pyhht Documentation

Release 0.0.1

Jaidev Deshpande

Aug 23, 2018

Contents

Contents:

CHAPTER 1

PyHHT Tutorials

The Hilbert Huang transform (HHT) is a time series analysis technique that is designed to handle nonlinear and nonstationary time series data. PyHHT is a Python module based on NumPy and SciPy which implements the HHT. These tutorials introduce HHT, the common vocabulary associated with it and the usage of the PyHHT module itself to analyze time series data.

This series of tutorials goes through the philosophy of the Hilbert Huang transform in detail. The first two tutorials lay the groundwork for the HHT, providing the motivation first for the Hilbert spectral analysis and then for the empirical mode decomposition algorithm. The third tutorial is an introduction to the PyHHT module. You may skip the first two sections if you are comfortable with the theory of HHT and if you want to start coding Python applications with PyHHT.

Contents:

1.1 Limitations of the Fourier Transform: Need For a Data Driven Approach

Methods based on the Fourier transform are almost synonymous with frequency domain processing of signals (funnily, I once had a classmate who thought "Fourier" was French for frequency). There is no doubt about how incredibly powerful Fourier analysis can be. However, its popularity and effectiveness have a downside. It has led to a very specific and limited view of frequency in the context of signal processing. Simply put, frequencies, in the context of Fourier methods, are just a collection of the individual frequencies of periodic signals that a given signal is composed of. The purpose of these tutorials is to demonstrate how restrictive this interpretation of frequency can be in some cases, and to lay the groundwork for complementary methods, like the Hilbert spectral analysis.

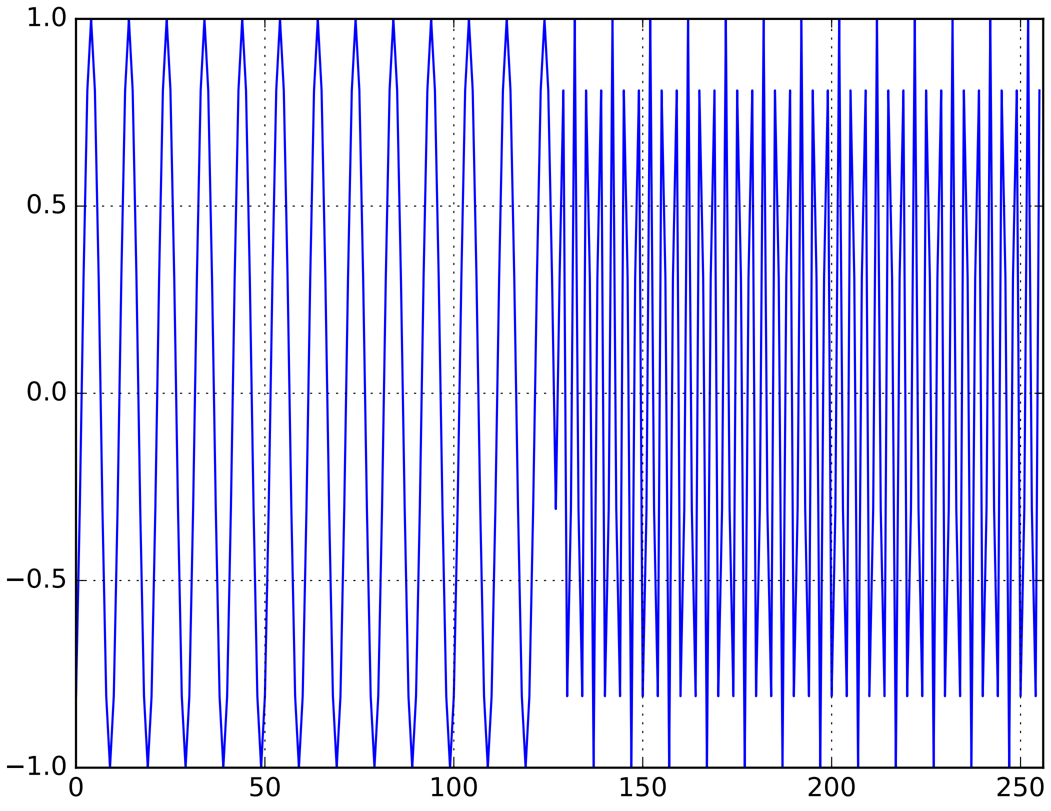
This is not to say that Hilbert spectral analysis can in any way replace Fourier, but that it provides an alternative interpretation of frequency, and an alternative view of nonlinear and nonstationary phenomena.

