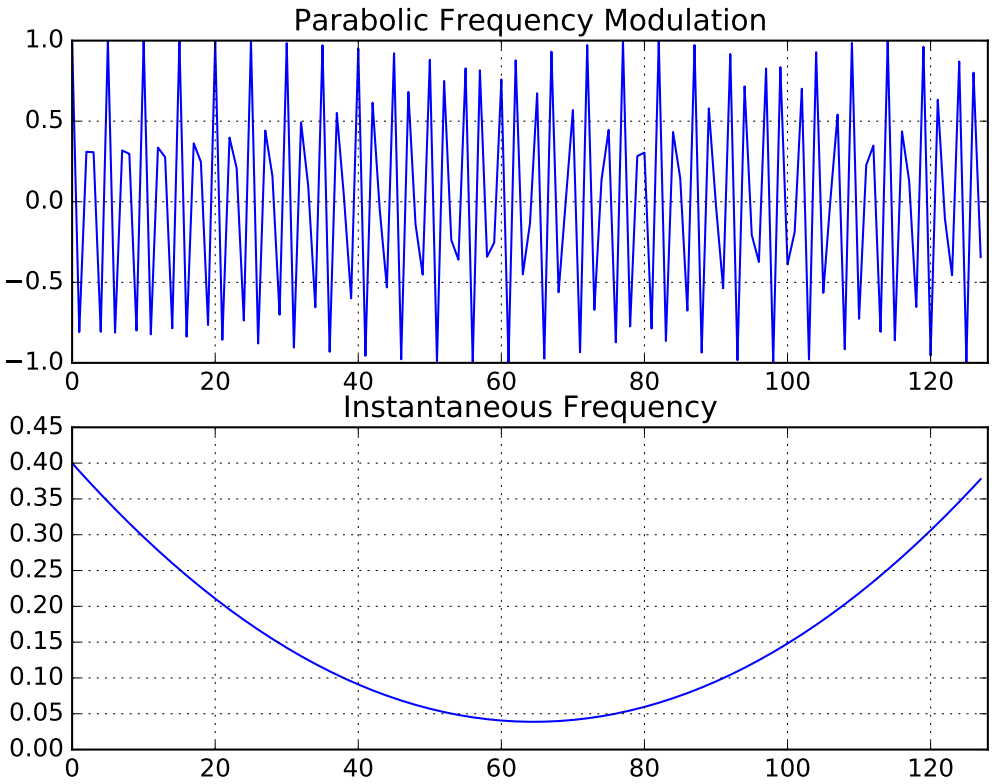
- **n_points** (*int*) – number of points
- **coefficients** (*tuple*) – coefficients of the parabolic function.

Returns Signal with parabolic frequency modulation law.

Return type tuple

Examples

```
>>> x, iflaw = fmpar(128, (0.4, -0.0112, 8.6806e-05))
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(iflaw)
```



tftb.generators.frequency_modulated.**fmpower**(*n_points*, *k*, *coefficients*)

Generate signal with power law frequency modulation.

Parameters

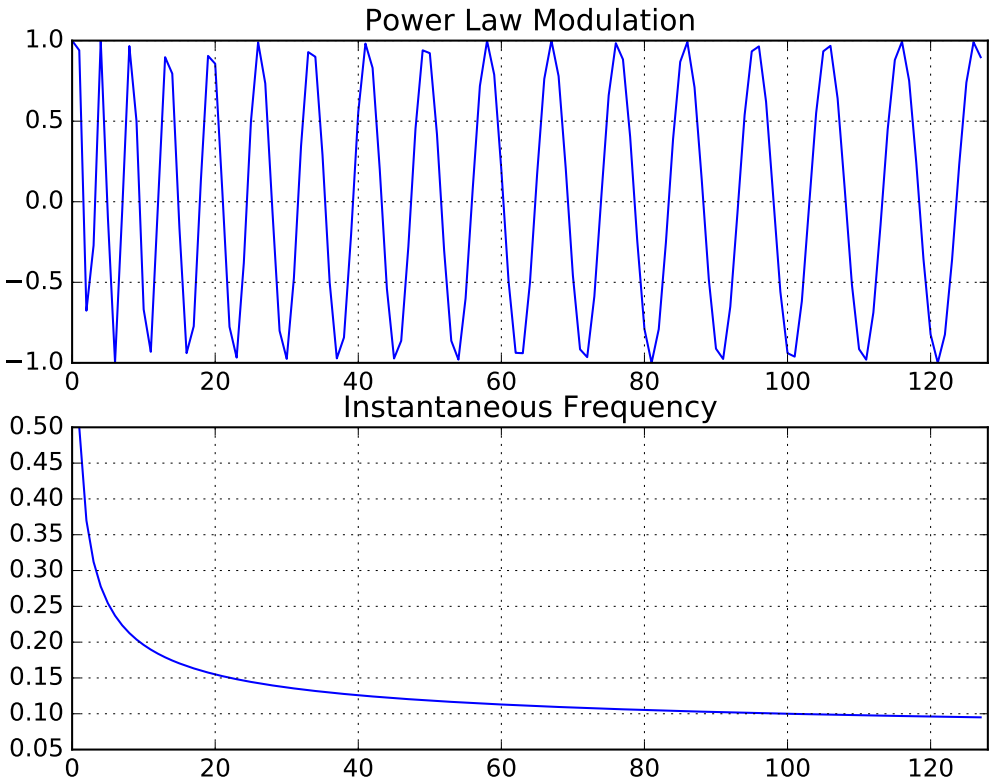
- **n_points** (*int*) – number of points.
- **k** (*int*) – degree of power law.
- **p2** (*float*) – coefficients of the power law.

Returns vector of modulated signal samples.

Return type numpy.ndarray

Examples

```
>>> x, iflaw = fmpower(128, 0.5, (1, 0.5, 100, 0.1))
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(iflaw)
```



tftb.generators.frequency_modulated.**fmsin**(*n_points*, *fnormin=0.05*, *fnormax=0.45*, *period=None*, *t0=None*, *fnorm0=None*, *pm1=1*)

Sinusodial frequency modulation.

Parameters

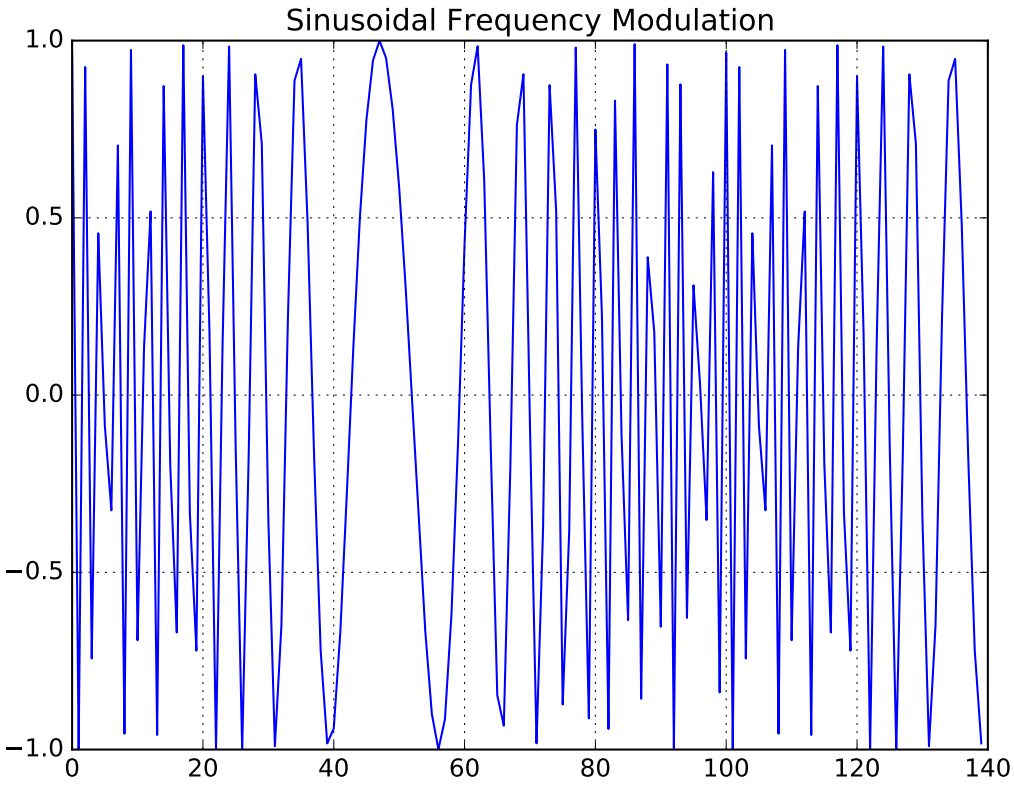
- **n_points** (*int*) – number of points
- **fnormin** (*float*) – smallest normalized frequency
- **fnormax** (*float*) – highest normalized frequency
- **period** (*int*) – period of sinusoidal fm
- **t0** (*float*) – time reference
- **fnorm0** (*float*) – normalized frequency at time t0
- **pm1** (*int*) – frequency direction at t0

Returns output signal

Return type numpy.ndarray

Examples

```
>>> z = fmsin(140, period=100, t0=20, fnorm0=0.3, pm1=-1)
>>> plot(real(z))
```



tftb.generators.misc module

tftb.generators.misc.**altes**(*n_points*, *fmin=0.05*, *fmax=0.5*, *alpha=300*)

Generate the Altes signal in time domain.

Parameters

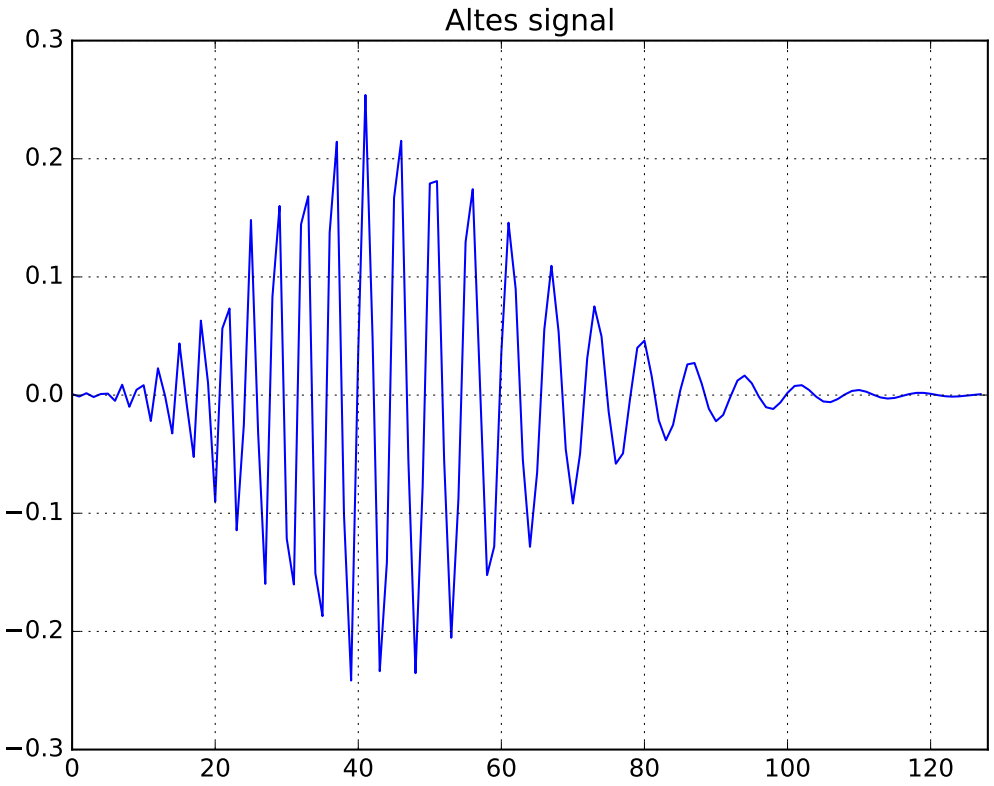
- **n_points** (*int*) – Number of points in time.
- **fmin** (*float*) – Lower frequency bound.
- **fmax** (*float*) – Higher frequency bound.
- **alpha** (*float*) – Attenuation factor of the envelope.

Returns Time vector containing the Altes signal samples.

Return type numpy.ndarray

Examples

```
>>> x = altes(128, 0.1, 0.45)
>>> plot(x)
```



tftb.generators.misc.**atoms**(*n_points*, *coordinates*)

Compute linear combination of elementary Gaussian atoms.

Parameters

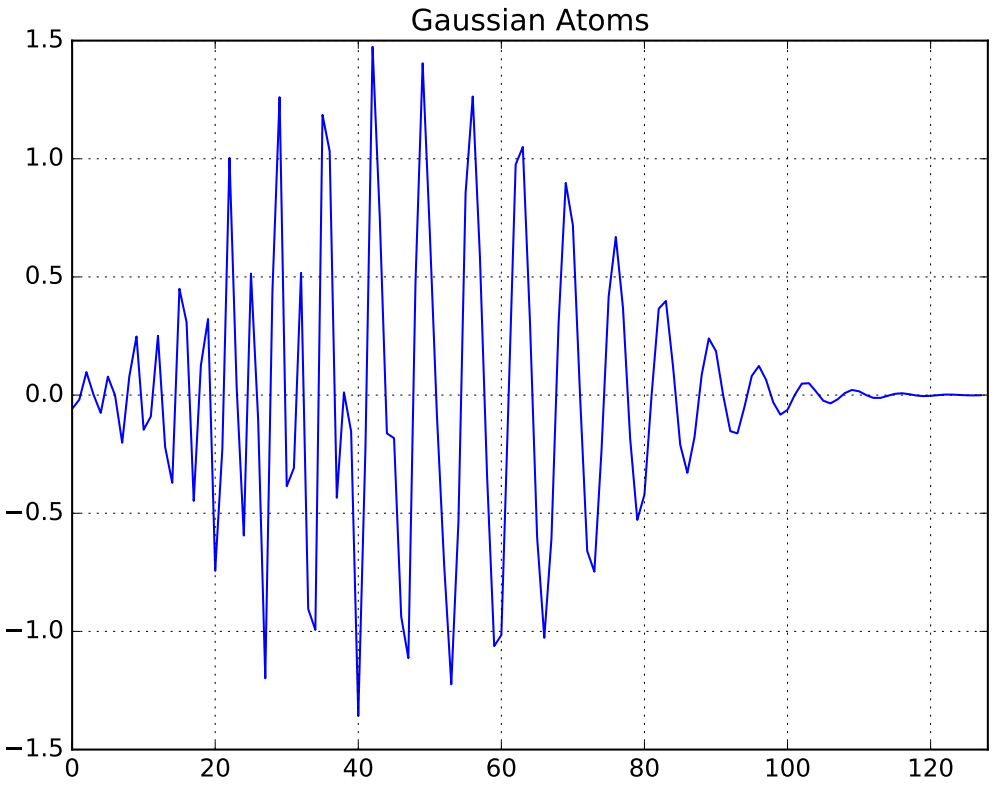
- **n_points** (*int*) – Number of points in a signal
- **coordinates** (*array-like*) – matrix of time-frequency centers

Returns signal

Return type array-like

Examples

```
>>> import numpy as np
>>> coordinates = np.array([[32.0, 0.3, 32.0, 1.0], [0.15, 0.15, 48.0, 1.22]])
>>> sig = atoms(128, coordinates)
>>> plot(real(sig))
```



tftb.generators.misc.**doppler**(*n_points*, *s_freq*, *f0*, *distance*, *v_target*, *t0=None*, *v_wave=340.0*)
Generate complex Doppler signal

Parameters

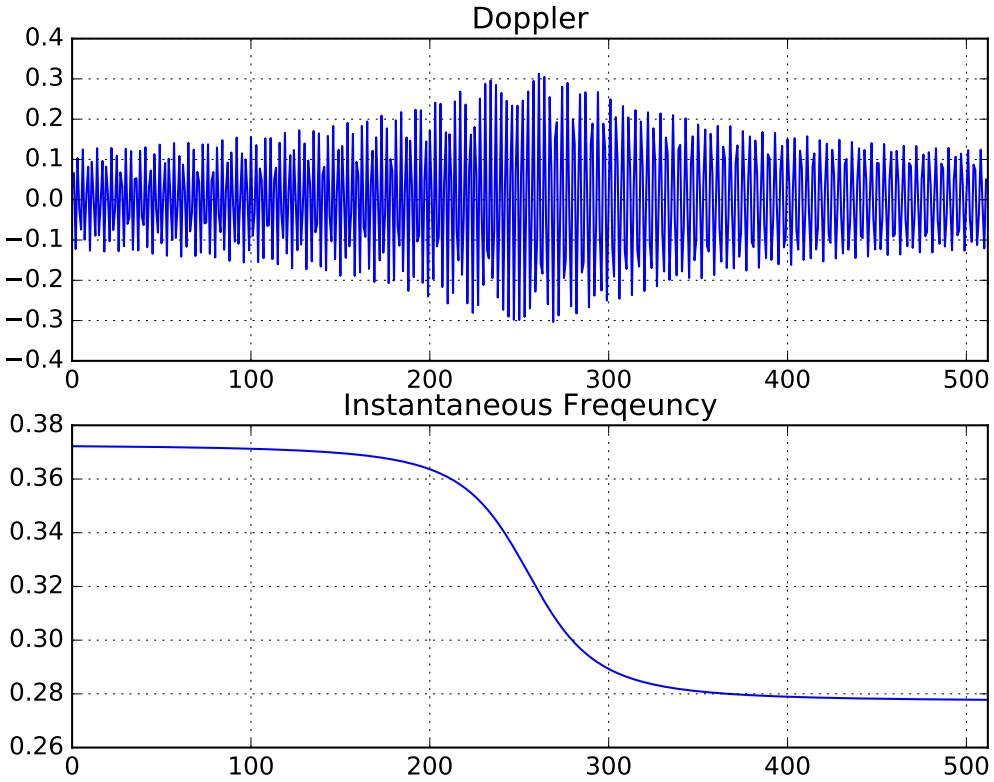
- **n_points** (*int*) – Number of points
- **s_freq** (*float*) – Sampling frequency
- **f0** (*float*) – Target frequency
- **distance** (*float*) – distance from line to observer
- **v_target** (*float*) – Target velocity
- **t0** (*float*) – Time center
- **v_wave** (*float*) – wave velocity.

Returns Tuple containing output frequency modulator, output amplitude modulator, output instantaneous frequency law.

Return type tuple

Example

```
>>> fm, am, iflaw = doppler(512, 200.0, 65.0, 10.0, 50.0)
>>> subplot(211), plot(real(am * fm))
>>> subplot(211), plot(iflaw)
```



tftb.generators.misc.**gdpower**(*n_points*, *degree=0.0*, *rate=1.0*)
Generate a signal with a power law group delay.

Parameters

- **n_points** (*int*) – Number of points in time.
- **degree** (*int # yoder: i'm pretty sure this is supposed to be a float. at least in test_misc.py, the test for this funct. passes degree=0.5*) – degree of the power law.
- **rate** (*float*) – rate-coefficient of the power law GD.

Returns Tuple of time row containing modulated samples, group delay, frequency bins.

Return type tuple

tftb.generators.misc.**klauder**(*n_points*, *attenuation=10.0*, *f0=0.2*)
Klauder wavelet in time domain.

Parameters

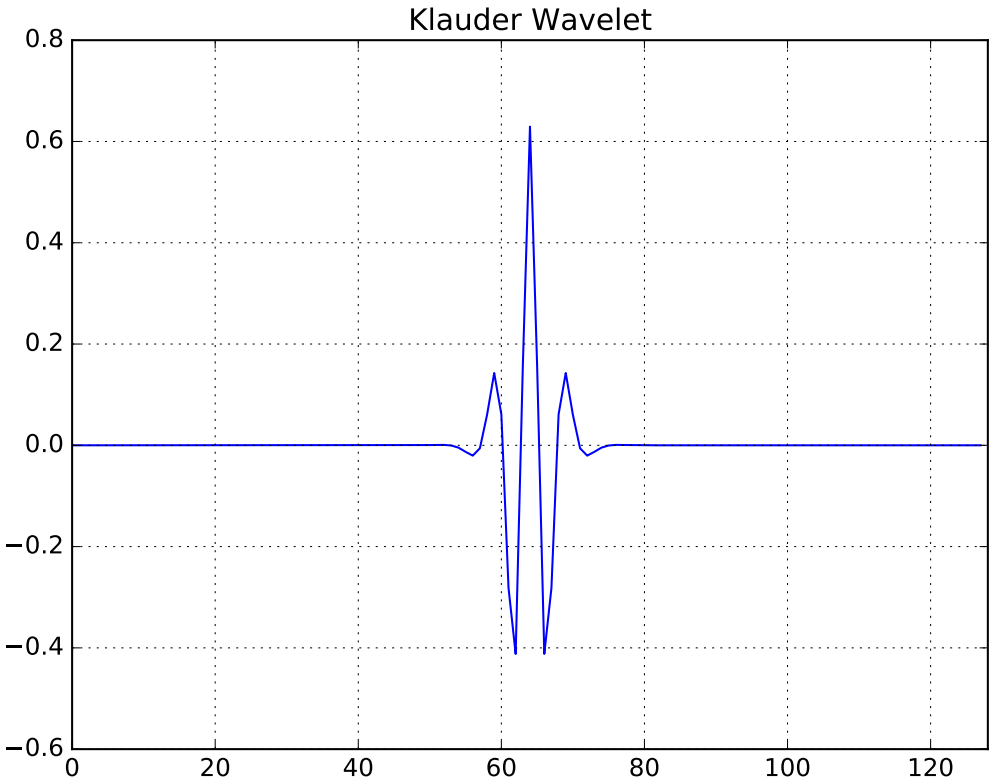
- **n_points** (*int*) – Number of points in time.
- **attenuation** (*float*) – attenuation factor of the envelope.
- **f0** (*float*) – central frequency of the wavelet.

Returns Time row vector containing klauder samples.

Return type numpy.ndarray

Example

```
>>> x = klauder(128)
>>> plot(x)
```



tftb.generators.misc.**mexhat**(*nu=0.05*)

Mexican hat wavelet in time domain.

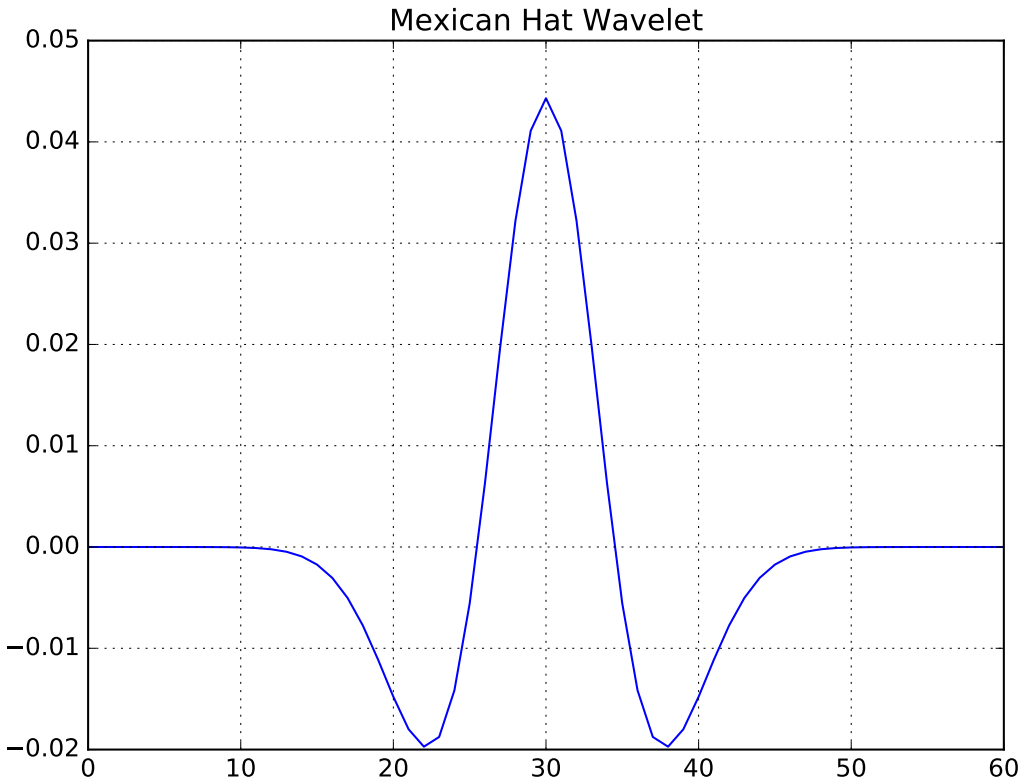
Parameters **nu** (*float*) – Central normalized frequency of the wavelet. Must be a real number between 0 and 0.5

Returns time vector containing mexhat samples.

Return type numpy.ndarray

Example

```
>>> plot(mexhat())
```



tftb.generators.noise module

tftb.generators.noise.**dopnoise**(*n_points*, *s_freq*, *f_target*, *distance*, *v_target*, *time_center=None*, *c=340*)

Generate complex noisy doppler signal, normalized to have unit energy.

Parameters

- **n_points** (*int*) – Number of points.
- **s_freq** (*float*) – Sampling frequency.
- **f_target** (*float*) – Frequency of target.
- **distance** (*float*) – Distnace from line to observer.
- **v_target** (*float*) – velocity of target relative to observer.
- **time_center** (*float*) – Time center. (Default n_points / 2)
- **c** (*float*) – Wave velocity (Default 340 m/s)

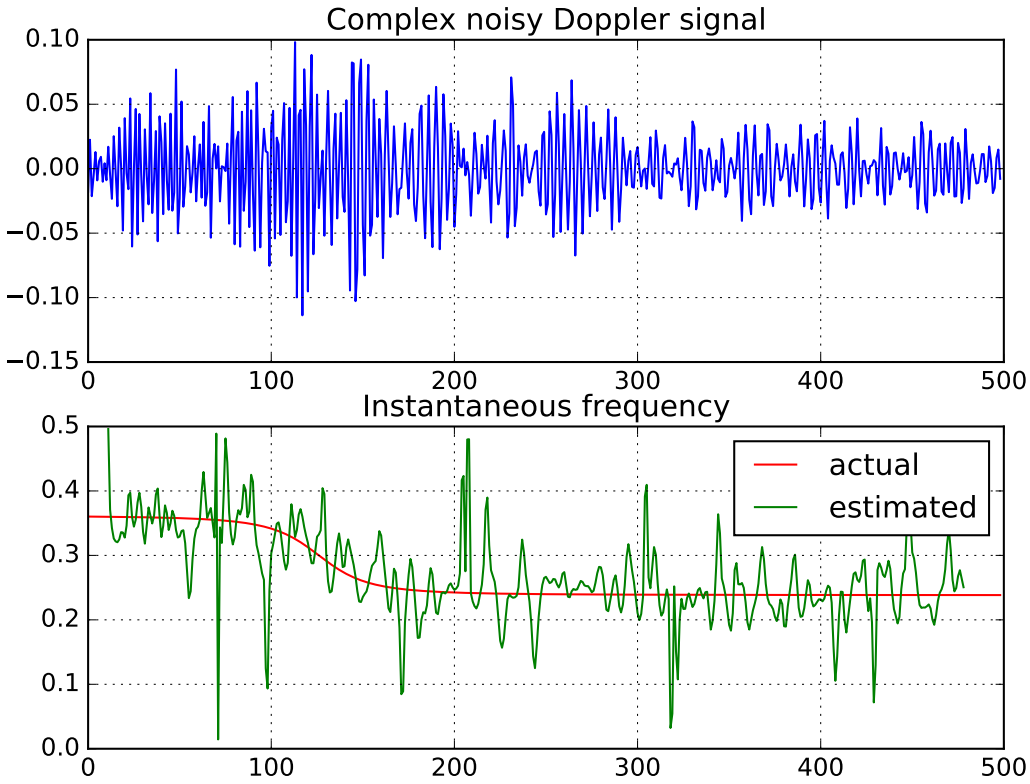
Returns tuple (output signal, instantaneous frequency law.)

Return type tuple(array-like)

Example

```
>>> import numpy as np
>>> from tftb.processing import inst_freq
```

```
>>> z, iflaw = dopnoise(500, 200.0, 60.0, 10.0, 70.0, 128.0)
>>> subplot(211), plot(real(z))
>>> ifl = inst_freq(z, np.arange(11, 479), 10)
>>> subplot(212), plot(iflaw, 'r', ifl, 'g')
```



tftb.generators.noise.**noisecg**(*n_points*, *a1=None*, *a2=None*)

Generate analytic complex gaussian noise with mean 0.0 and variance 1.0.

Parameters

- **n_points** (*int*) – Length of the desired output signal.
- **a1** (*float*) – Coefficients of the filter through which the noise is passed.
- **a2** (*float*) – Coefficients of the filter through which the noise is passed.

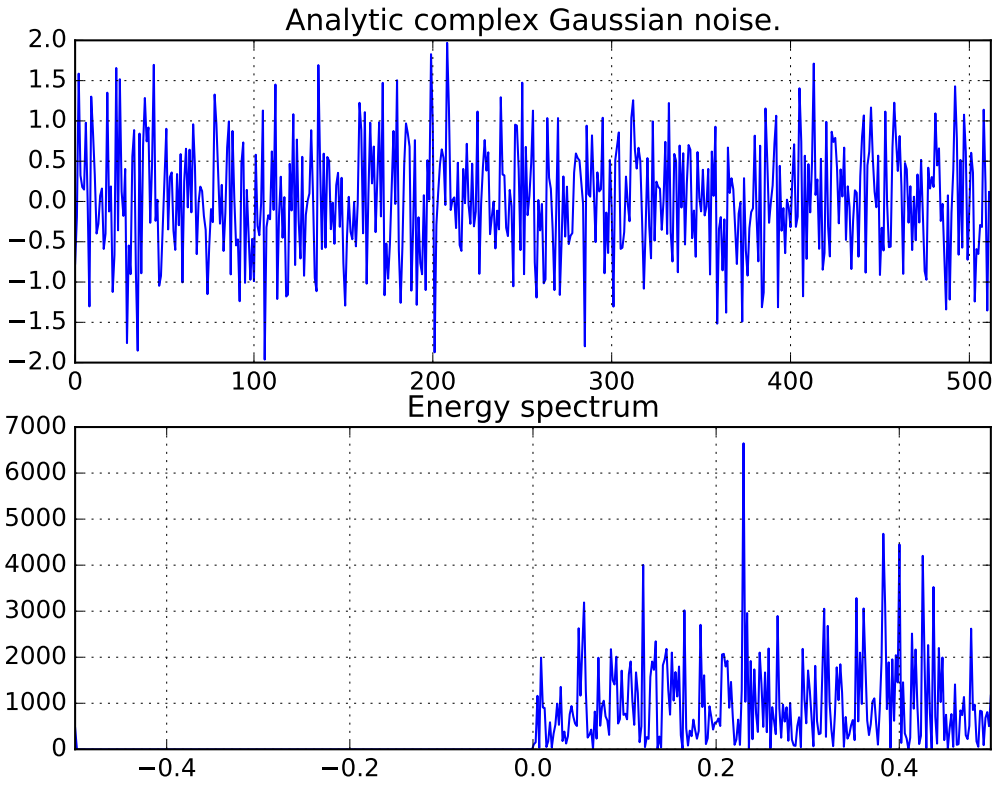
Returns Analytic complex Gaussian noise of length n_points.

Return type numpy.ndarray

Examples

```
>>> import numpy as np
>>> noise = noisecg(512)
>>> print("%.2f" % abs((noise ** 2).mean()))
0.00
>>> print("%.1f" % np.std(noise) ** 2)
1.0
```

```
>>> subplot(211), plot(real(noise))
>>> subplot(212), plot(linspace(-0.5, 0.5, 512), abs(fftshift(fft(noise))) ** 2)
```



tftb.generators.noise.**noisecu**(*n_points*)

Compute analytic complex uniform white noise.

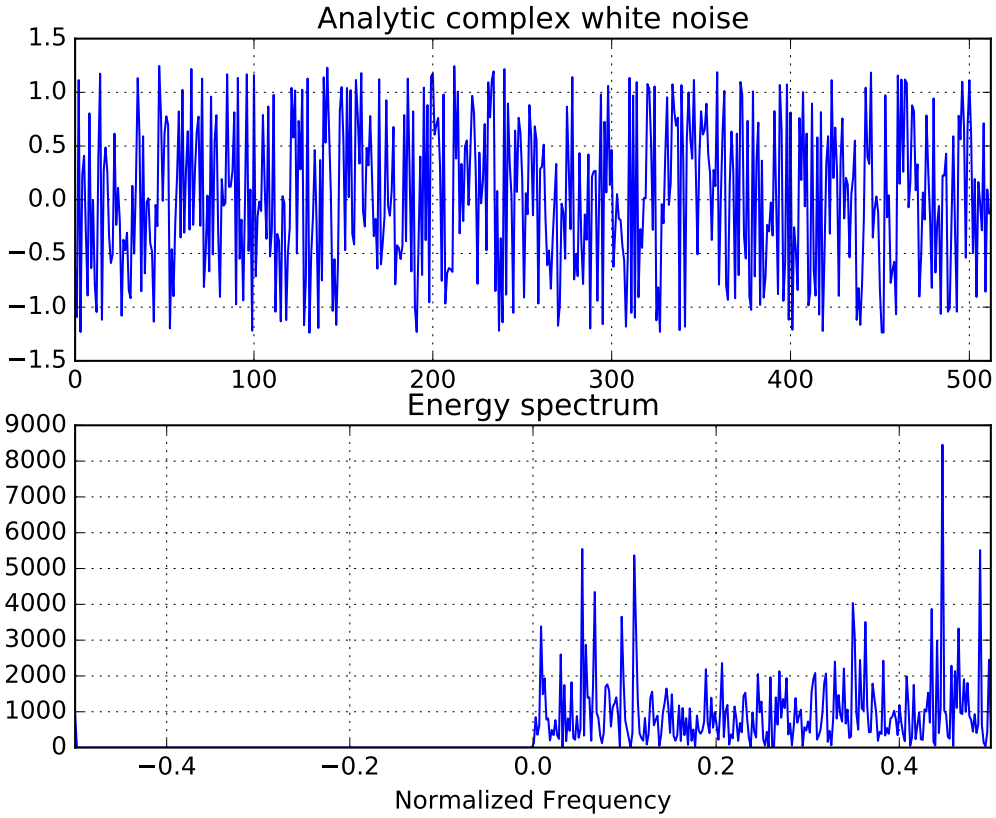
Parameters **n_points** (*int*) – Length of the noise signal.

Returns analytic complex uniform white noise signal of length N

Return type numpy.ndarray

Examples

```
>>> import numpy as np
>>> noise = noisecu(512)
>>> print("%.2f" % abs((noise ** 2).mean()))
0.00
>>> print("%.1f" % np.std(noise) ** 2)
1.0
>>> subplot(211), plot(real(noise))
>>> subplot(212), plot(linspace(-0.5, 0.5, 512), abs(fftshift(fft(noise))) ** 2)
```



tftb.generators.utils module

tftb.generators.utils.**scale**(*X*, *a*, *fmin*, *fmax*, *N*)
Scale a signal with the Mellin transform.

Parameters

- **X** (*array-like*) – signal to be scaled.
- **a** (*float*) – scale factor
- **fmin** (*float*) – lower frequency bound
- **fmax** (*float*) – higher frequency bound
- **N** (*int*) – number of analyzed voices

Returns A-scaled version of X.

Return type array-like

tftb.generators.utils.**sigmerge**(*x1*, *x2*, *ratio=0.0*)
Add two signals with a specific energy ratio in decibels.

Parameters

- **x1** (*numpy.ndarray*) – 1D numpy.ndarray
- **x2** (*numpy.ndarray*) – 1D numpy.ndarray

- **ratio** (*float*) – Energy ratio in decibels.

Returns The merged signal

Return type numpy.ndarray

Module contents

tftb.generators.**amexpos**(*n_points*, *t0=None*, *spread=None*, *kind='bilateral'*)

Exponential amplitude modulation.

amexpos generates an exponential amplitude modulation starting at time *t0* and spread proportioanl to *spread*.

Parameters

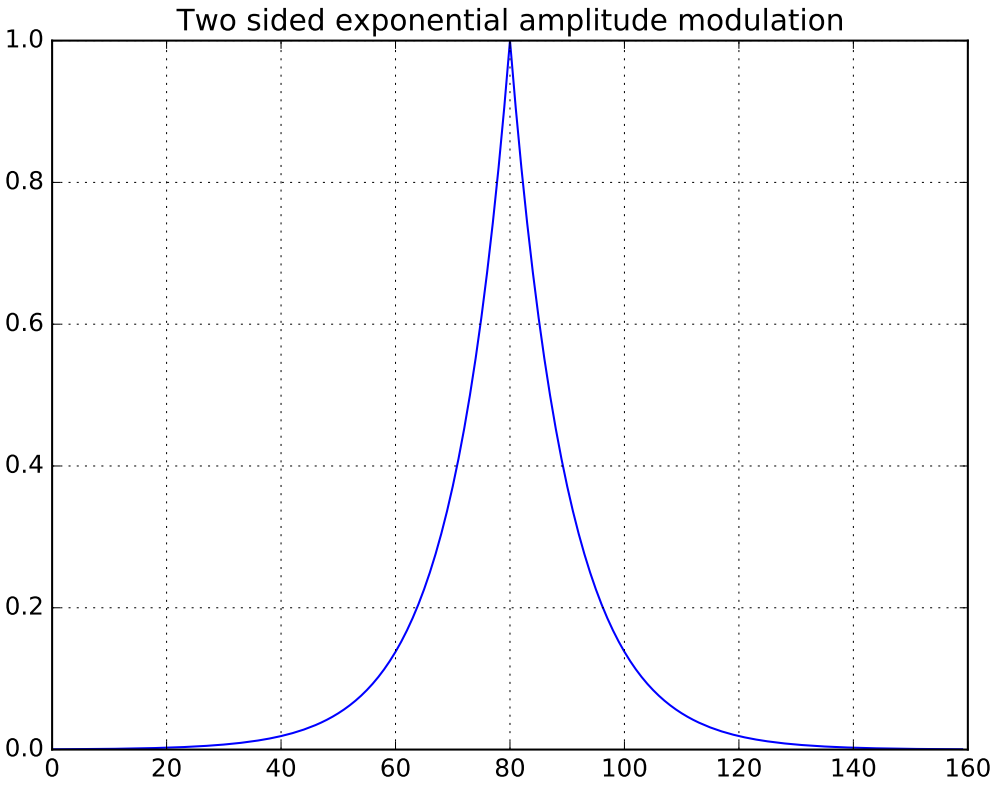
- **n_points** (*int*) – Number of points.
- **kind** (*str*) – "bilateral" (default) or "unilateral"
- **t0** (*float*) – Time center.
- **spread** (*float*) – Standard deviation.

Returns exponential function

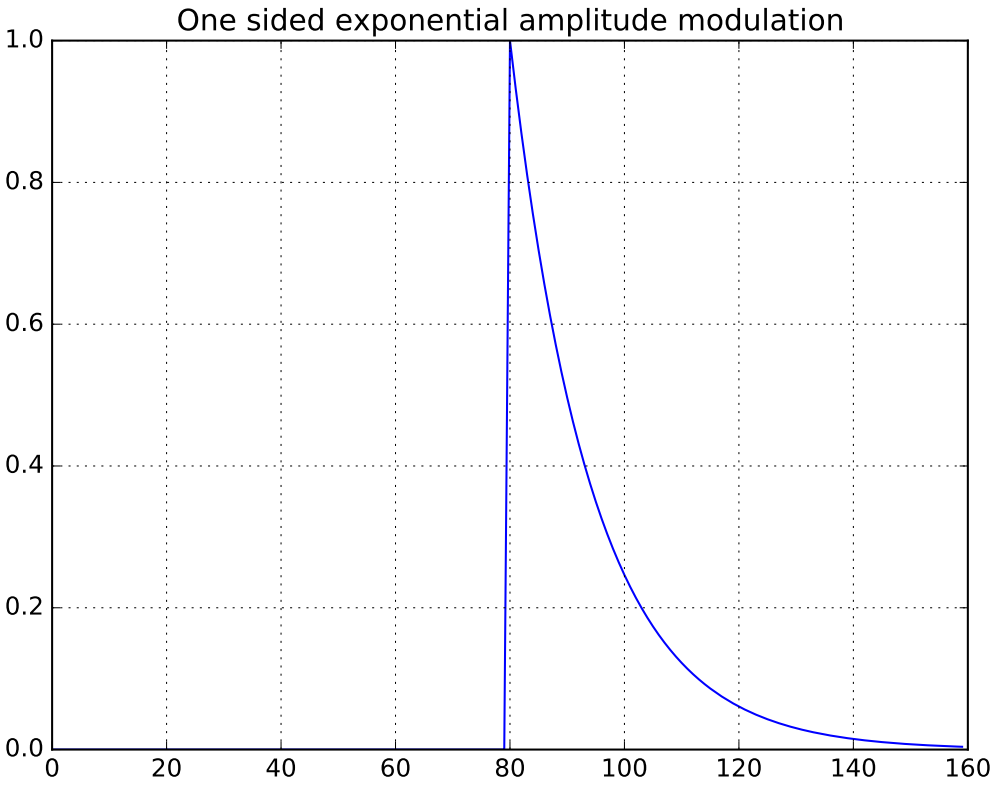
Return type numpy.ndarray

Examples

```
>>> x = amexpos(160)
>>> plot(x)
```




```
>>> x = amexpos(160, kind='unilateral')
>>> plot(x)
```



tftb.generators.**amgauss**(*n_points*, *t0=None*, *spread=None*)

Generate a Gaussian amplitude modulated signal.