1.1.1 The Problem

Note: To run the code snippets on this page, you will need the PyTFTB module, which can be found here.

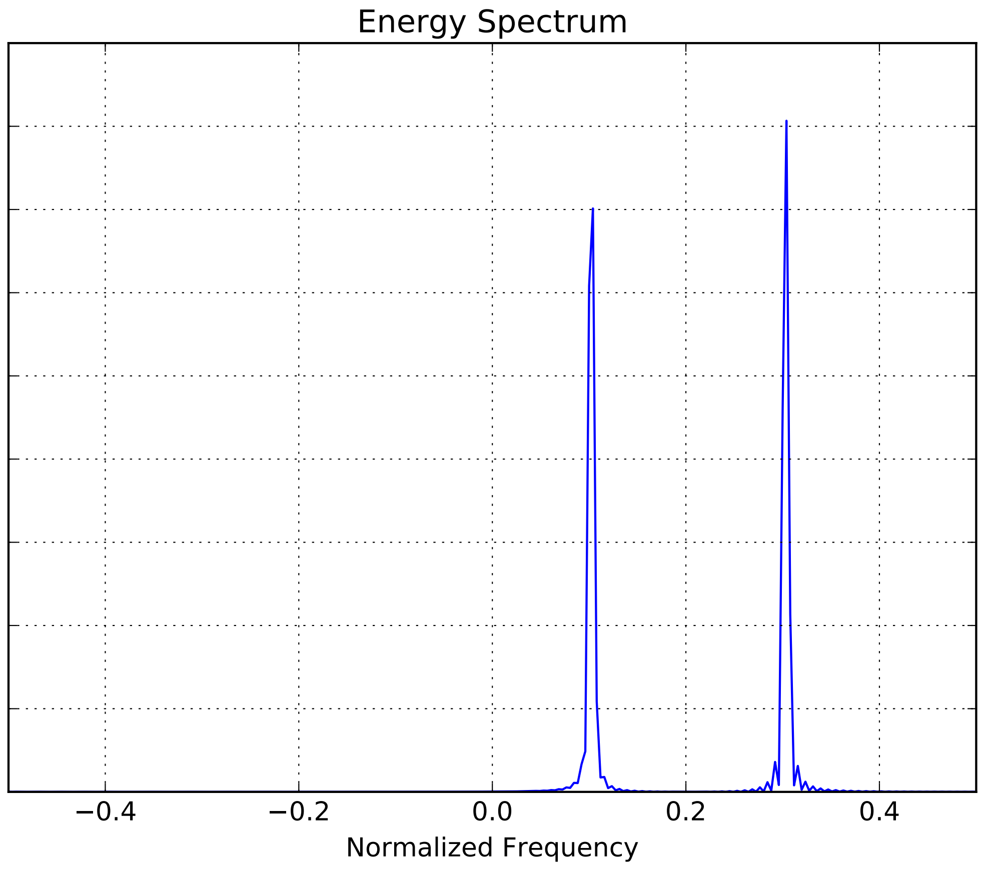
To begin with, let us construct a nonstationary signal, and try to glean its time and frequency characteristics. Consider the signal obtained as follows:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from tftb.generators import fmconst
>>> n_points = 128
>>> mode1, iflaw1 = fmconst(n_points, fnorm=0.1)
>>> mode2, iflaw2 = fmconst(n_points, fnorm=0.3)
>>> signal = np.r_[mode1, mode2]
>>> plt.plot(np.real(signal)), plt.grid(), plt.show()
```



This first half of the signal is a sinusoid with a normalized frequency of 0.1 and the other half has a normalized frequency of 0.3. If we look at the energy spectrum of this signal, sure enough, there are two peaks at the respective frequencies:

```
>>> X = np.fft.fftshift(np.fft.fft(signal))
>>> plt.plot(np.linspace(-0.5, 0.5, 256), np.abs(X) ** 2)
```



Note that the signal produced by the `fmconst` function produces an Analytic Signal. Analytic signals are complex valued, and by definition do not have negative frequency components.

1.1.2 A note on time-frequency analysis

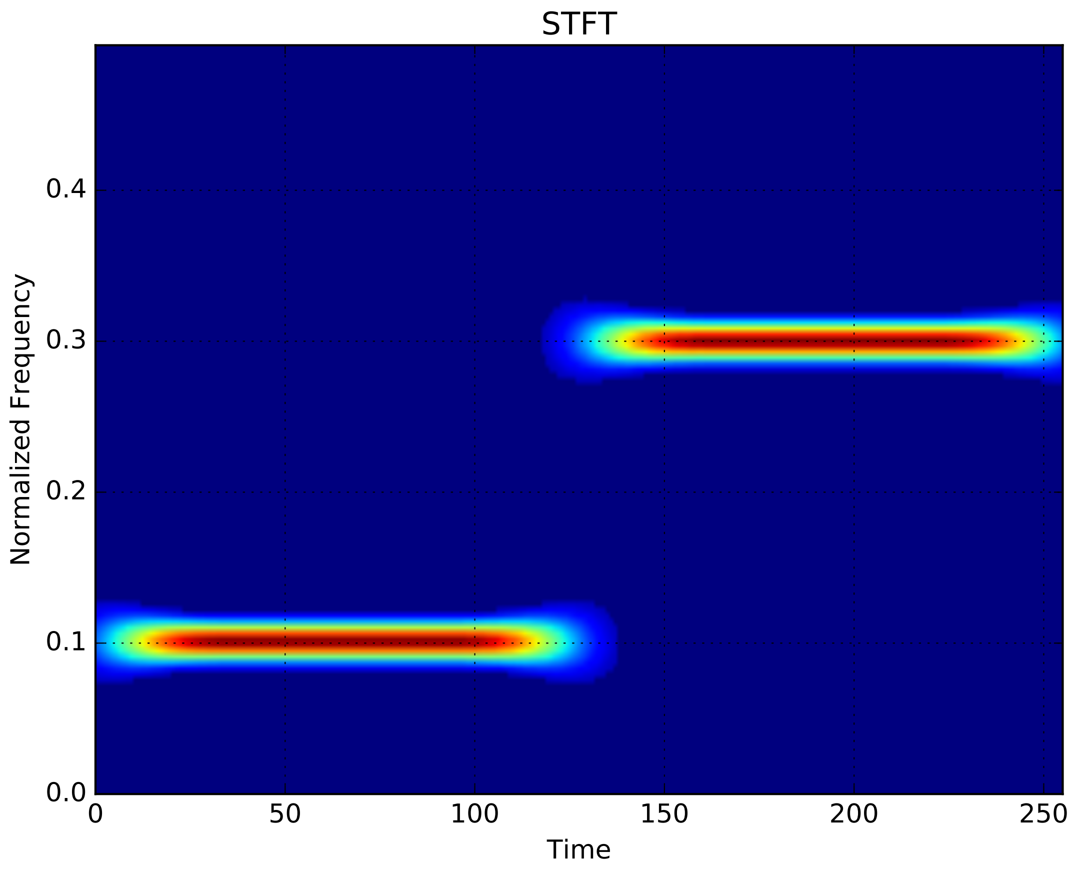
The energy spectrum is perfectly valid, but the Fourier transform is essentially an integral over time. Thus, we lose all information that varies with time. All we can tell from the spectrum is that the signal has two distinct frequency components. In other words, we can comment on *what* happens a signal, 