Parameters

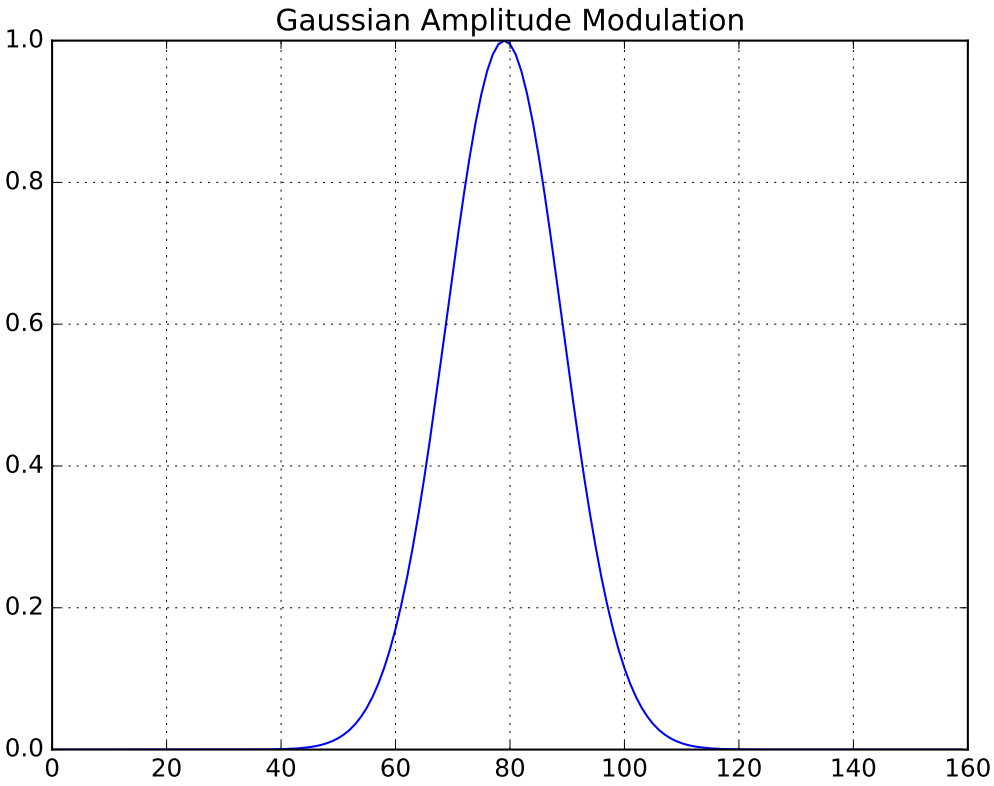
- **n_points** (*int*) – Number of points in the output.
- **t0** (*float*) – Center of the Gaussian function. (default: t0 / 2)
- **spread** (*float*) – Standard deviation of the Gaussian. (default 2 * sqrt(n_points)

Returns Gaussian function centered at time `t0`.

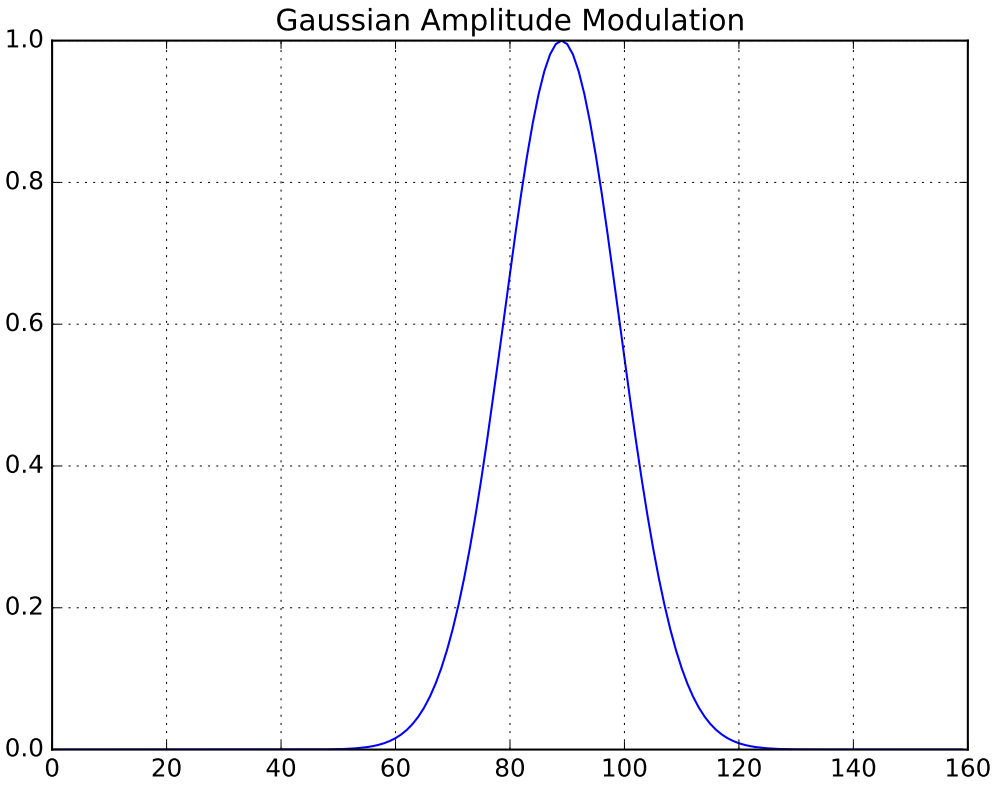
Return type numpy.ndarray

Example

```
>>> x = amgauss(160)
>>> plot(x)
```




```
>>> x = amgauss(160, 90)
>>> plot(x)
```



tftb.generators.**amrect**(*n_points*, *t0=None*, *spread=None*)

Generate a rectangular amplitude modulation.

Parameters

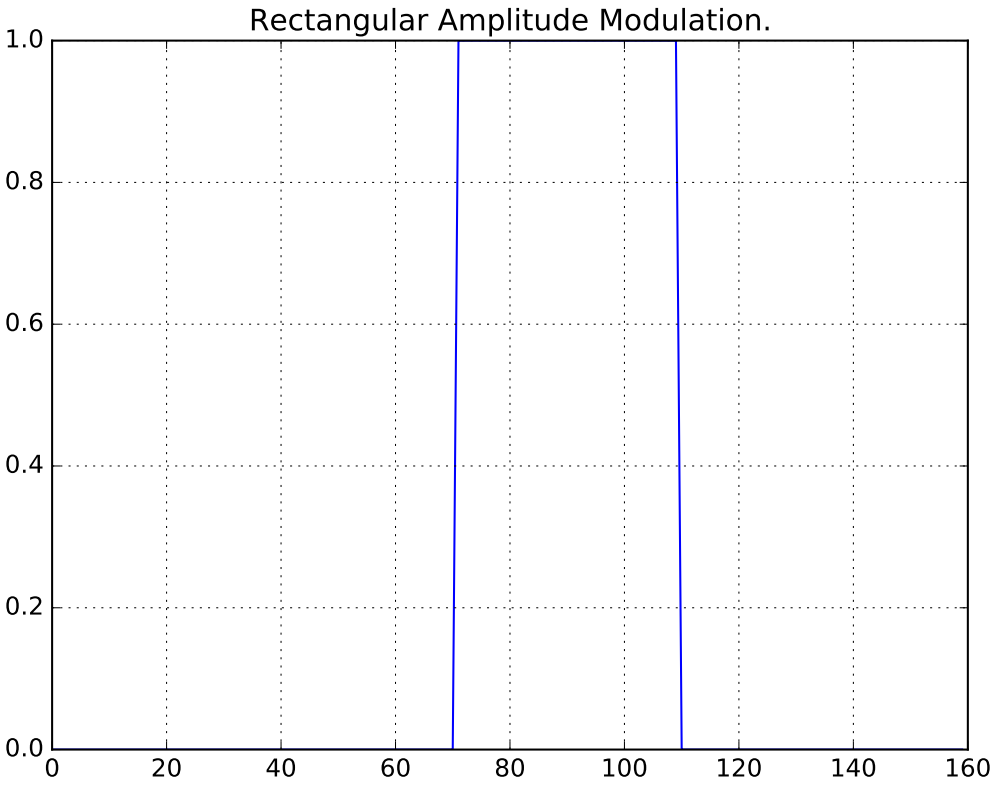
- **n_points** (*int*) – Number of points in the function.
- **t0** (*float*) – Time center
- **spread** (*float*) – standard deviation of the function.

Returns A rectangular amplitude modulator.

Return type numpy.ndarray.

Examples

```
>>> x = amrect(160, 90, 40.0)
>>> plot(x)
```



tftb.generators.**amtriang**(*n_points*, *t0=None*, *spread=None*)

Generate a triangular amplitude modulation.

Parameters

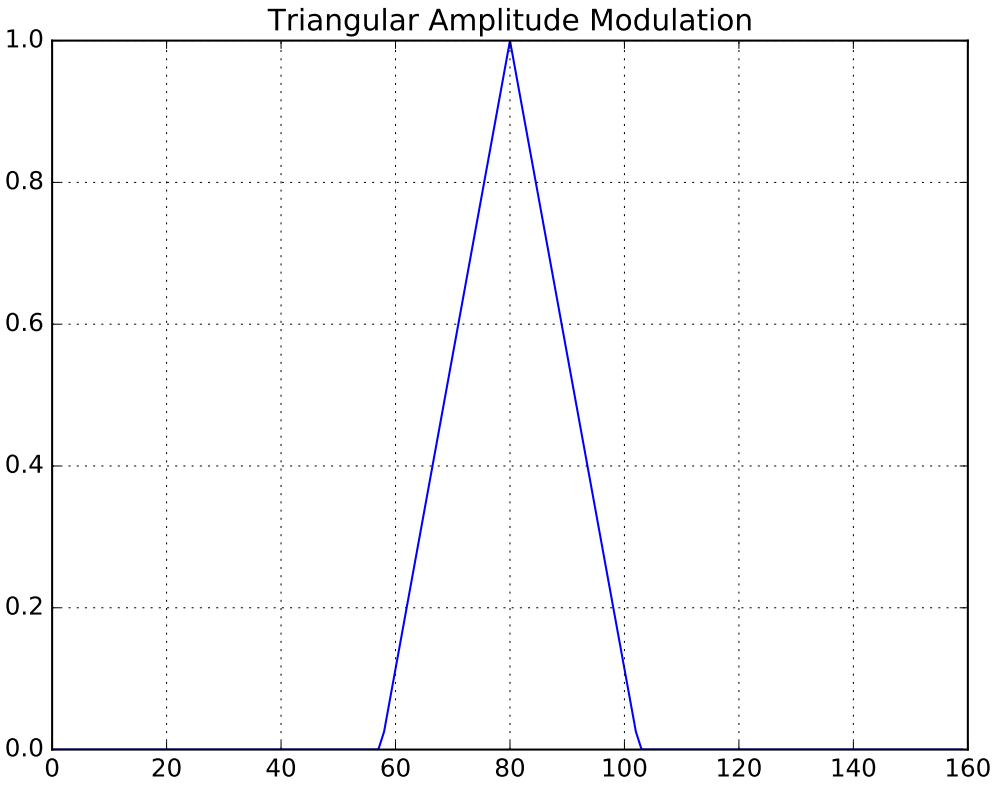
- **n_points** (*int*) – Number of points in the function.
- **t0** (*float*) – Time center
- **spread** (*float*) – standard deviation of the function.

Returns A triangular amplitude modulator.

Return type numpy.ndarray.

Examples

```
>>> x = amtriang(160)
>>> plot(x)
```



tftb.generators.**fmconst**(*n_points*, *fnorm=0.25*, *t0=None*)

Generate a signal with constant frequency modulation.

Parameters

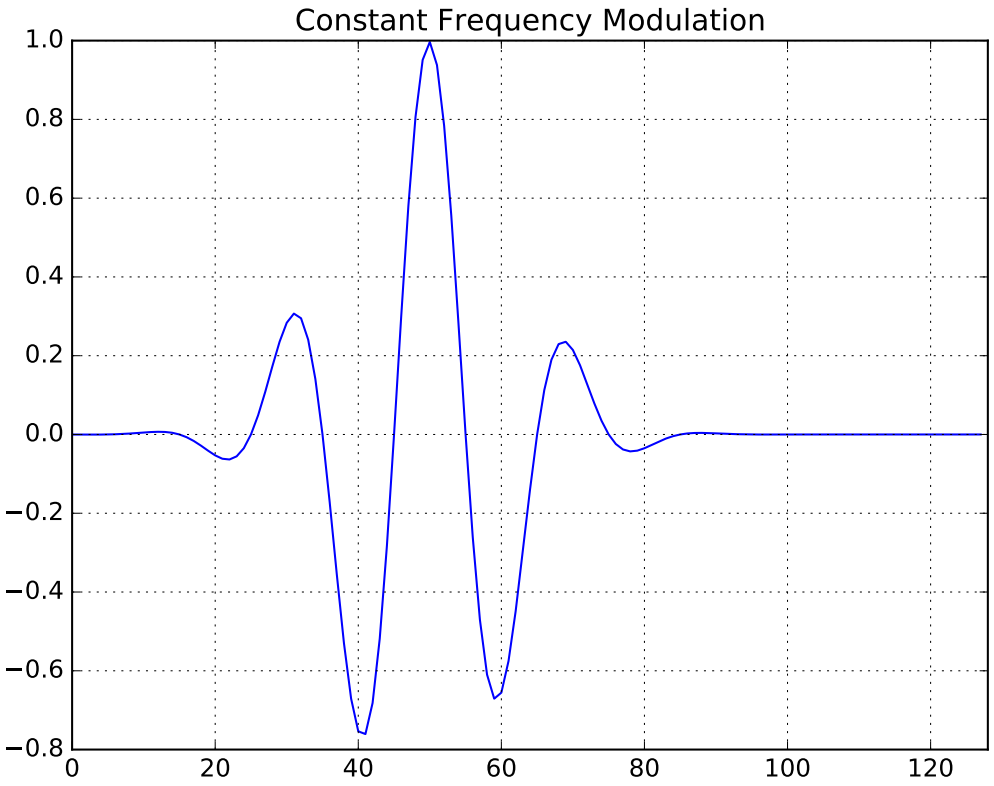
- **n_points** (*int*) – number of points
- **fnorm** (*float*) – normalized frequency
- **t0** (*float*) – time center

Returns frequency modulation signal with frequency fnorm

Return type numpy.ndarray

Examples

```
>>> from tftb.generators import amgauss
>>> z = amgauss(128, 50, 30) * fmconst(128, 0.05, 50)[0]
>>> plot(real(z))
```



tftb.generators.**fmhyp**(*n_points*, *p1*, *p2*)

Signal with hyperbolic frequency modulation.

Parameters

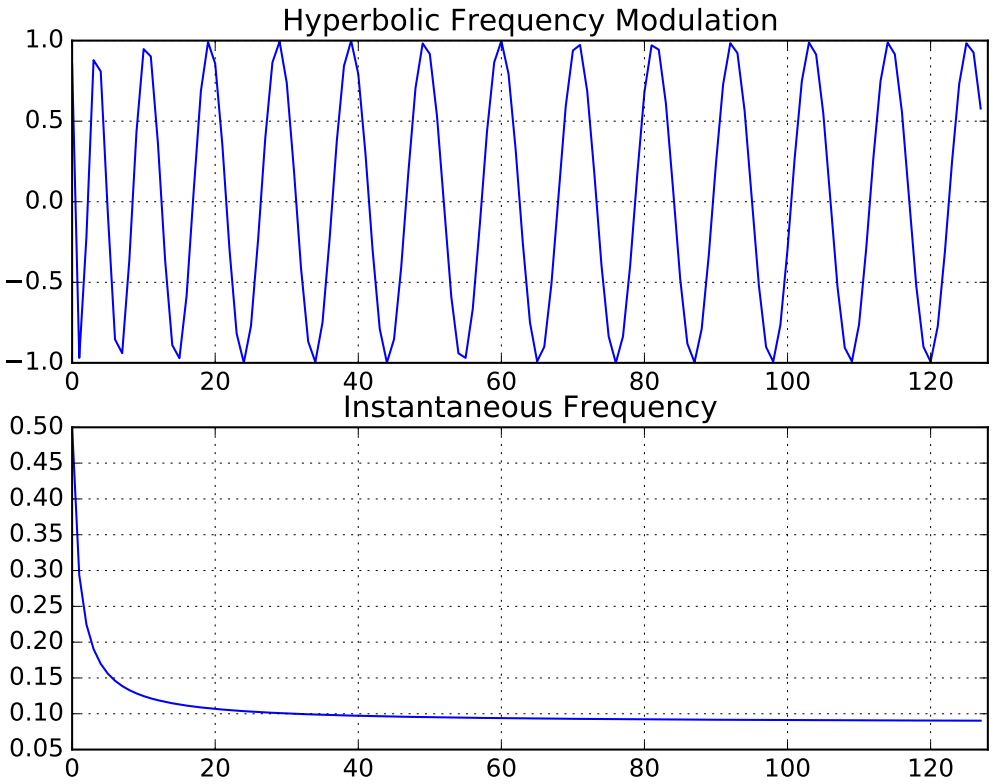
- **n_points** (*int*) – number of points.
- **p2** (*float*) – coefficients of the hyperbolic function.

Returns vector containing the modulated signal samples.

Return type numpy.ndarray

Examples

```
>>> signal, iflaw = fmhyp(128, (1, 0.5), (32, 0.1))
>>> subplot(211), plot(real(signal))
>>> subplot(212), plot(iflaw)
```



tftb.generators.**fmlin**(*n_points*, *fnormi=0.0*, *fnormf=0.5*, *t0=None*)

Generate a signal with linear frequency modulation.

Parameters

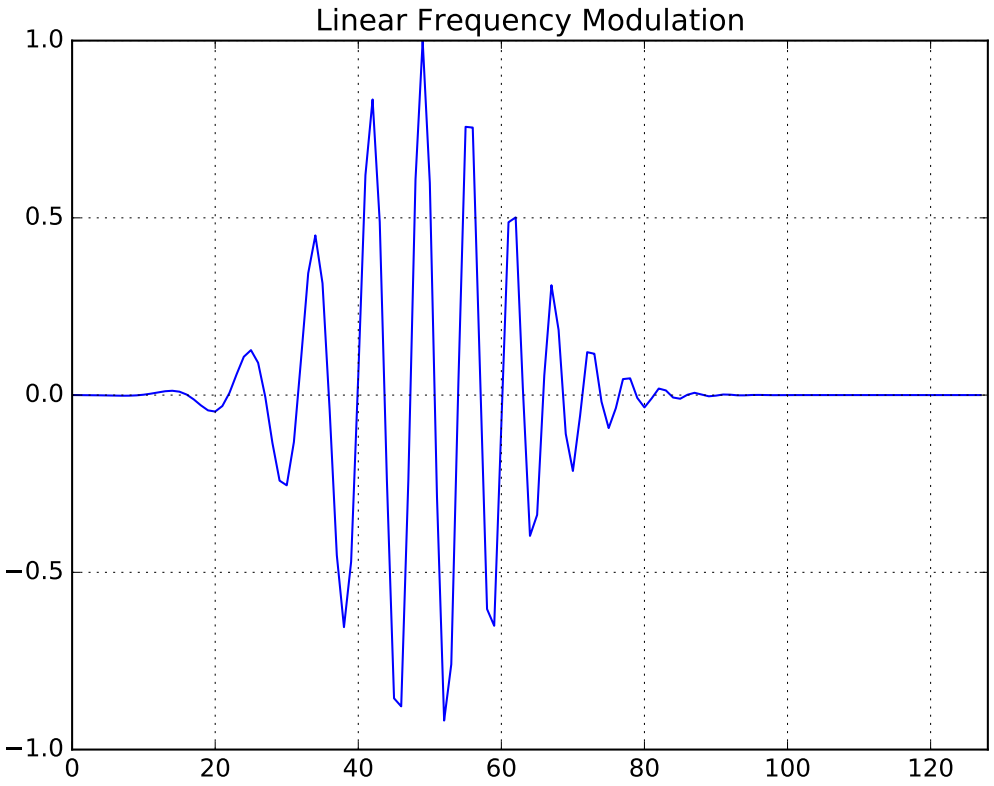
- **n_points** (*int*) – number of points
- **fnormi** (*float*) – initial normalized frequency
- **fnormf** (*float*) – final normalized frequency
- **t0** (*float*) – time center

Returns The modulated signal, and the instantaneous amplitude law.

Return type tuple(array-like)

Examples

```
>>> from tftb.generators import amgauss
>>> z = amgauss(128, 50, 40) * fmlin(128, 0.05, 0.3, 50)[0]
>>> plot(real(z))
```



tftb.generators.**fmodany**(*iflaw*, *t0=1*)

Arbitrary frequency modulation.

Parameters

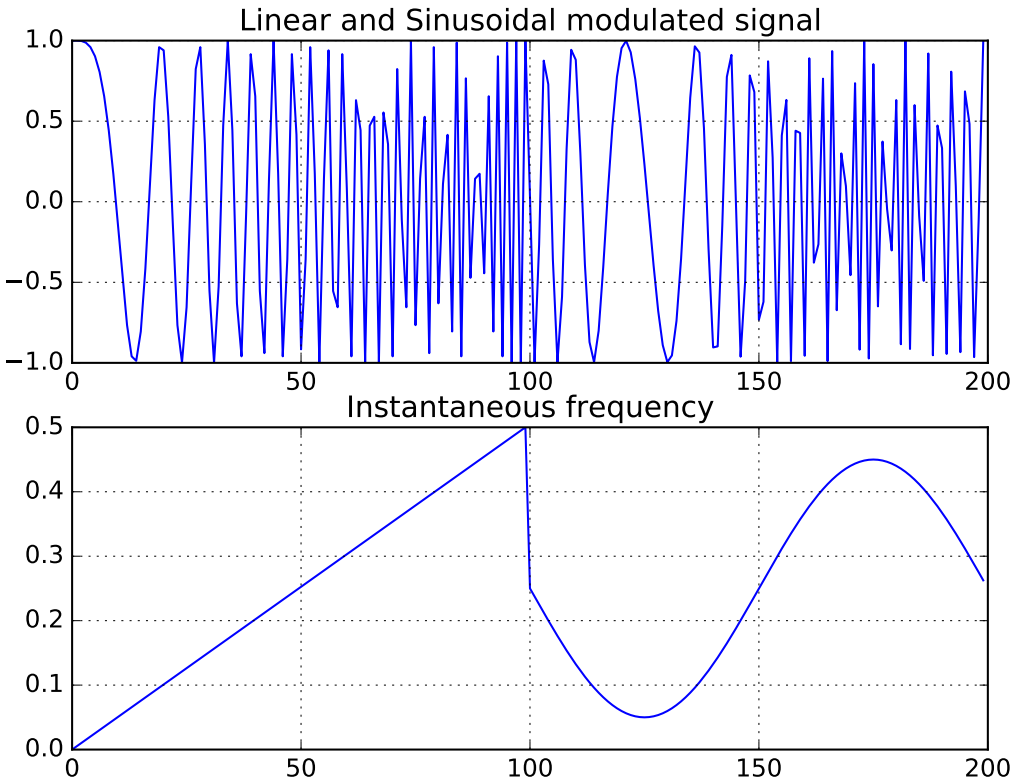
- **iflaw** (*numpy.ndarray*) – Vector of instantaneous frequency law samples.
- **t0** (*float*) – time center

Returns output signal

Return type

Examples

```
>>> from tftb.generators import fmlin
>>> import numpy as np
>>> y1, ifl1 = fmlin(100)  # A linear instantaneous frequency law.
>>> y2, ifl2 = fmsin(100)  # A sinusoidal instantaneous frequency law.
>>> iflaw = np.append(ifl1, ifl2)  # combination of the two
>>> sig = fmodany(iflaw)
>>> subplot(211), plot(real(sig))
>>> subplot(212), plot(iflaw)
```



tftb.generators.**fmpar**(*n_points*, *coefficients*)

Parabolic frequency modulated signal.

Parameters

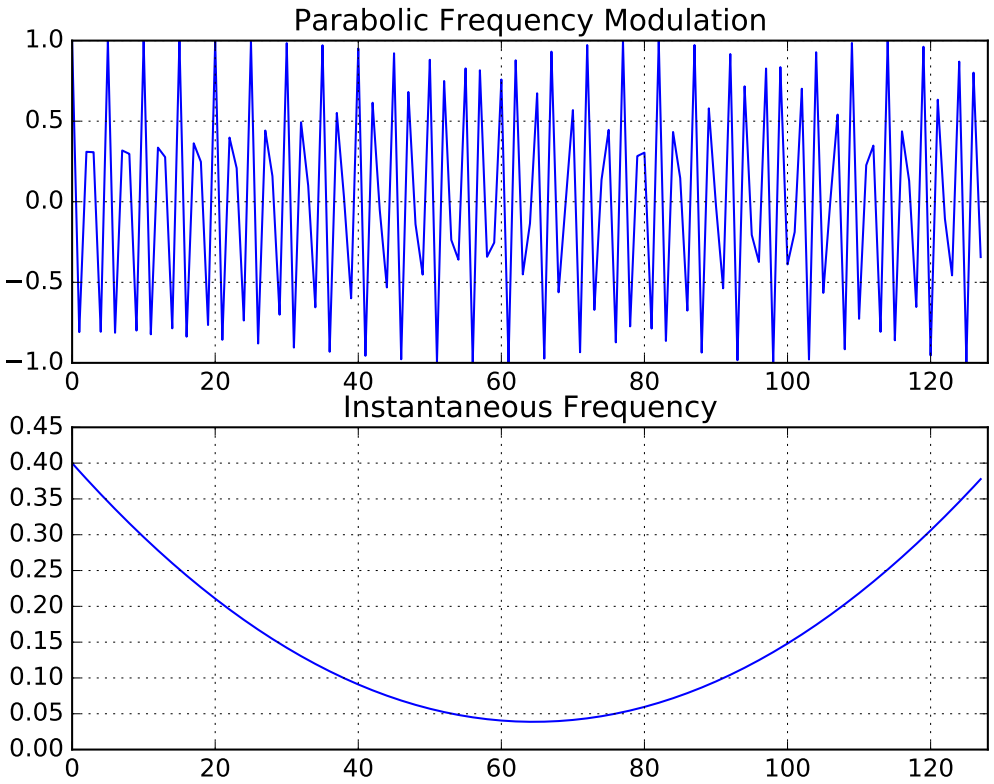
- **n_points** (*int*) – number of points
- **coefficients** (*tuple*) – coefficients of the parabolic function.

Returns Signal with parabolic frequency modulation law.

Return type tuple

Examples

```
>>> x, iflaw = fmpar(128, (0.4, -0.0112, 8.6806e-05))
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(iflaw)
```



tftb.generators.**fmpower**(*n_points*, *k*, *coefficients*)

Generate signal with power law frequency modulation.

Parameters

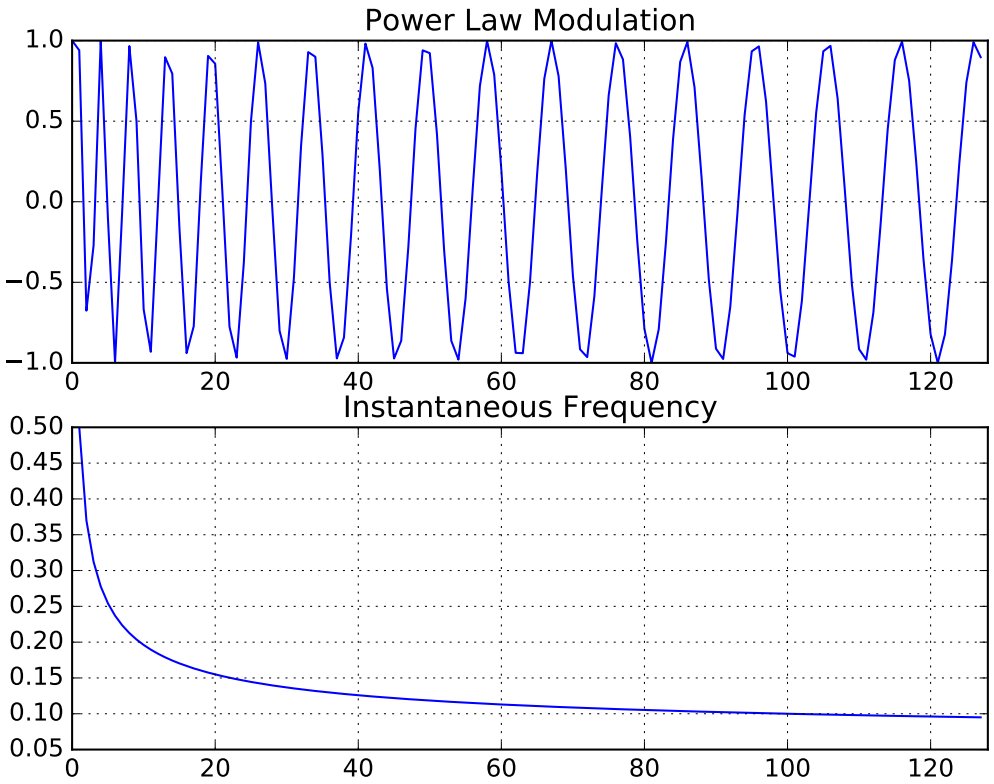
- **n_points** (*int*) – number of points.
- **k** (*int*) – degree of power law.
- **p2** (*float*) – coefficients of the power law.

Returns vector of modulated signal samples.

Return type numpy.ndarray

Examples

```
>>> x, iflaw = fmpower(128, 0.5, (1, 0.5, 100, 0.1))
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(iflaw)
```



tftb.generators.**fmsin**(*n_points*, *fnormin=0.05*, *fnormax=0.45*, *period=None*, *t0=None*, *fnorm0=None*, *pm1=1*)

Sinusodial frequency modulation.

Parameters

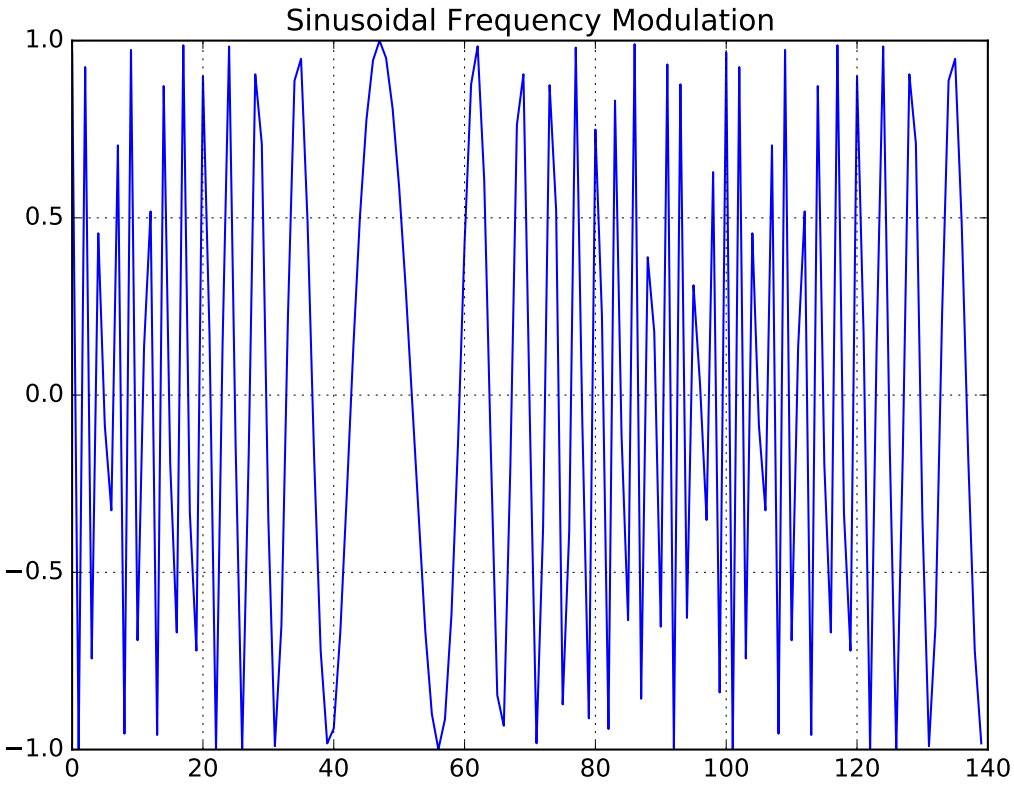
- **n_points** (*int*) – number of points
- **fnormin** (*float*) – smallest normalized frequency
- **fnormax** (*float*) – highest normalized frequency
- **period** (*int*) – period of sinusoidal fm
- **t0** (*float*) – time reference
- **fnorm0** (*float*) – normalized frequency at time t0
- **pm1** (*int*) – frequency direction at t0

Returns output signal

Return type numpy.ndarray

Examples

```
>>> z = fmsin(140, period=100, t0=20, fnorm0=0.3, pm1=-1)
>>> plot(real(z))
```



tftb.generators.**sigmerge**(*x1*, *x2*, *ratio=0.0*)

Add two signals with a specific energy ratio in decibels.

Parameters

- **x1** (*numpy.ndarray*) – 1D numpy.ndarray
- **x2** (*numpy.ndarray*) – 1D numpy.ndarray
- **ratio** (*float*) – Energy ratio in decibels.

Returns The merged signal

Return type numpy.ndarray

tftb.generators.**scale**(*X*, *a*, *fmin*, *fmax*, *N*)

Scale a signal with the Mellin transform.

Parameters

- **X** (*array-like*) – signal to be scaled.
- **a** (*float*) – scale factor
- **fmin** (*float*) – lower frequency bound
- **fmax** (*float*) – higher frequency bound
- **N** (*int*) – number of analyzed voices

Returns A-scaled version of X.

Return type array-like

tftb.generators.**dopnoise**(*n_points*, *s_freq*, *f_target*, *distance*, *v_target*, *time_center=None*, *c=340*)

Generate complex noisy doppler signal, normalized to have unit energy.

Parameters

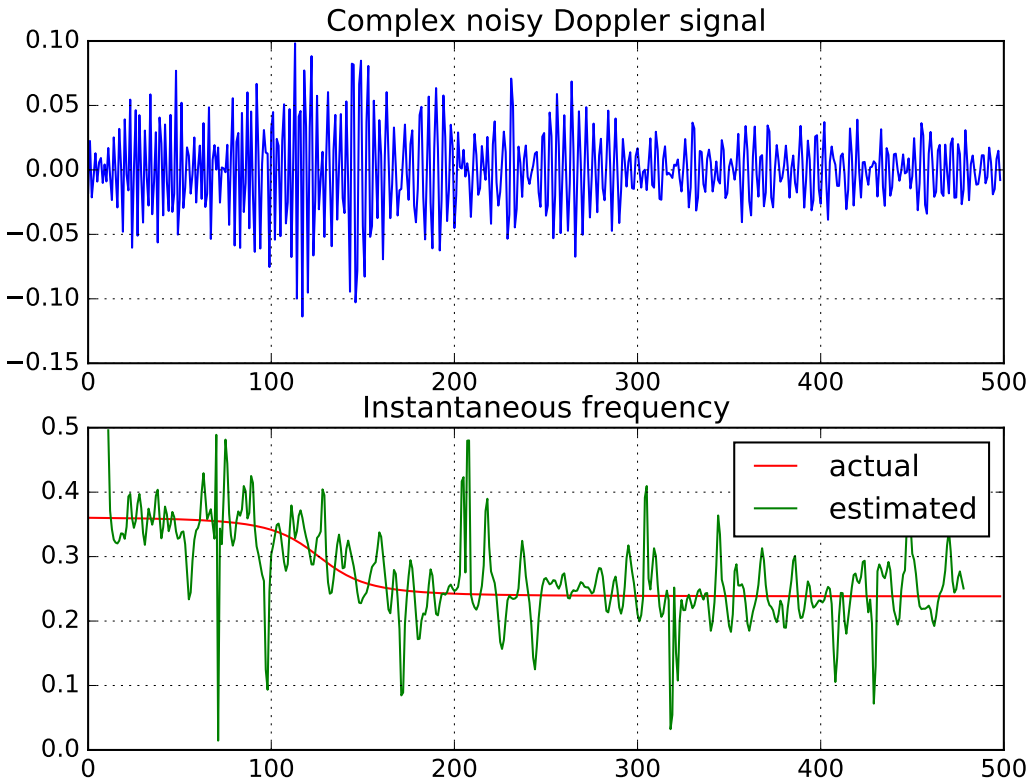
- **n_points** (*int*) – Number of points.
- **s_freq** (*float*) – Sampling frequency.
- **f_target** (*float*) – Frequency of target.
- **distance** (*float*) – Distnace from line to observer.
- **v_target** (*float*) – velocity of target relative to observer.
- **time_center** (*float*) – Time center. (Default n_points / 2)
- **c** (*float*) – Wave velocity (Default 340 m/s)

Returns tuple (output signal, instantaneous frequency law.)

Return type tuple(array-like)

Example

```
>>> import numpy as np
>>> from tftb.processing import inst_freq
>>> z, iflaw = dopnoise(500, 200.0, 60.0, 10.0, 70.0, 128.0)
>>> subplot(211), plot(real(z))
>>> ifl = inst_freq(z, np.arange(11, 479), 10)
>>> subplot(212), plot(iflaw, 'r', ifl, 'g')
```



tftb.generators.**noisecg**(*n_points*, *a1=None*, *a2=None*)

Generate analytic complex gaussian noise with mean 0.0 and variance 1.0.

Parameters

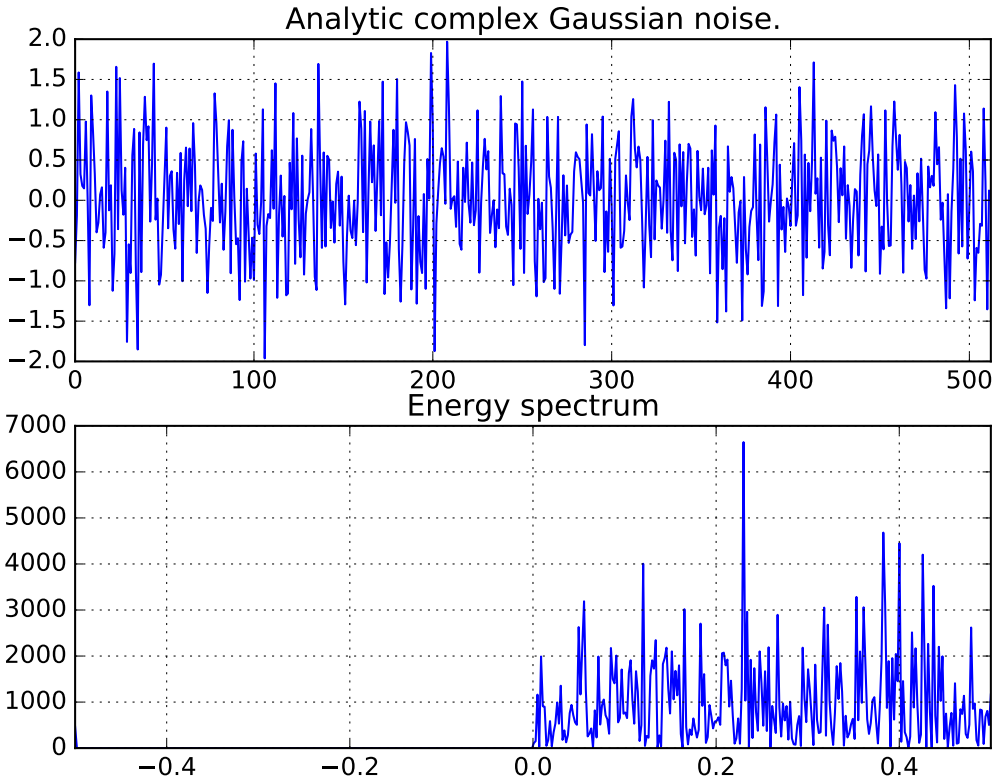
- **n_points** (*int*) – Length of the desired output signal.
- **a1** (*float*) – Coefficients of the filter through which the noise is passed.
- **a2** (*float*) – Coefficients of the filter through which the noise is passed.

Returns Analytic complex Gaussian noise of length n_points.

Return type numpy.ndarray

Examples

```
>>> import numpy as np
>>> noise = noisecg(512)
>>> print("%.2f" % abs((noise ** 2).mean()))
0.00
>>> print("%.1f" % np.std(noise) ** 2)
1.0
>>> subplot(211), plot(real(noise))
>>> subplot(212), plot(linspace(-0.5, 0.5, 512), abs(fftshift(fft(noise))) ** 2)
```



tftb.generators.**noisecu**(*n_points*)

Compute analytic complex uniform white noise.