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 in, 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 is 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. HHT too is a tool for time-frequency analysis, as we shall see.

1.1.3 Time-Frequency representations of the signal

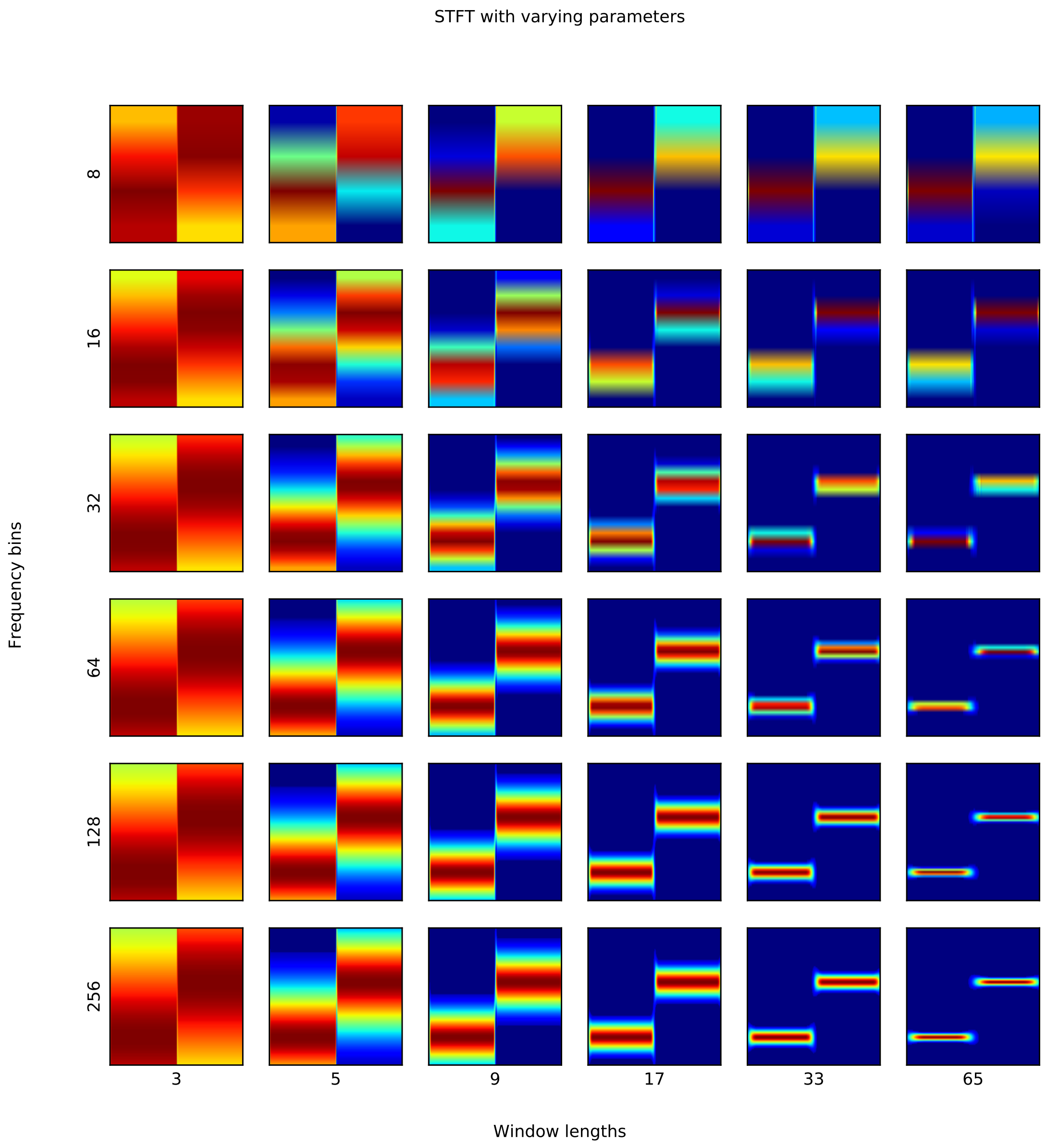
A popular choice to represent both time and frequency characteristics is the short-time Fourier transform (STFT), which, simply put, transforms contiguous chunks of the input and aggregates the result in a 2 dimensional form, where one axis represents frequency and the other represents time. PyTFTB has an STFT implementation which we can use as follows:

```
>>> from tftb.processing import ShortTimeFourierTransform
>>> stft = ShortTimeFourierTransform(signal)
>>> stft.run()
>>> stft.plot()
```



This representation is quite statisfactory. However, there are a number of reasons why it might not always work.

First of all, the short time Fourier transform is parameterized by two important things, other than the signal itself - the number of bins into which the frequency range of the signal is partitioned, and the window function used for smoothing the frequencies. Let's see what happens when we vary the number of frequency bins and the length of the window function.

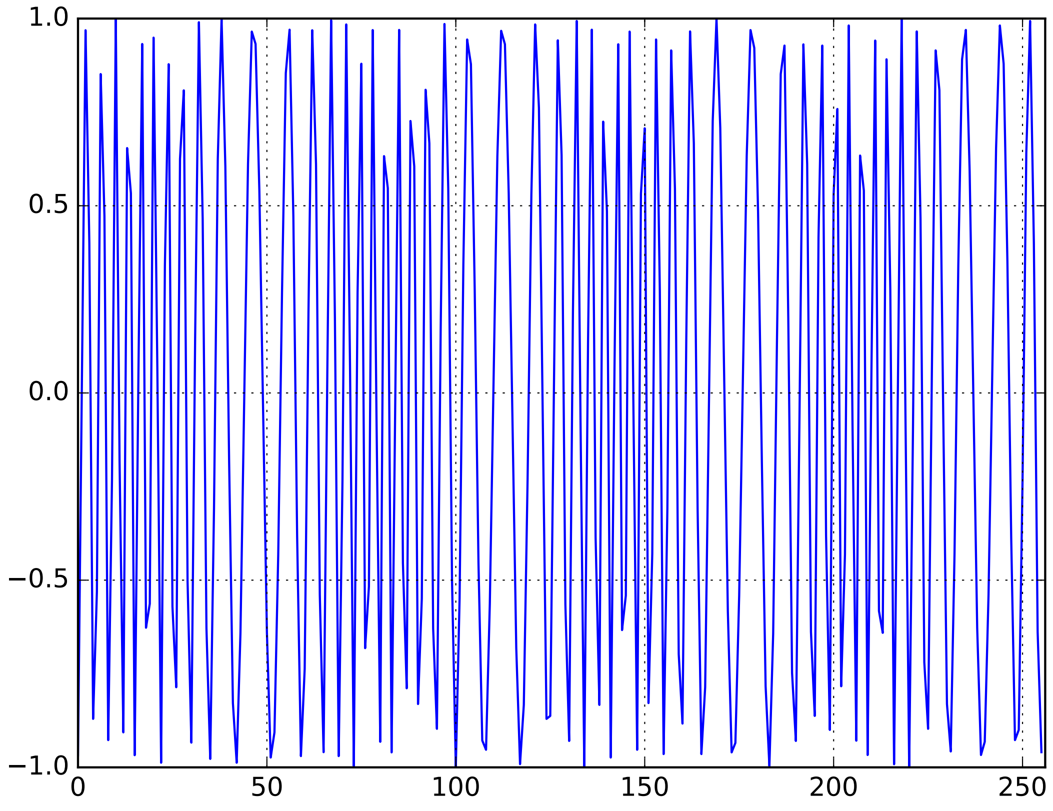


The number of frequency bins increases from top to bottom and the length of the window function increases from left to right. Notice that the representation we are looking for is at the bottom right corner, obtained with 256 frequency bins and a (Hamming) window