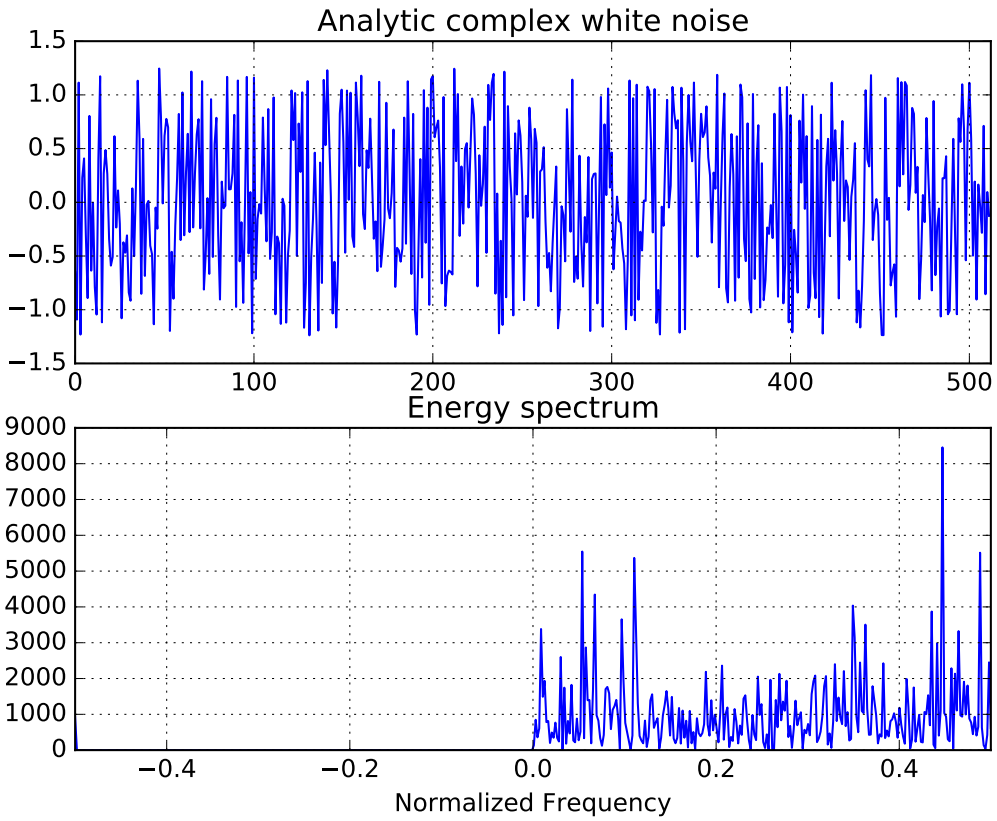
Parameters **n_points** (*int*) – Length of the noise signal.

Returns analytic complex uniform white noise signal of length N

Return type numpy.ndarray

Examples

```
>>> import numpy as np
>>> noise = noisecu(512)
>>> print("%.2f" % abs((noise ** 2).mean()))
0.00
>>> print("%.1f" % np.std(noise) ** 2)
1.0
>>> subplot(211), plot(real(noise))
>>> subplot(212), plot(linspace(-0.5, 0.5, 512), abs(fftshift(fft(noise))) ** 2)
```



tftb.generators.**anaask**(*n_points*, *n_comp=None*, *f0=0.25*)

Generate an amplitude shift (ASK) keying signal.

Parameters

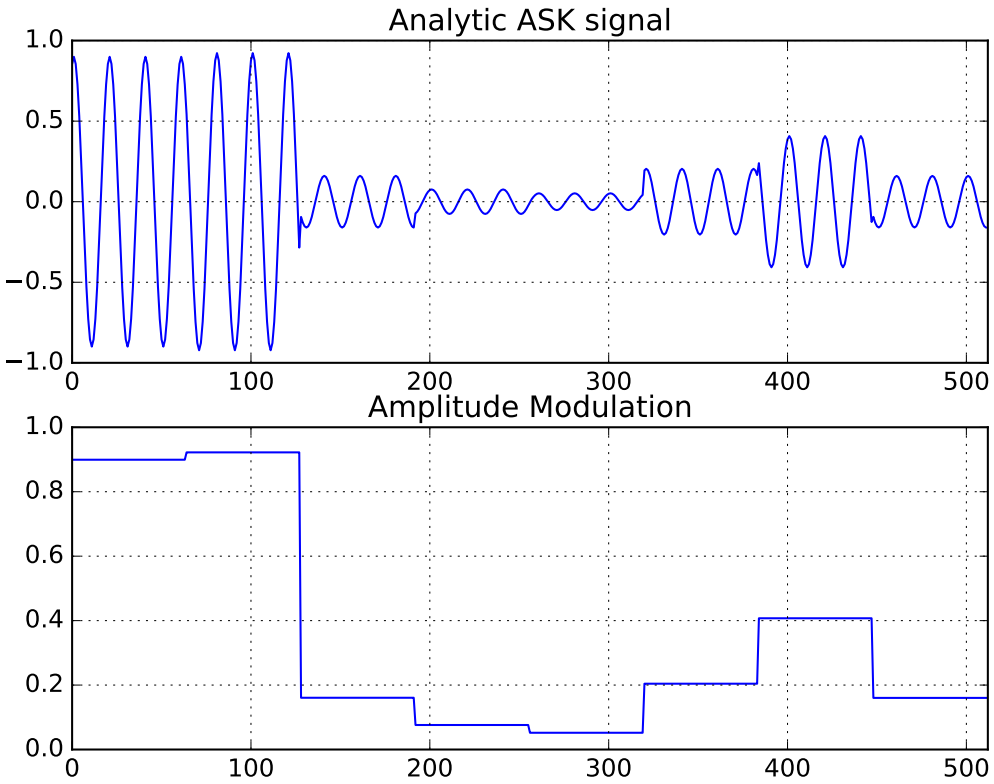
- **n_points** (*int*) – number of points.
- **n_comp** (*int*) – number of points of each component.
- **f0** (*float*) – normalized frequency.

Returns Tuple containing the modulated signal and the amplitude modulation.

Return type tuple(numpy.ndarray)

Examples

```
>>> x, am = anaask(512, 64, 0.05)
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(am)
```



tftb.generators.**anabpsk**(*n_points*, *n_comp=None*, *f0=0.25*)

Binary phase shift keying (BPSK) signal.

Parameters

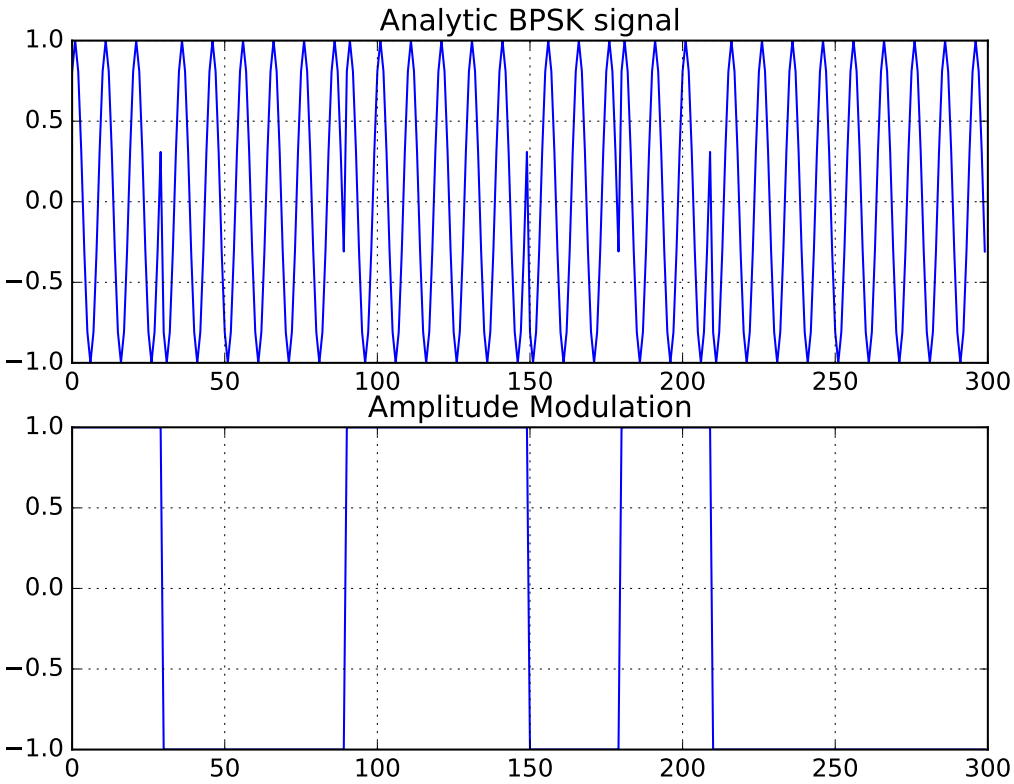
- **n_points** (*int*) – number of points.
- **n_comp** (*int*) – number of points in each component.
- **f0** (*float*) – normalized frequency.

Returns BPSK signal

Return type numpy.ndarray

Examples

```
>>> x, am = anabpsk(300, 30, 0.1)
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(am)
```



tftb.generators.**anafsk**(*n_points*, *n_comp=None*, *Nbf=4*)

Frequency shift keying (FSK) signal.

Parameters

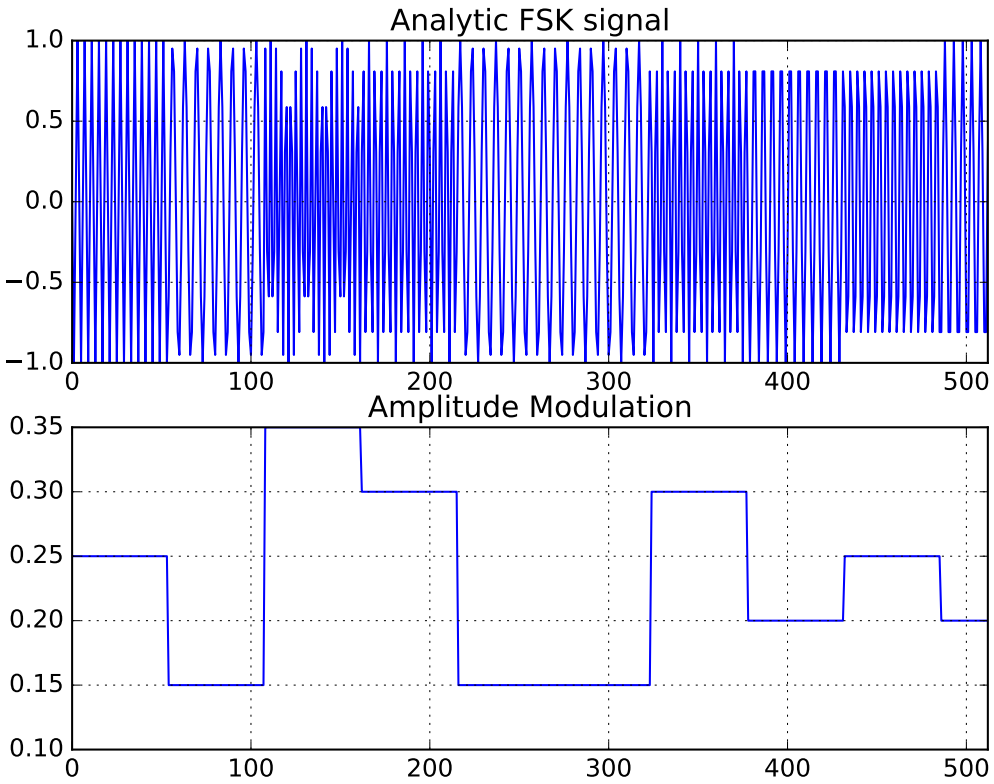
- **n_points** (*int*) – number of points.
- **n_comp** (*int*) – number of points in each components.
- **Nbf** (*int*) – number of distinct frequencies.

Returns FSK signal.

Return type numpy.ndarray

Examples

```
>>> x, am = anafsk(512, 54.0, 5.0)
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(am)
```



tftb.generators.**anapulse**(*n_points*, *ti=None*)

Analytic projection of unit amplitude impulse signal.

Parameters

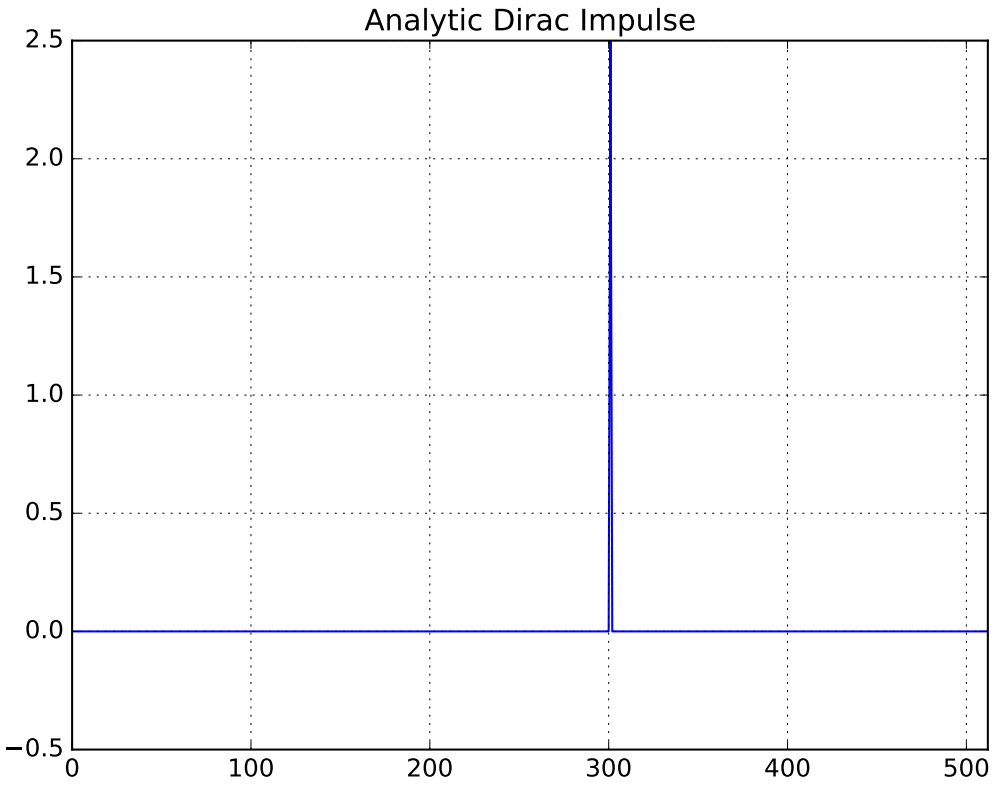
- **n_points** (*int*) – Number of points.
- **ti** (*float*) – time position of the impulse.

Returns analytic impulse signal.

Return type numpy.ndarray

Examples

```
>>> x = 2.5 * anapulse(512, 301)
>>> plot(real(x))
```



tftb.generators.**anaqpsk**(*n_points*, *n_comp=None*, *f0=0.25*)

Quaternary Phase Shift Keying (QPSK) signal.

Parameters

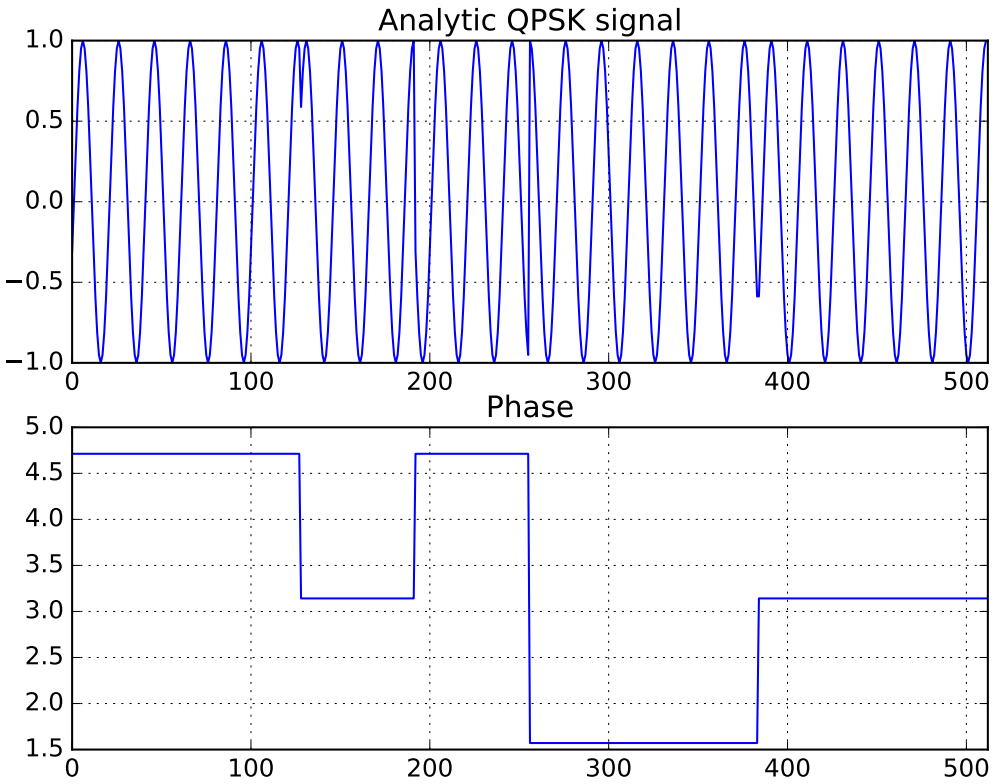
- **n_points** (*int*) – number of points.
- **n_comp** (*int*) – number of points in each component.
- **f0** (*float*) – normalized frequency

Returns complex phase modulated signal of normalized frequency f0 and initial phase.

Return type tuple

Examples

```
>>> x, phase = anaqpsk(512, 64.0, 0.05)
>>> subplot(211), plot(real(x))
>>> subplot(212), plot(phase)
```



tftb.generators.**anasing**(*n_points*, *t0=None*, *h=0.0*)

Lipschitz singularity. Refer to the wiki page on *Lipschitz condition*, good test case.

Parameters

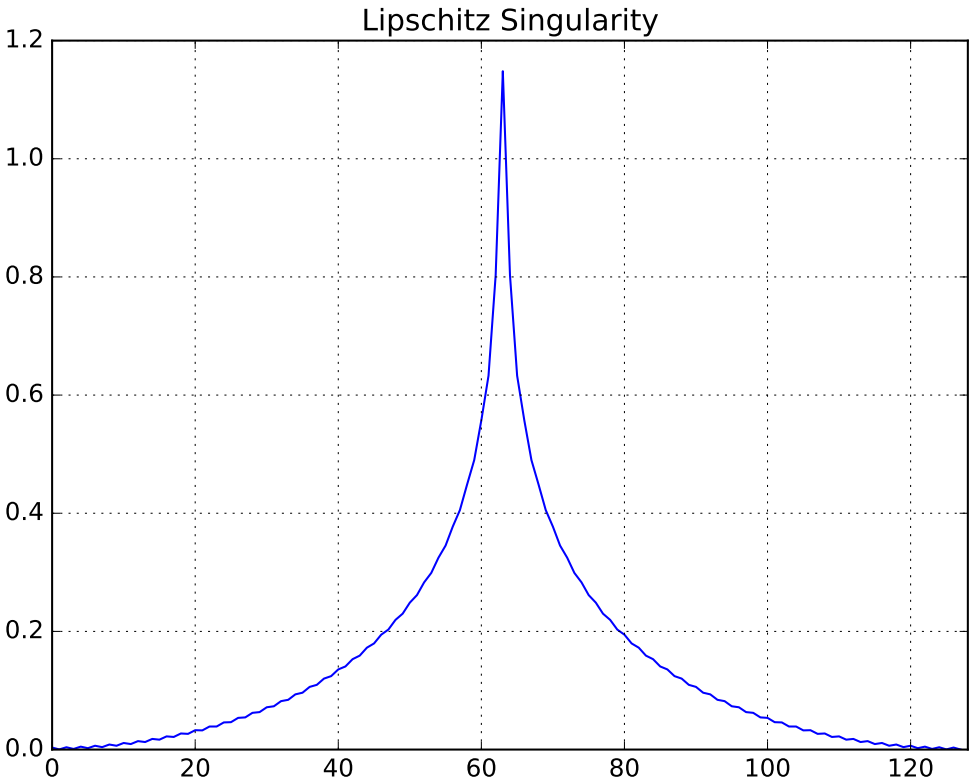
- **n_points** (*int*) – number of points in time.
- **t0** (*float*) – time localization of singularity
- **h** (*float*) – strength of the singularity

Returns N-point Lipschitz singularity centered around t0

Return type numpy.ndarray

Examples

```
>>> x = anasing(128)
>>> plot(real(x))
```



tftb.generators.**anastep**(*n_points*, *ti=None*)

Analytic projection of unit step signal.