of length 65. Note that the PyTFTB implementation of the STFT is naive - the default number of frequency bins it uses is equal to the length of the input signal. We can, of course, specify a smaller number, but anything less than 256 would lead to a less than ideal representation. Moreover, these many bins sufficed in this particular case because the frequencies in the input signal are relatively low, in that a sufficient number of cycles can be accommodated within 256 samples. Also, the frequency components are sufficiently separated for the window function to separate them properly. Thus, to find a suitable time-frequency for an arbitrary nonstationary signal, it is likely that we might end up searching the grid shown above, which is highly impractical.

1.1.4 A counterexample

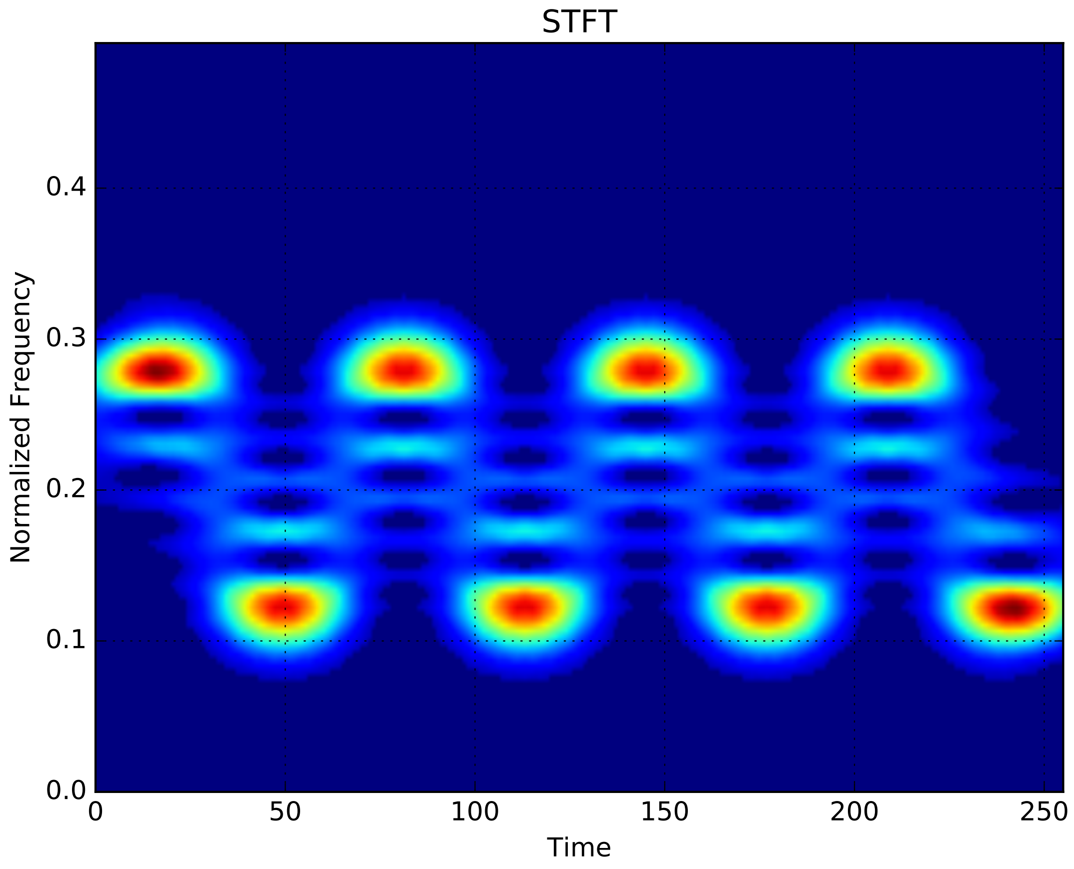
As an example of how this approach can go wrong, take a look at the following signal:

```
>>> from tftb.generators import fmsin
>>> sig, iflaw = fmsin(256, 0.1, 0.3, period=64)
>>> plt.plot(np.real(sig))
```



This signal contains frequencies which are modulated such that they vary sinusoidally between 0.1 and 0.3. The time-frequency distribution of this signal should look somewhat like a sine wave. Here's the spectrogram of the STFT of this signal:

```
>>> stft = ShortTimeFourierTransform(sig)
>>> stft.run()
>>> stft.plot()
```



From this representation, the sinusoidal nature of the frequencies can be made out, and even the concentration of energy at the extrema of the sine wave makes sense. But the artifacts between the high energy areas are quite ambiguous, and bear little resemblance to the signal's true characterisitcs.

Of course, there are a number of heuristics one can apply to make this representation more reasonable - like tweaking the parameters of the STFT, increasing the sampling frequency of the signal, or to use another time-frequency representation altogether. Unfortunately none of these methods are fully data driven, in that they rely very strongly on a parametric model of the data, and the representation is only as good as the model. A major drawback of time frequency distributions that depend on Fourier or wavelet models is that they don't allow for an "unsupervised" or data driven approach to time series analysis.

The Hilbert Huang transform fixes this to a great extent. The following section will deal with how Hilbert spectral analysis is better suited for nonlinear and nonstationary time series data, and how the empirical mode decomposition algorithm makes the results of the Hilber spectral analysis more reasonable.

1.2 Motivation for Hilbert Spectral Analysis

The Fourier transform generalizes Fourier coefficients of a signal over time. Since the Fourier coefficients are the measures of the signal amplitude as a function of frequency, the time information is totally lost, as we saw in the last section. To address this issue there have developed further modifications of the Fourier transform, the most popular of which is the short-time Fourier transform (STFT). The STFT divides the input signal into windows of time and then considers the Fourier transforms of those time windows, thereby achieving some localization of frequency information along the time axis. While practically powerful for most signals, this method cannot be generalized for a broad class of signals because of its *a priori* window lengths. Particularly, the window lengths must be long enough to capture at least one cycle of a component frequency, but not so long as to be redundant. On the other hand, most real-life signals are nonstationary, or have multiple frequency components. The duration of the STFT windows should not be so long as to mix the multiple components during a single operation of the kernel. This might lead to highly undesirable results like the frequency analysis representing multiple components of a nonstationary signal as harmonics of lower components.