Parameters

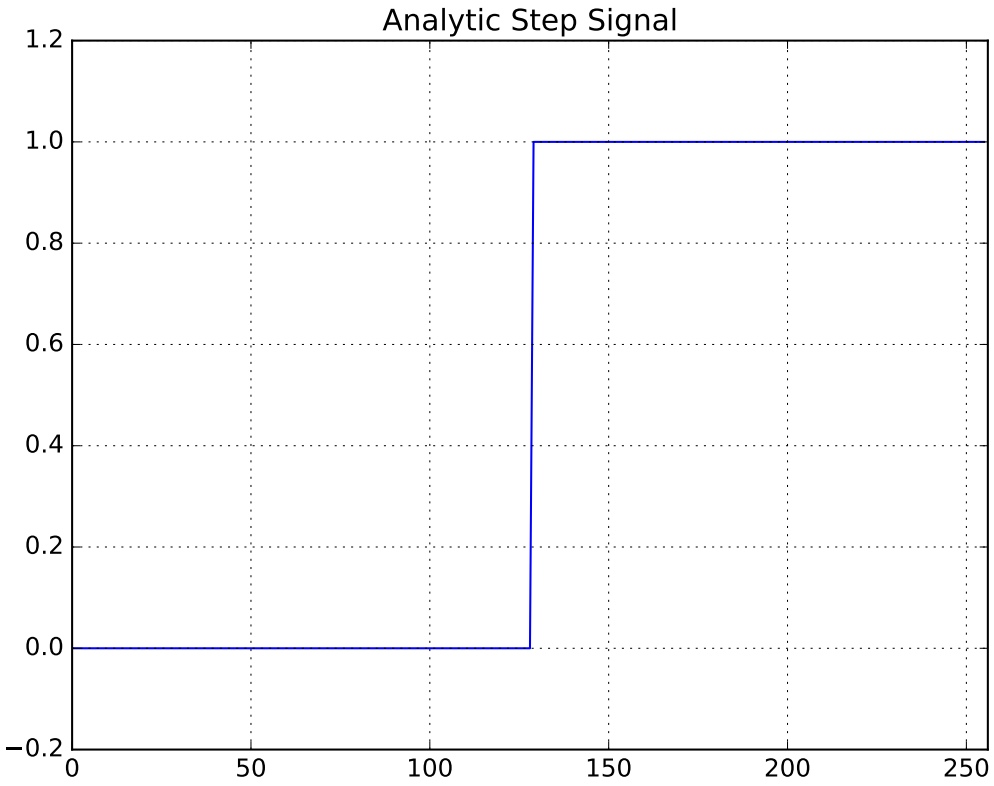
- **n_points** (*int*) – Number of points.
- **ti** (*float*) – starting position of unit step.

Returns output signal

Return type numpy.ndarray

Examples

```
>>> x = anastep(256, 128)
>>> plot(real(x))
```



tftb.generators.**doppler**(*n_points*, *s_freq*, *f0*, *distance*, *v_target*, *t0=None*, *v_wave=340.0*)

Generate complex Doppler signal

Parameters

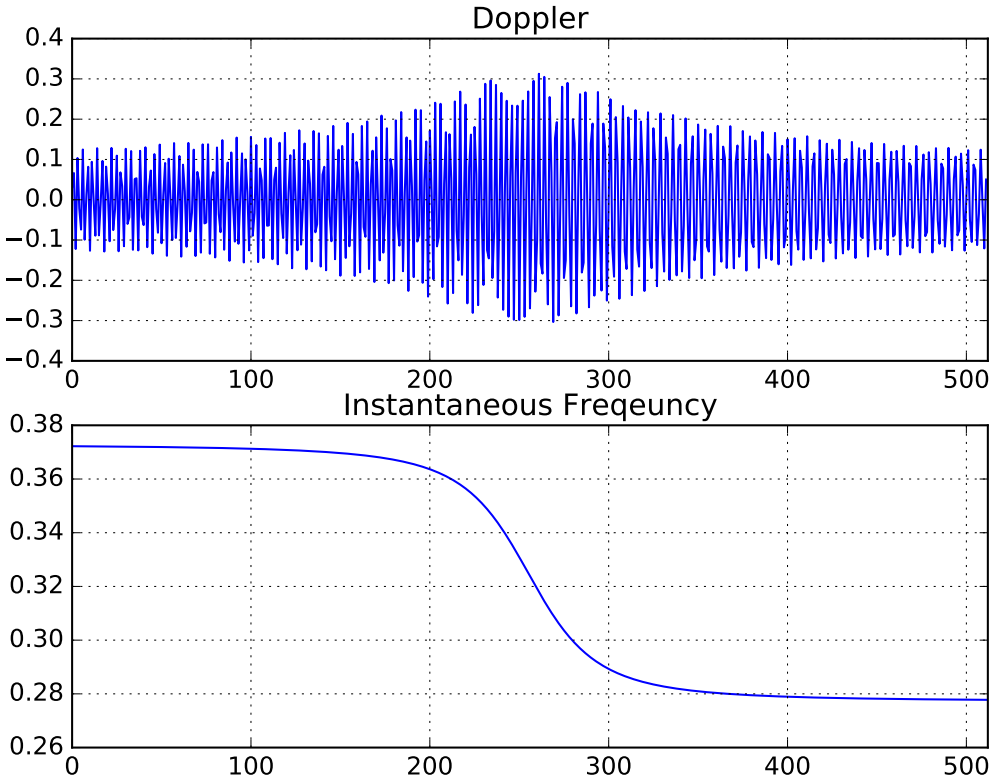
- **n_points** (*int*) – Number of points
- **s_freq** (*float*) – Sampling frequency
- **f0** (*float*) – Target frequency
- **distance** (*float*) – distance from line to observer
- **v_target** (*float*) – Target velocity
- **t0** (*float*) – Time center
- **v_wave** (*float*) – wave velocity.

Returns Tuple containing output frequency modulator, output amplitude modulator, output instantaneous frequency law.

Return type tuple

Example

```
>>> fm, am, iflaw = doppler(512, 200.0, 65.0, 10.0, 50.0)
>>> subplot(211), plot(real(am * fm))
>>> subplot(211), plot(iflaw)
```



tftb.generators.**gdpower**(*n_points*, *degree=0.0*, *rate=1.0*)

Generate a signal with a power law group delay.

Parameters

- **n_points** (*int*) – Number of points in time.
- **degree** (*int # yoder: i'm pretty sure this is supposed to be a float. at least in test_misc.py, the test for this funct. passes degree=0.5*) – degree of the power law.
- **rate** (*float*) – rate-coefficient of the power law GD.

Returns Tuple of time row containing modulated samples, group delay, frequency bins.

Return type tuple

tftb.generators.**klauder**(*n_points*, *attenuation=10.0*, *f0=0.2*)

Klauder wavelet in time domain.

Parameters

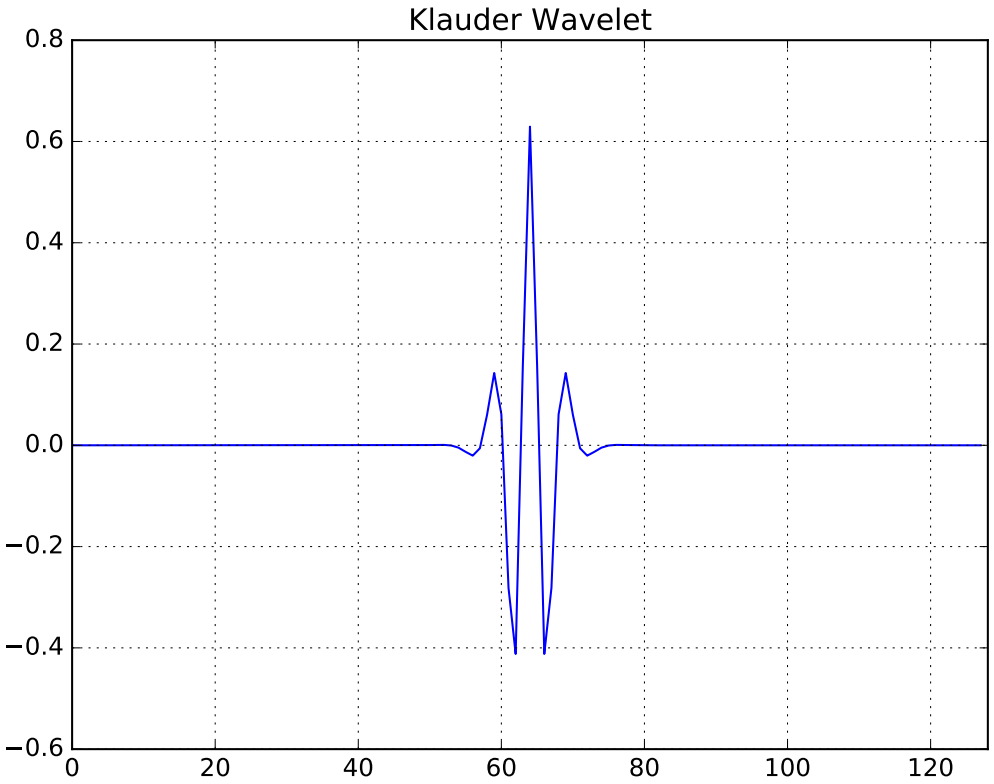
- **n_points** (*int*) – Number of points in time.
- **attenuation** (*float*) – attenuation factor of the envelope.
- **f0** (*float*) – central frequency of the wavelet.

Returns Time row vector containing klauder samples.

Return type numpy.ndarray

Example

```
>>> x = klauder(128)
>>> plot(x)
```



tftb.generators.**mexhat**(*nu=0.05*)

Mexican hat wavelet in time domain.

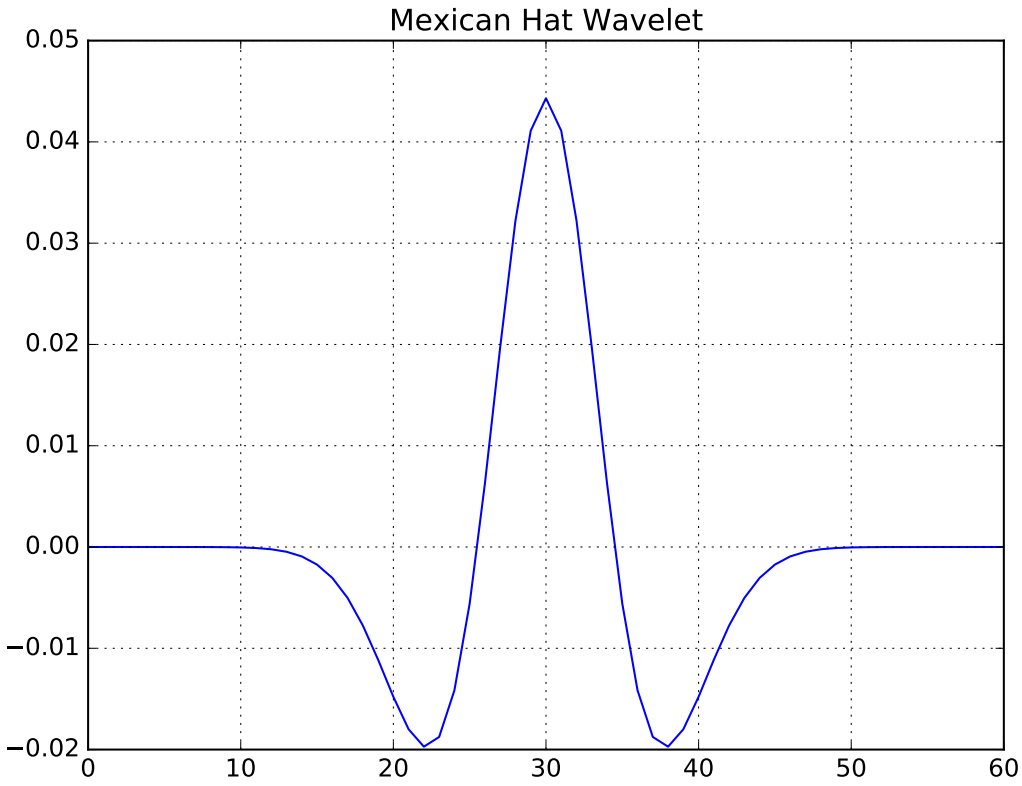
Parameters **nu** (*float*) – Central normalized frequency of the wavelet. Must be a real number between 0 and 0.5

Returns time vector containing mexhat samples.

Return type numpy.ndarray

Example

```
>>> plot(mexhat())
```



tftb.generators.**altes**(*n_points*, *fmin=0.05*, *fmax=0.5*, *alpha=300*)

Generate the Altes signal in time domain.

Parameters

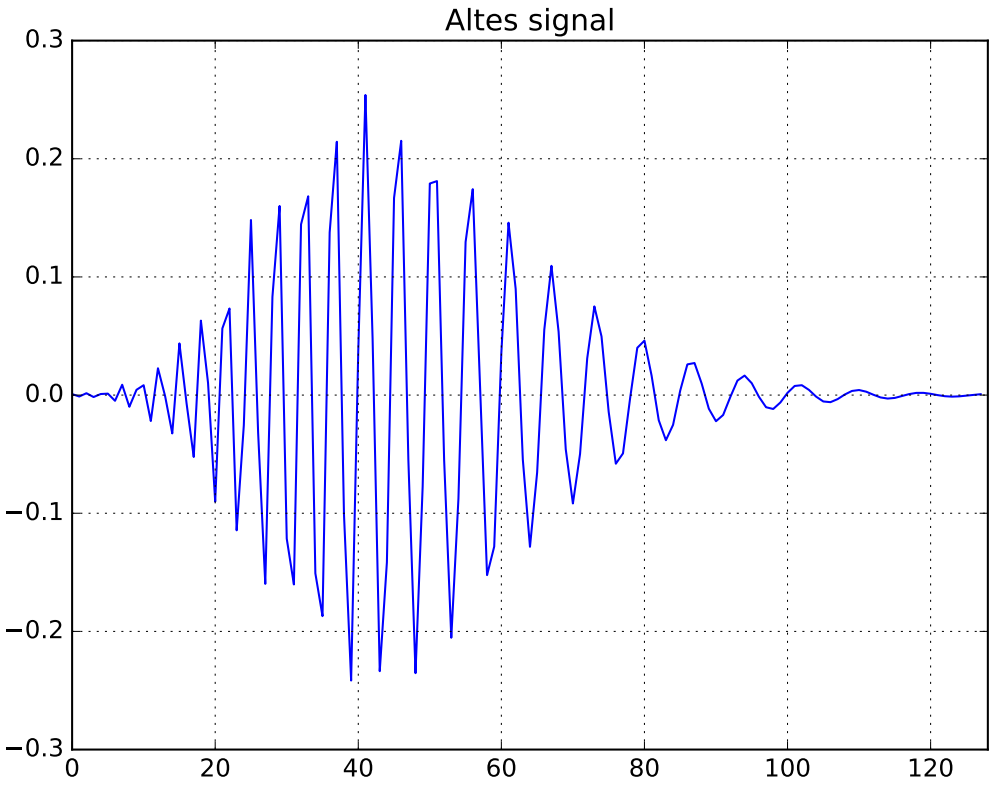
- **n_points** (*int*) – Number of points in time.
- **fmin** (*float*) – Lower frequency bound.
- **fmax** (*float*) – Higher frequency bound.
- **alpha** (*float*) – Attenuation factor of the envelope.

Returns Time vector containing the Altes signal samples.

Return type numpy.ndarray

Examples

```
>>> x = altes(128, 0.1, 0.45)
>>> plot(x)
```



tftb.generators.**atoms**(*n_points*, *coordinates*)

Compute linear combination of elementary Gaussian atoms.

Parameters

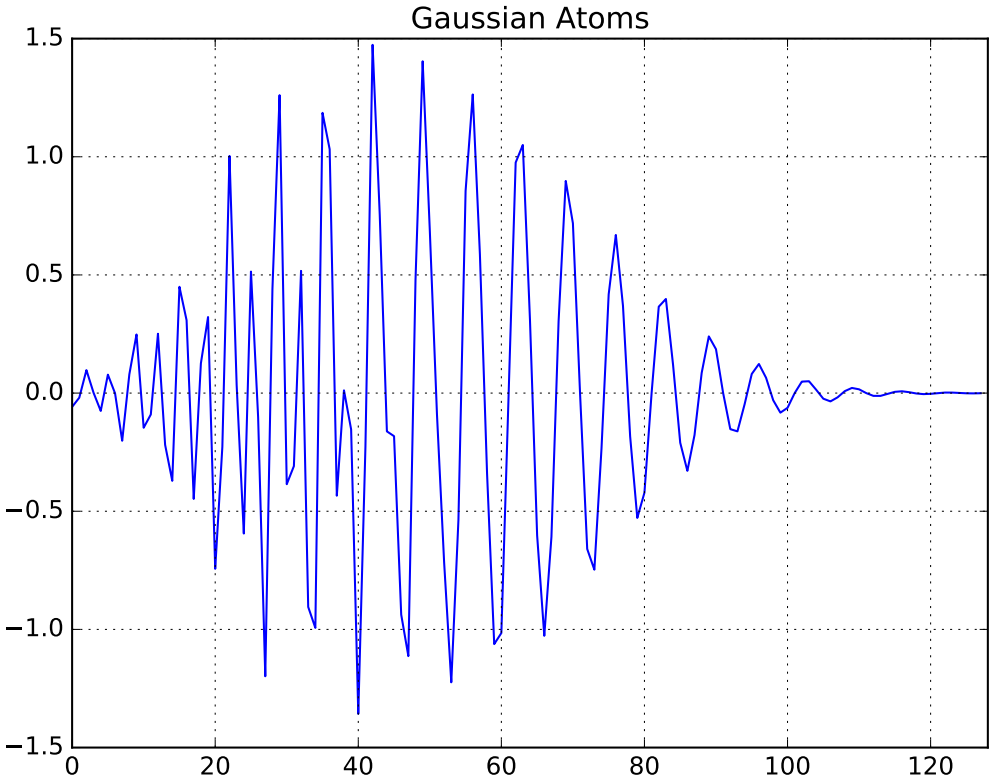
- **n_points** (*int*) – Number of points in a signal
- **coordinates** (*array-like*) – matrix of time-frequency centers

Returns signal

Return type array-like

Examples

```
>>> import numpy as np
>>> coordinates = np.array([[32.0, 0.3, 32.0, 1.0], [0.15, 0.15, 48.0, 1.22]])
>>> sig = atoms(128, coordinates)
>>> plot(real(sig))
```



tftb.processing package

Submodules

tftb.processing.affine module

Bilinear Time-Frequency Processing in the Affine Class.

class tftb.processing.affine.**AffineDistribution**(*signal*, *fmin=None*, *fmax=None*, ***kwargs*)

Bases: *tftb.processing.base.BaseTFRepresentation*

__init__(*signal*, *fmin=None*, *fmax=None*, ***kwargs*)

isaffine = True

plot(*kind='contour'*, *show_tf=True*, *threshold=0.05*, ***kwargs*)

run()

class tftb.processing.affine.**BertrandDistribution**(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

Bases: *tftb.processing.affine.AffineDistribution*

__init__(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

name = 'bertrand'

run()

class tftb.processing.affine.**DFlandrinDistribution**(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

Bases: *tftb.processing.affine.AffineDistribution*

__init__(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

name = 'd-flandrin'

run()

class tftb.processing.affine.**Scalogram**(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, *waveparams=None*, ***kwargs*)

Bases: *tftb.processing.affine.AffineDistribution*

Morlet Scalogram.

__init__(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, *waveparams=None*, ***kwargs*)

isaffine = False

name = 'scalogram'

run()

class tftb.processing.affine.**UnterbergerDistribution**(*signal*, *form='A'*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

Bases: *tftb.processing.affine.AffineDistribution*

__init__(*signal*, *form='A'*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

name = 'unterberger'

run()

tftb.processing.affine.**lambdak**(*u*, *k*)

tftb.processing.affine.**smoothed_pseudo_wigner**(*signal*, *timestamps=None*, *K='bertrand'*, *nh0=None*, *ng0=0*, *fmin=None*, *fmax=None*, *n_voices=None*)

Parameters

- **signal** –
- **timestamps** –
- **K** –
- **nh0** –
- **ng0** –
- **fmin** –
- **fmax** –
- **n_voices** –

Returns

Return type

tftb.processing.affine.**umaxdfla_solve**(*ratio*)

tftb.processing.ambiguity module

Ambiguity functions.

tftb.processing.ambiguity.**narrow_band**(*signal*, *lag=None*, *n_fbins=None*)
Narrow band ambiguity function.

Parameters

- **signal** (*array-like*) – Signal to be analyzed.
- **lag** (*array-like*) – vector of lag values.
- **n_fbins** (*int*) – number of frequency bins

Returns Doppler lag representation

Return type array-like

tftb.processing.ambiguity.**wide_band**(*signal*, *fmin=None*, *fmax=None*, *N=None*)

tftb.processing.base module

Base time-frequency representation class.

class tftb.processing.base.**BaseTFRepresentation**(*signal*, ***kwargs*)
Bases: object

__init__(*signal*, ***kwargs*)
Create a base time-frequency representation object.

Parameters

- **signal** (*array-like*) – Signal to be analyzed.
- ****kwargs** – Other arguments required for performing the analysis.

Returns BaseTFRepresentation object

Return type

isaffine = False

plot(*ax=None*, *kind='cmap'*, *show=True*, *default_annotation=True*, *show_tf=False*, *scale='linear'*, *threshold=0.05*, ***kwargs*)
Visualize the time frequency representation.

Parameters

- **ax** (*matplotlib.axes.Axes object*) – Axes object to draw the plot on.
- **kind** (*str*) – One of "cmap" (default), "contour".
- **show** (*bool*) – Whether to call plt.show().
- **default_annotation** (*bool*) – Whether to make default annotations for the plot. Default annotations consist of setting the X and Y axis labels to "Time" and "Normalized Frequency" respectively, and setting the title to the name of the particular time-frequency distribution.
- **show_tf** – Whether to show the signal and it's spectrum alongwith the plot. In this is True, the ax argument is ignored.
- ****kwargs** – Parameters to be passed to the plotting function.

Returns None

Return type None

tftb.processing.cohen module

Bilinear Time-Frequency Processing in the Cohen's Class.

class tftb.processing.cohen.**MargenauHillDistribution**(*signal*, ***kwargs*)
Bases: *tftb.processing.base.BaseTFRepresentation*

name = 'margenau-hill'

plot(*kind='cmap'*, *threshold=0.05*, *sqmod=True*, ***kwargs*)

run()

class tftb.processing.cohen.**PageRepresentation**(*signal*, ***kwargs*)
Bases: *tftb.processing.base.BaseTFRepresentation*

name = 'page representation'

plot(*kind='cmap'*, *threshold=0.05*, *sqmod=True*, ***kwargs*)

run()

class tftb.processing.cohen.**PseudoMargenauHillDistribution**(*signal*, ***kwargs*)
Bases: *tftb.processing.cohen.MargenauHillDistribution*

name = 'pseudo margenau-hill'

run()

class tftb.processing.cohen.**PseudoPageRepresentation**(*signal*, ***kwargs*)
Bases: *tftb.processing.cohen.PageRepresentation*

name = 'pseudo page'

run()

class tftb.processing.cohen.**PseudoWignerVilleDistribution**(*signal*, ***kwargs*)
Bases: *tftb.processing.cohen.WignerVilleDistribution*

name = 'pseudo winger-ville'

plot(***kwargs*)

run()

class tftb.processing.cohen.**Spectrogram**(*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)
Bases: *tftb.processing.linear.ShortTimeFourierTransform*

name = 'spectrogram'

plot(*kind='cmap'*, ***kwargs*)

run()

class tftb.processing.cohen.**WignerVilleDistribution**(*signal*, ***kwargs*)
Bases: *tftb.processing.base.BaseTFRepresentation*

name = 'wigner-ville'

plot(*kind='cmap'*, *threshold=0.05*, *sqmod=False*, ***kwargs*)

run()

tftb.processing.cohen.**smoothed_pseudo_wigner_ville**(*signal*, *timestamps=None*, *freq_bins=None*, *twindow=None*, *fwindow=None*)

Smoothed Pseudo Wigner-Ville time-frequency distribution. :param signal: signal to be analyzed :param timestamps: time instants of the signal :param freq_bins: number of frequency bins :param twindow: time smoothing window :param fwindow: frequency smoothing window :type signal: array-like :type timestamps: array-like :type freq_bins: int :type twindow: array-like :type fwindow: array-like :return: Smoothed pseudo Wigner Ville distribution :rtype: array-like

tftb.processing.freq_domain module

tftb.processing.freq_domain.**group_delay**(*x*, *fnorm=None*)

Compute the group delay of a signal at normalized frequencies.

Parameters

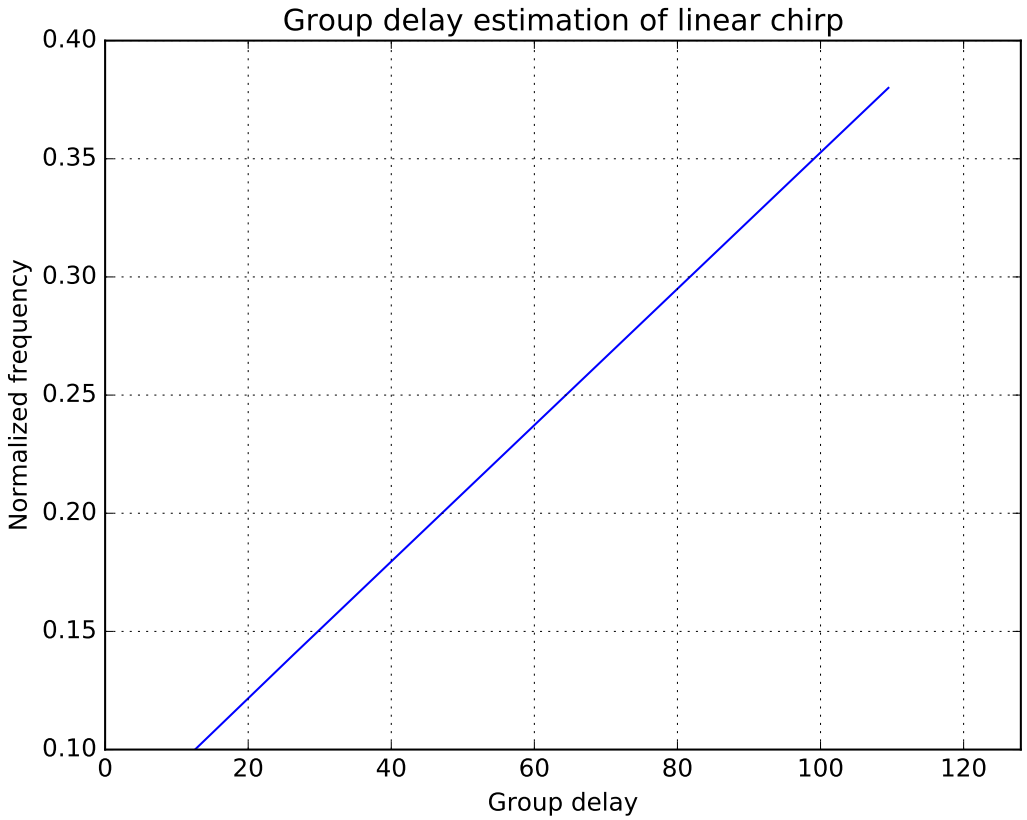
- **x** (*numpy.ndarray*) – time domain signal
- **fnorm** (*float*) – normalized frequency at which to calculate the group delay.

Returns group delay

Return type numpy.ndarray

Example

```
>>> import numpy as np
>>> from tftb.generators import amgauss, fmlin
>>> x = amgauss(128, 64.0, 30) * fmlin(128, 0.1, 0.4)[0]
>>> fnorm = np.arange(0.1, 0.38, step=0.04)
>>> gd = group_delay(x, fnorm)
>>> plot(gd, fnorm)
```



tftb.processing.freq_domain.**inst_freq**(*x*, *t=None*, *L=1*)

Compute the instantaneous frequency of an analytic signal at specific time instants using the trapezoidal integration rule.

Parameters

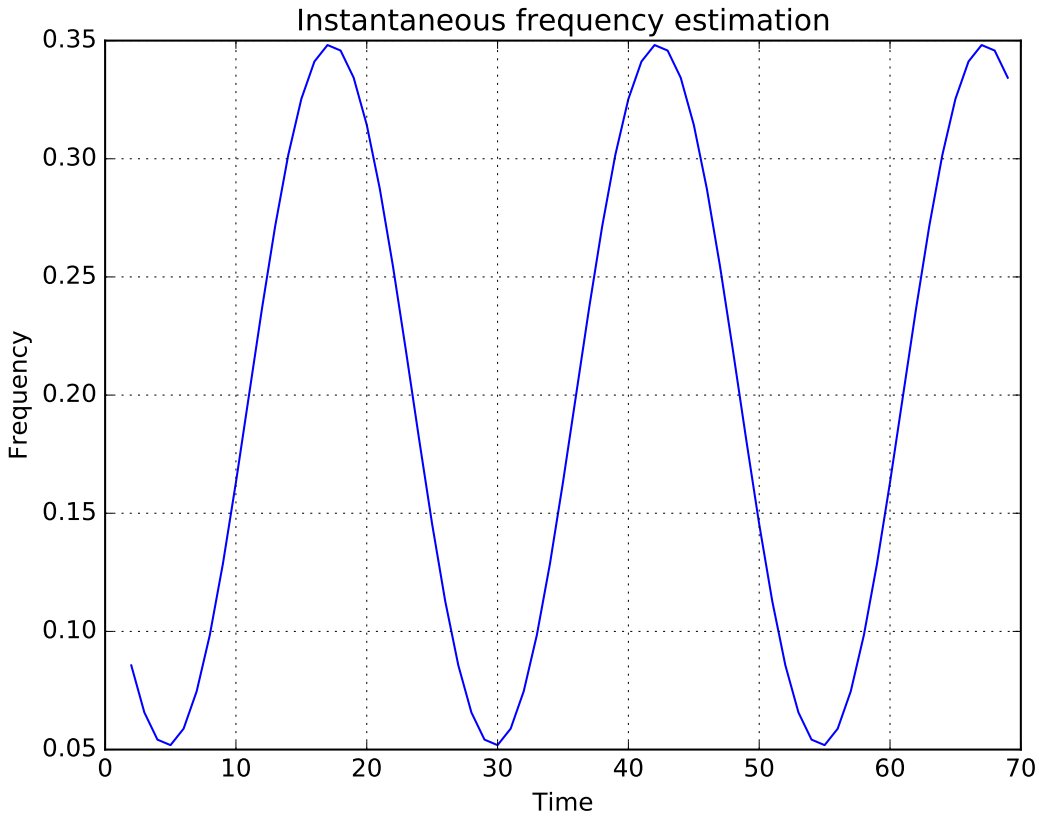
- **x** (*numpy.ndarray*) – The input analytic signal
- **t** (*numpy.ndarray*) – The time instants at which to calculate the instantaneous frequencies.
- **L** (*int*) – Non default values are currently not supported. If L is 1, the normalized instantaneous frequency is computed. If L > 1, the maximum likelihood estimate of the instantaneous frequency of the deterministic part of the signal.

Returns instantaneous frequencies of the input signal.

Return type numpy.ndarray

Example

```
>>> from tftb.generators import fmsin
>>> x = fmsin(70, 0.05, 0.35, 25)[0]
>>> instf, timestamps = inst_freq(x)
>>> plot(timestamps, instf)
```



tftb.processing.freq_domain.**locfreq**(*signal*)

Compute the frequency localization characteristics.

Parameters **sig** (*numpy.ndarray*) – input signal

Returns average normalized frequency center, frequency spreading

Return type tuple

Example

```
>>> from tftb.generators import amgauss
>>> z = amgauss(160, 80, 50)
>>> fm, B = locfreq(z)
>>> print("%.4g" % fm)
-9.183e-14
>>> print("%.4g" % B)
0.02
```

tftb.processing.linear module

Linear Time Frequency Processing.

class tftb.processing.linear.**ShortTimeFourierTransform**(*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)

Bases: *tftb.processing.base.BaseTFRepresentation*

Short time Fourier transform.

__init__ (*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)
Create a ShortTimeFourierTransform object.

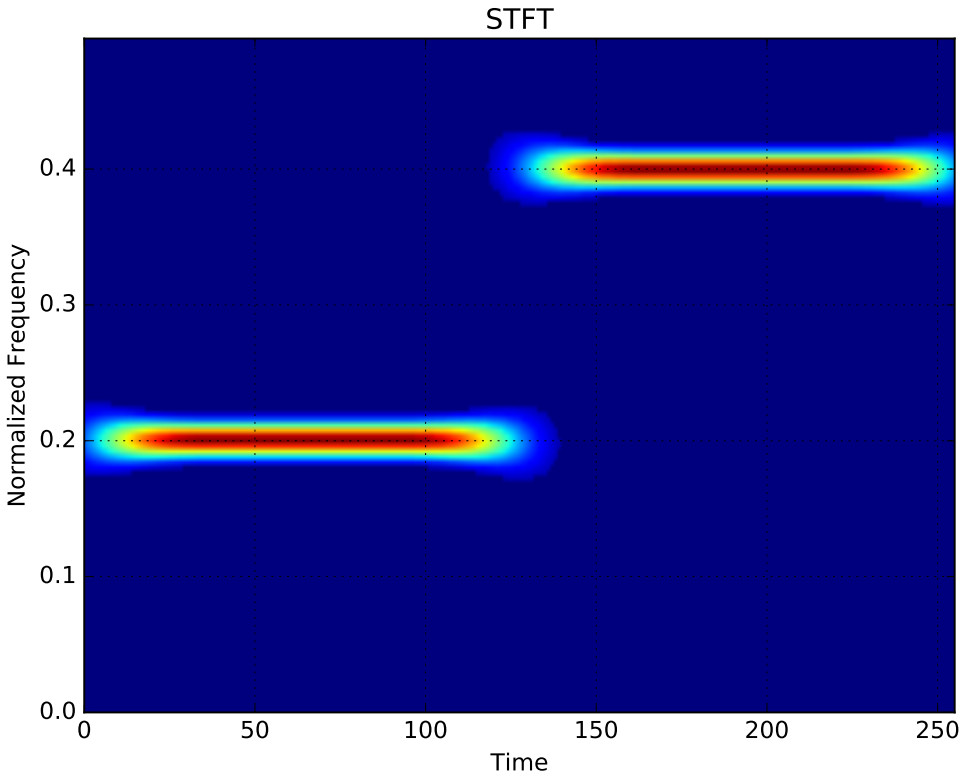
Parameters

- **signal** (*array-like*) – Signal to be analyzed.
- **timestamps** (*array-like*) – Time instants of the signal (default: `np.arange(len(signal))`)
- **n_fbins** (*int*) – Number of frequency bins (default: `len(signal)`)
- **fwindow** (*array-like*) – Frequency smoothing window (default: Hamming window of length `len(signal) / 4`)

Returns ShortTimeFourierTransform object

Example

```
>>> from tftb.generators import fmconst
>>> sig = np.r_[fmconst(128, 0.2)[0], fmconst(128, 0.4)[0]]
>>> stft = ShortTimeFourierTransform(sig)
>>> tfr, t, f = stft.run()
>>> stft.plot()
```



name = 'stft'

plot (*ax=None*, *kind='cmap'*, *sqmod=True*, *threshold=0.05*, ***kwargs*)
Display the spectrogram of an STFT.

Parameters

- **ax** (*matplotlib.axes.Axes object*) – axes object to draw the plot on. If None(default), one will be created.
- **kind** (*str*) – Choice of visualization type, either "cmap"(default) or "contour".
- **sqmod** (*bool*) – Whether to take squared modulus of TFR before plotting. (Default: True)
- **threshold** (*float*) – Percentage of the maximum value of the TFR, below which all values are set to zero before plotting.
- ****kwargs** – parameters passed to the plotting function.

Returns None

Return type None

run()

Compute the STFT according to:

$$X[m, w] = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-j\omega n}$$

Where w is a Hamming window.

tftb.processing.linear.**gabor**(*signal*, *n_coeff=None*, *q_oversample=None*, *window=None*)

Compute the Gabor representation of a signal.

Parameters

- **signal** (*array-like*) – Singal to be analyzed.
- **n_coeff** (*integer*) – number of Gabor coefficients in time.
- **q_oversample** (*int*) – Degree of oversampling
- **window** (*array-like*) – Synthesis window

Returns Tuple of Gabor coefficients, biorthogonal window associated with the synthesis window.

Return type tuple

tftb.processing.plotifl module

tftb.processing.plotifl.**plotifl**(*time_instants*, *iflaws*, *signal=None*, ***kwargs*)

Plot normalized instantaneous frequency laws.

Parameters

- **time_instants** (*array-like*) – timestamps of the signal
- **iflaws** (*array-like*) – instantaneous freqency law(s) of the signal.
- **signal** (*array-like*) – if provided, display it.

Returns None

tftb.processing.postprocessing module

Postprocessing functions.

tftb.processing.postprocessing.**friedman_density**(*tfr*, *re_mat*, *timestamps=None*)

Parameters

- **tfr** –
- **re_mat** –
- **timestamps** –

Returns

Return type

tftb.processing.postprocessing.**hough_transform**(*image*, *m=None*, *n=None*)

Parameters

- **image** –
- **m** –
- **n** –

Returns

Return type

tftb.processing.postprocessing.**ideal_tfr**(*iflaws*, *timestamps=None*, *n_fbins=None*)

Parameters

- **iflaws** –
- **timestamps** –
- **n_fbins** –

Returns

Return type

tftb.processing.postprocessing.**renyi_information**(*tfr*, *timestamps=None*, *freq=None*, *alpha=3.0*)

Parameters

- **tfr** –
- **timestamps** –
- **freq** –
- **alpha** –

Returns

Return type

tftb.processing.postprocessing.**ridges**(*tfr*, *re_mat*, *timestamps=None*, *method='rsp'*)

Parameters

- **tfr** –
- **re_mat** –
- **timestamps** –
- **method** –

Returns

Return type

tftb.processing.reassigned module

Reassigned TF processing.

tftb.processing.reassigned.**morlet_scalogram**(*signal*, *timestamps=None*, *n_fbins=None*, *tbp=0.25*)

param signal

param timestamps

param n_fbins

param tbp

type signal

type timestamps

type n_fbins

type tbp

Returns

Return type

tftb.processing.reassigned.**pseudo_margenau_hill**(*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)

param signal

param timestamps

param n_fbins

param fwindow

type signal

type timestamps

type n_fbins

type fwindow

Returns

Return type

tftb.processing.reassigned.**pseudo_page**(*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)

param signal

param timestamps

param n_fbins

param fwindow

type signal

type timestamps

type n_fbins

type fwindow

Returns

Return type

tftb.processing.reassigned.**pseudo_wigner_ville**(*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)

param signal

param timestamps

param n_fbins

param fwindow

type signal

type timestamps

type n_fbins

type fwindow

Returns

Return type

tftb.processing.reassigned.**smoothed_pseudo_wigner_ville**(*signal*, *timestamps=None*, *n_fbins=None*, *twindow=None*, *fwindow=None*)

param signal

param timestamps

param n_fbins

param twindow

param fwindow

type signal

type timestamps

type n_fbins

type twindow

type fwindow

Returns

Return type

tftb.processing.reassigned.**spectrogram**(*signal*, *time_samples=None*, *n_fbins=None*, *window=None*)

Compute the spectrogram and reassigned spectrogram.

Parameters

- **signal** (*array-like*) – signal to be analzsed
- **time_samples** (*array-like*) – time instants (default: np.arange(len(signal)))
- **n_fbins** (*int*) – number of frequency bins (default: len(signal))

- **window** (*array-like*) – frequency smoothing window (default: Hamming with size=len(signal)/4)

Returns spectrogram, reassigned specstrogram and matrix of reassignment

vectors :rtype: tuple(array-like)

tftb.processing.time_domain module

tftb.processing.time_domain.**loctime**(*sig*)

Compute the time localization characteristics.

Parameters **sig** (*numpy.ndarray*) – input signal

Returns Average time center and time spreading

Return type tuple

Example

```
>>> from tftb.generators import amgauss
>>> x = amgauss(160, 80.0, 50.0)
>>> tm, T = loctime(x)
>>> print("%.2f" % tm)
79.00
>>> print("%.2f" % T)
50.00
```

tftb.processing.utils module

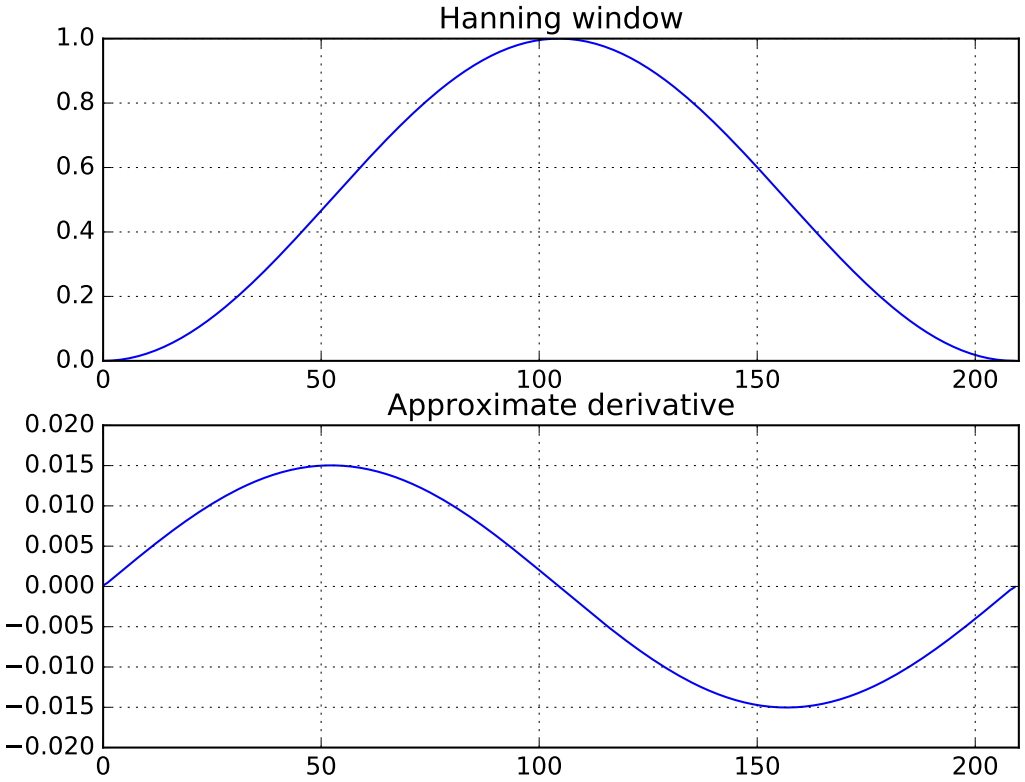
Miscellaneous processing utilities.

tftb.processing.utils.**derive_window**(*window*)

Calculate derivative of a window function.

Parameters **window** – Window function to be differentiated. This is expected to be

a standard window function with an odd length. :type window: array-like :return: Derivative of the input window :rtype: array-like :Example: >>> from scipy.signal import hanning >>> import matplotlib.pyplot as plt #doctest: +SKIP >>> window = hanning(210) >>> derivation = derive_window(window) >>> plt.subplot(211), plt.plot(window) #doctest: +SKIP >>> plt.subplot(212), plt.plot(derivation) #doctest: +SKIP



tftb.processing.utils.**get_spectrum**(*signal*)

tftb.processing.utils.**integrate_2d**(*mat*, *x=None*, *y=None*)

param mat

param x

param y

type mat

type x

type y

Returns

Return type

Example

```
>>> from __future__ import print_function
>>> from tftb.generators import altes
>>> from tftb.processing import Scalogram
>>> x = altes(256, 0.1, 0.45, 10000)
>>> tfr, t, f, _ = Scalogram(x).run()
>>> print("%.3f" % integrate_2d(tfr, t, f))
2.000
```

Module contents

tftb.processing.**loctime**(*sig*)

Compute the time localization characteristics.

Parameters **sig** (*numpy.ndarray*) – input signal

Returns Average time center and time spreading

Return type tuple

Example

```
>>> from tftb.generators import amgauss
>>> x = amgauss(160, 80.0, 50.0)
>>> tm, T = loctime(x)
>>> print("%.2f" % tm)
79.00
>>> print("%.2f" % T)
50.00
```

tftb.processing.**locfreq**(*signal*)

Compute the frequency localization characteristics.

Parameters **sig** (*numpy.ndarray*) – input signal

Returns average normalized frequency center, frequency spreading

Return type tuple

Example

```
>>> from tftb.generators import amgauss
>>> z = amgauss(160, 80, 50)
>>> fm, B = locfreq(z)
>>> print("%.4g" % fm)
-9.183e-14
>>> print("%.4g" % B)
0.02
```

tftb.processing.**inst_freq**(*x*, *t=None*, *L=1*)

Compute the instantaneous frequency of an analytic signal at specific time instants using the trapezoidal integration rule.

Parameters

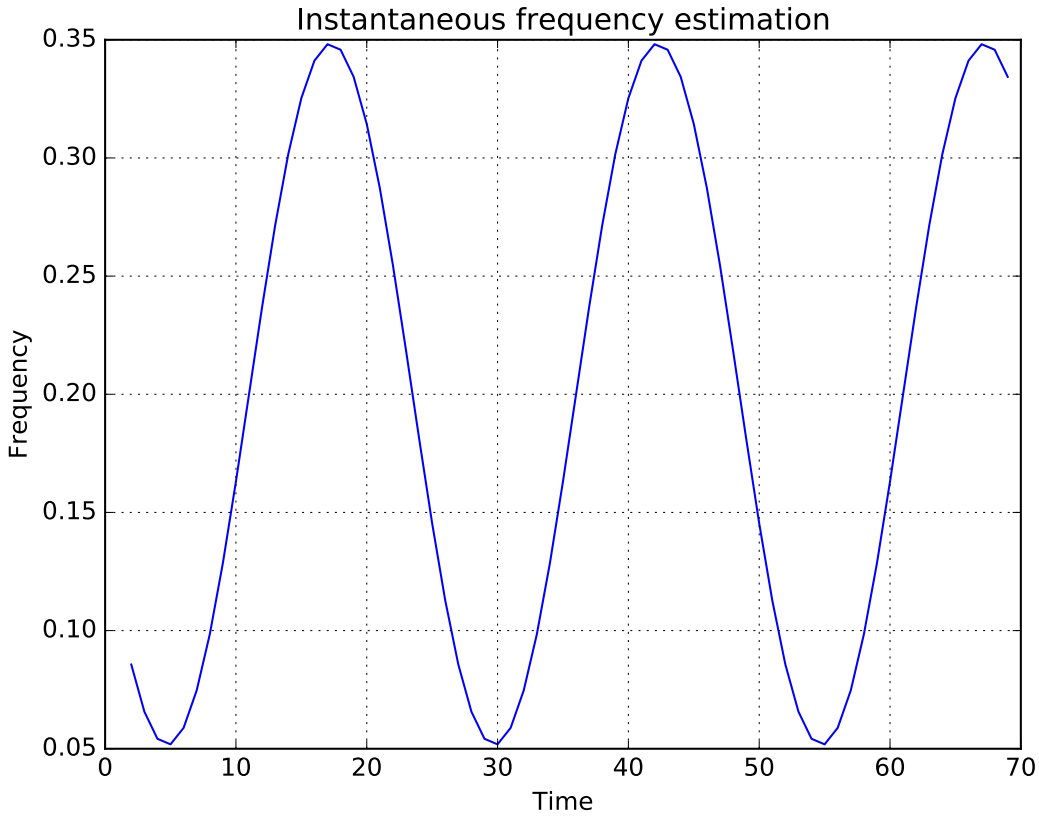
- **x** (*numpy.ndarray*) – The input analytic signal
- **t** (*numpy.ndarray*) – The time instants at which to calculate the instantaneous frequencies.
- **L** (*int*) – Non default values are currently not supported. If L is 1, the normalized instantaneous frequency is computed. If L > 1, the maximum likelihood estimate of the instantaneous frequency of the deterministic part of the signal.

Returns instantaneous frequencies of the input signal.

Return type numpy.ndarray

Example

```
>>> from tftb.generators import fmsin
>>> x = fmsin(70, 0.05, 0.35, 25)[0]
>>> instf, timestamps = inst_freq(x)
>>> plot(timestamps, instf)
```



tftb.processing.**group_delay**(*x*, *fnorm=None*)

Compute the group delay of a signal at normalized frequencies.

Parameters

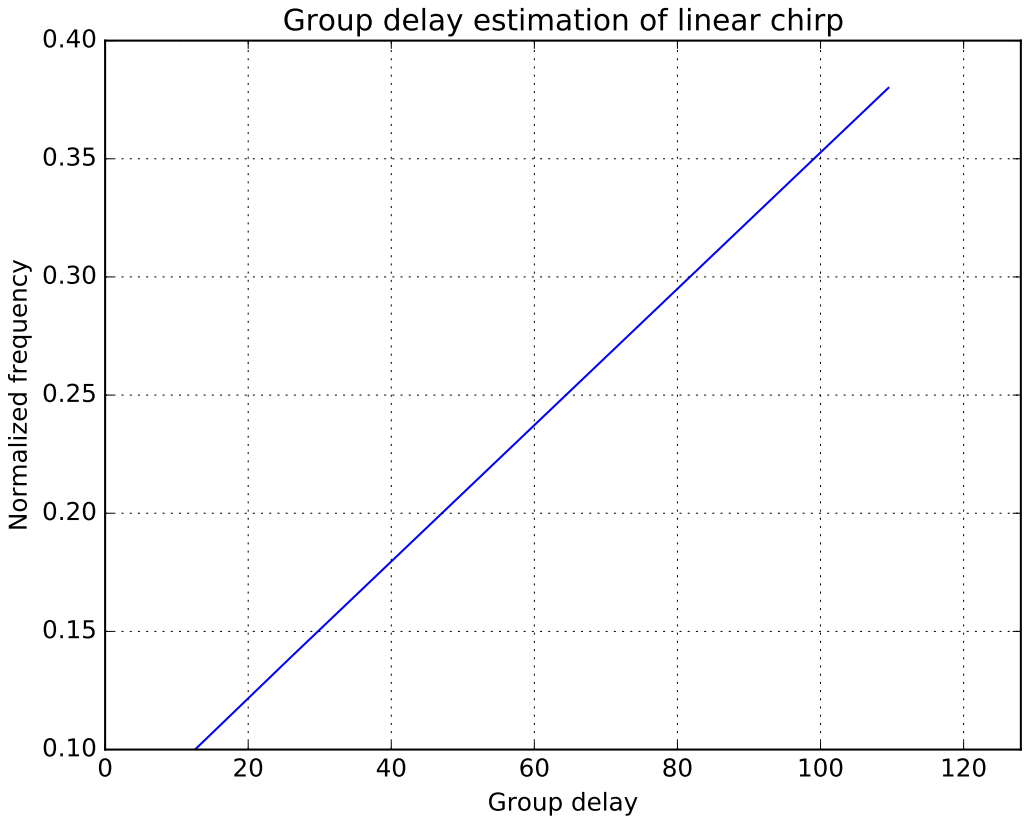
- **x** (*numpy.ndarray*) – time domain signal
- **fnorm** (*float*) – normalized frequency at which to calculate the group delay.

Returns group delay

Return type numpy.ndarray

Example

```
>>> import numpy as np
>>> from tftb.generators import amgauss, fmlin
>>> x = amgauss(128, 64.0, 30) * fmlin(128, 0.1, 0.4)[0]
>>> fnorm = np.arange(0.1, 0.38, step=0.04)
>>> gd = group_delay(x, fnorm)
>>> plot(gd, fnorm)
```



tftb.processing.**plotifl**(*time_instants*, *iflaws*, *signal=None*, ***kwargs*)
Plot normalized instantaneous frequency laws.

Parameters

- **time_instants** (*array-like*) – timestamps of the signal
- **iflaws** (*array-like*) – instantaneous freqency law(s) of the signal.
- **signal** (*array-like*) – if provided, display it.

Returns None

class tftb.processing.**WignerVilleDistribution**(*signal*, ***kwargs*)
Bases: *tftb.processing.base.BaseTFRepresentation*

name = 'wigner-ville'

plot(*kind='cmap'*, *threshold=0.05*, *sqmod=False*, ***kwargs*)

run()

class tftb.processing.**PseudoWignerVilleDistribution**(*signal*, ***kwargs*)
Bases: *tftb.processing.cohen.WignerVilleDistribution*

name = 'pseudo winger-ville'

plot(***kwargs*)

run()

tftb.processing.**smoothed_pseudo_wigner_ville**(*signal*, *timestamps=None*, *freq_bins=None*, *twindow=None*, *fwindow=None*)
Smoothed Pseudo Wigner-Ville time-frequency distribution. :param signal: signal to be analyzed :param timestamps: time instants of the signal :param freq_bins: number of frequency bins :param twindow: time smoothing window :param fwindow: frequency smoothing window :type signal: array-like :type timestamps: array-like :type freq_bins: int :type twindow: array-like :type fwindow: array-like :return: Smoothed pseudo Wigner Ville distribution :rtype: array-like

class tftb.processing.**MargenauHillDistribution**(*signal*, ***kwargs*)
Bases: *tftb.processing.base.BaseTFRepresentation*

name = 'margenau-hill'

plot(*kind='cmap'*, *threshold=0.05*, *sqmod=True*, ***kwargs*)

run()

class tftb.processing.**Spectrogram**(*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)
Bases: *tftb.processing.linear.ShortTimeFourierTransform*

name = 'spectrogram'

plot(*kind='cmap'*, ***kwargs*)

run()

tftb.processing.**reassigned_spectrogram**(*signal*, *time_samples=None*, *n_fbins=None*, *window=None*)
Compute the spectrogram and reassigned spectrogram.

Parameters

- **signal** (*array-like*) – signal to be analzsed
- **time_samples** (*array-like*) – time instants (default: np.arange(len(signal)))
- **n_fbins** (*int*) – number of frequency bins (default: len(signal))
- **window** (*array-like*) – frequency smoothing window (default: Hamming with size=len(signal)/4)

Returns spectrogram, reassigned specstrogram and matrix of reassignment

vectors :rtype: tuple(array-like)

tftb.processing.**reassigned_smoothed_pseudo_wigner_ville**(*signal*, *timestamps=None*, *n_fbins=None*, *twindow=None*, *fwindow=None*)
smoothed_pseudo_wigner_ville

param signal

param timestamps

param n_fbins

param twindow

param fwindow

type signal

type timestamps

type n_fbins

type twindow

type fwindow

Returns

Return type

tftb.processing.**ideal_tfr**(*iflaws*, *timestamps=None*, *n_fbins=None*)

Parameters

- **iflaws** –
- **timestamps** –
- **n_fbins** –

Returns

Return type

tftb.processing.**renyi_information**(*tfr*, *timestamps=None*, *freq=None*, *alpha=3.0*)

Parameters

- **tfr** –
- **timestamps** –
- **freq** –
- **alpha** –

Returns

Return type

class tftb.processing.**Scalogram**(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, *waveparams=None*, ***kwargs*)

Bases: *tftb.processing.affine.AffineDistribution*

Morlet Scalogram.

__init__(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, *waveparams=None*, ***kwargs*)

isaffine = False

name = 'scalogram'

run()

class tftb.processing.**BertrandDistribution**(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

Bases: *tftb.processing.affine.AffineDistribution*

__init__(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

name = 'bertrand'

run()

class tftb.processing.**DFlandrinDistribution**(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

Bases: *tftb.processing.affine.AffineDistribution*

__init__(*signal*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

name = 'd-flandrin'

run()

class tftb.processing.**UnterbergerDistribution**(*signal*, *form='A'*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

Bases: *tftb.processing.affine.AffineDistribution*

__init__(*signal*, *form='A'*, *fmin=None*, *fmax=None*, *n_voices=None*, ***kwargs*)

name = 'unterberger'

run()

class tftb.processing.**ShortTimeFourierTransform**(*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)

Bases: *tftb.processing.base.BaseTFRepresentation*

Short time Fourier transform.

__init__(*signal*, *timestamps=None*, *n_fbins=None*, *fwindow=None*)

Create a ShortTimeFourierTransform object.

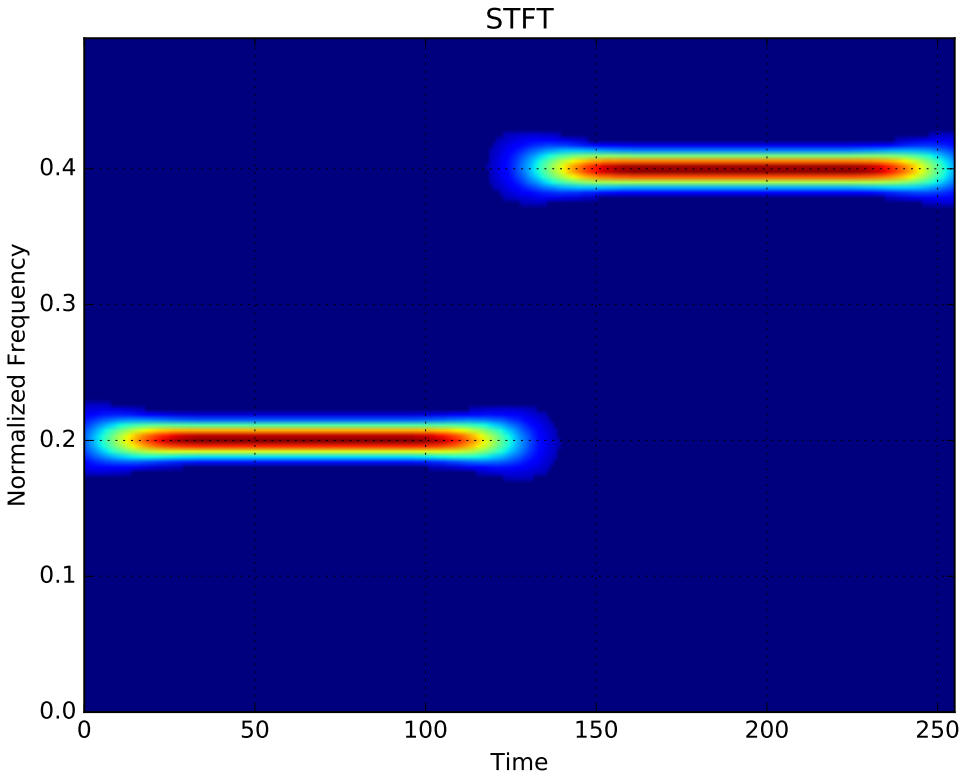
Parameters

- **signal** (*array-like*) – Signal to be analyzed.
- **timestamps** (*array-like*) – Time instants of the signal (default: `np.arange(len(signal))`)
- **n_fbins** (*int*) – Number of frequency bins (default: `len(signal)`)
- **fwindow** (*array-like*) – Frequency smoothing window (default: Hamming window of length `len(signal) / 4`)

Returns ShortTimeFourierTransform object

Example

```
>>> from tftb.generators import fmconst
>>> sig = np.r_[fmconst(128, 0.2)[0], fmconst(128, 0.4)[0]]
>>> stft = ShortTimeFourierTransform(sig)
>>> tfr, t, f = stft.run()
>>> stft.plot()
```



name = 'stft'

plot(*ax=None*, *kind='cmap'*, *sqmod=True*, *threshold=0.05*, ***kwargs*)

Display the spectrogram of an STFT.

Parameters

- **ax** (*matplotlib.axes.Axes object*) – axes object to draw the plot on. If None(default), one will be created.
- **kind** (*str*) – Choice of visualization type, either "cmap"(default) or "contour".
- **sqmod** (*bool*) – Whether to take squared modulus of TFR before plotting. (Default: True)
- **threshold** (*float*) – Percentage of the maximum value of the TFR, below which all values are set to zero before plotting.
- ****kwargs** – parameters passed to the plotting function.

Returns None

Return type None

run()

Compute the STFT according to:

$$X[m, w] = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-j\omega n}$$

Where w is a Hamming window.

tftb.tests package

Submodules

tftb.tests.base module

tftb.tests.test_utils module

Tests for tftb.utils

class `tftb.tests.test_utils.`**`TestUtils`**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

classmethod `setUpClass`()

`test_divider`()
Test the divider function.

`test_is_linear`()
Test the is_linear function.

`test_modulo`()
Test the modulo function.

`test_nearest_odd`()
Test the nearest_odd function.

`test_nextpow2`()
Test the nextpow2 function.

Module contents

4.1.2 Submodules

4.1.3 tftb.utils module

Miscellaneous utilities.

`tftb.utils.`**`divider`**(*N*)
Compute two factors of N such that they are as close as possible to sqrt(N).

Parameters **`N`** (*int*) – Number to be divided.

Returns A tuple of two integers such that their product is *N* and they are the closest possible to $\sqrt{(N)}$

Return type tuple(int)

Example

```
>>> from __future__ import print_function
>>> print(divider(256))
(16.0, 16.0)
>>> print(divider(10))
(2.0, 5.0)
>>> print(divider(101))
(1.0, 101.0)
```

tftb.utils.**is_linear**(*x*, *decimals=5*)
Check if an array is linear.

Parameters

- **x** (*numpy.ndarray*) – Array to be checked for linearity.
- **decimals** (*int*) – decimal places upto which the derivative of the array should be rounded off (default=5)

Returns If the array is linear

Return type boolean

Example

```
>>> import numpy as np
>>> x = np.linspace(0, 2 * np.pi, 100)
>>> is_linear(x)
True
>>> is_linear(np.sin(x))
False
```

tftb.utils.**izak**(*x*)
Inverse Zak transform.

tftb.utils.**modulo**(*x*, *N*)
Compute the congruence of each element of x modulo N.

Returns array-like

Example

```
>>> from __future__ import print_function
>>> print(modulo(range(1, 11), 2))
[1 2 1 2 1 2 1 2 1 2]
```

tftb.utils.**nearest_odd**(*N*)
Get the nearest odd number for each value of N.

Parameters **N** – int / sequence of ints

Returns int / sequence of ints

Example

```
>>> from __future__ import print_function
>>> print(nearest_odd(range(1, 11)))
[  1.   3.   3.   5.   5.   7.   7.   9.   9.  11.]
>>> nearest_odd(0)
1
>>> nearest_odd(3)
3.0
```

tftb.utils.**nextpow2**(*n*)
Compute the integer exponent of the next higher power of 2.

Parameters **n** (*int, np.ndarray*) – Number whose next higest power of 2 needs to be computed.

Return type int, np.ndarray

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> x = np.arange(1, 9)
>>> print(nextpow2(x))
[ 0.  1.  2.  2.  3.  3.  3.  3.]
```

4.1.4 Module contents

4.2 tftb

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

D

F

G

H

I

K

L

M

N

P

R

S

T

U

W