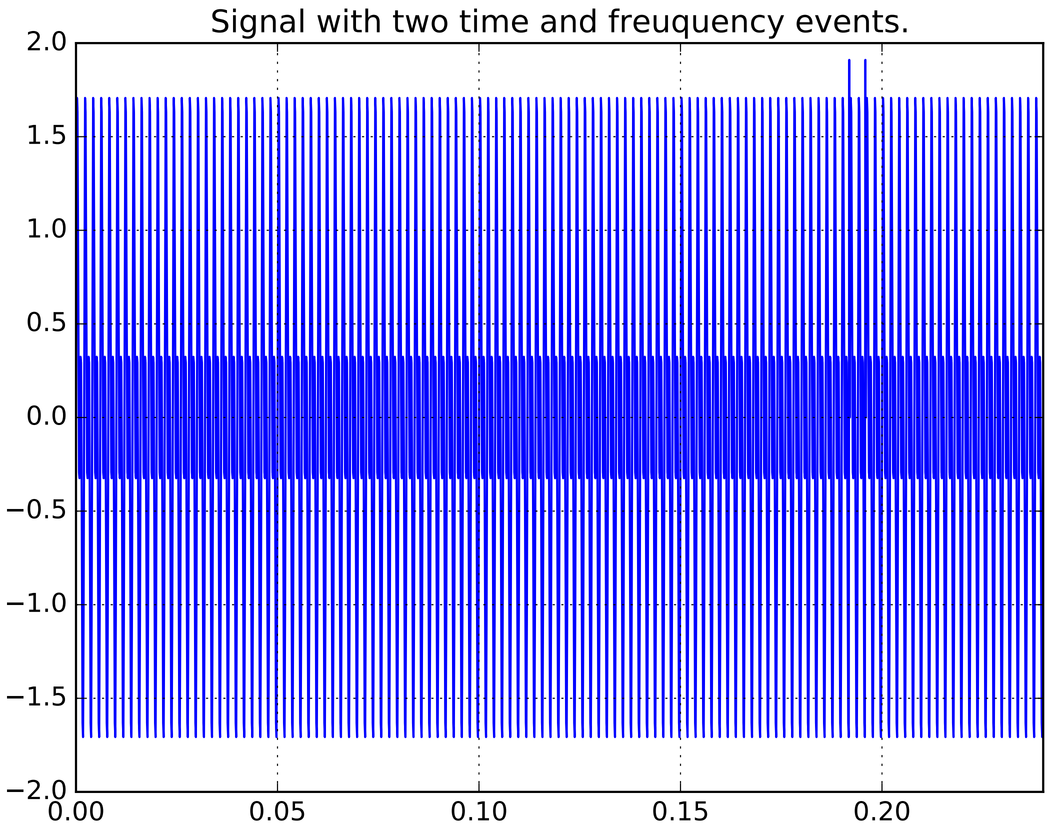
A powerful variant of the Fourier transform is the wavelet transform. By using finite-support basis functions, wavelets are able to approximate even nonstationary data. These basis functions possess most of the desirable properties required for linear decomposition (like orthogonality, completeness , etc) and they can be drawn from a large dictionary of wavelets. This makes the wavelet transform a versatile tool for analysis of nonstationary data. But the wavelet transform is still a linear decomposition and hence suffers from related problems like the uncertainty principle. Moreover, like Fourier, the wavelet transform too is non-adaptive. The basis functions are selected *a priori* and consequently make the wavelet decomposition prone to spurious harmonics and ultimately incorrect interpretations of the data.

A remarkable advantage of Fourier based methods is their mathematical framework. Fourier based methods are so elegant that they make building models for a given dataset very easy. Although such models can represent most of the data and are extensive enough for a practical application, the fact remains that there is some amount of data slipping through the gaps left behind by linear approximations. Despite all these shortcomings, wavelet analysis still remains the best possible method for analysis of nonstationary data, and hence should be used as a reference to establish the validity of other methods.

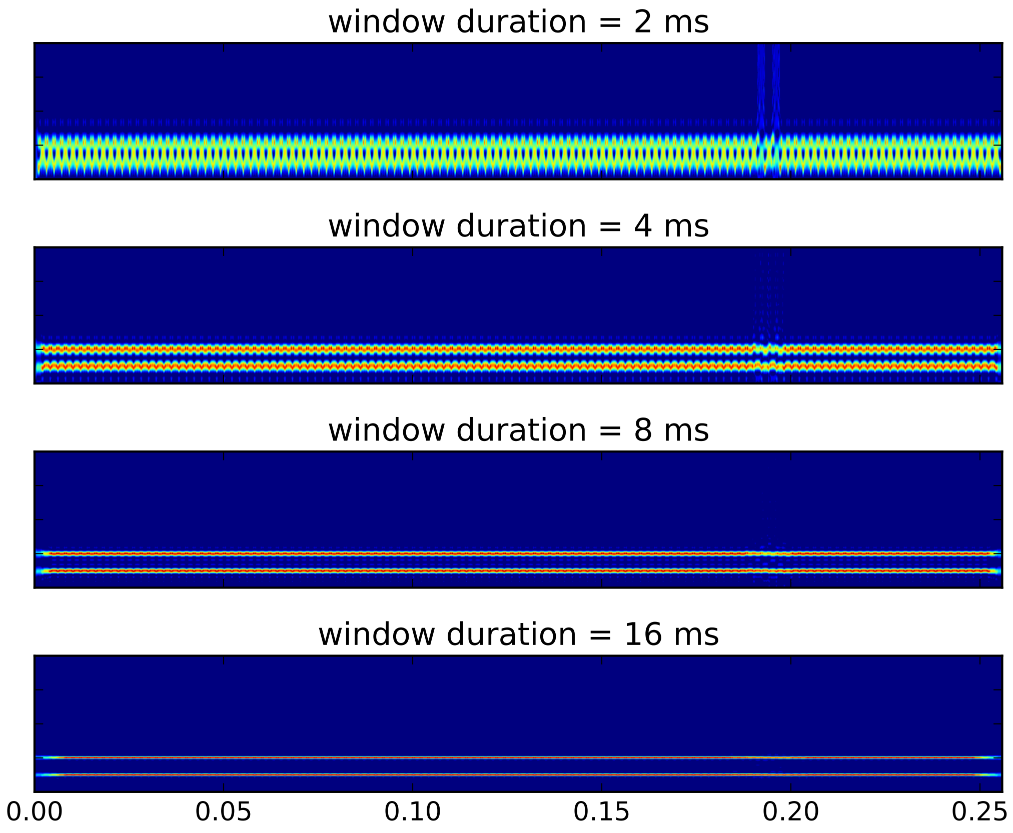
1.2.1 1. The Uncertainty Principle

A very manifest limitation of the Fourier transform can be seen as the uncertainty principle. 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)
```



It is a sum of two sinusiodal signals of frequencies 500 Hz and 1000 Hz. It has two spikes at t = 0.192s and t = 0.196s. 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 the STFTs 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.

This phenomenon is called the Uncertainty principle. Informally, it 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. But unfortunately the period length is defined *a priori*, thereby making the results uncertain in either frequency or time. Mathematically this uncertainty can be quantified with the Heisenberg-Gabor Inequality (also sometimes called the Gabor limit):

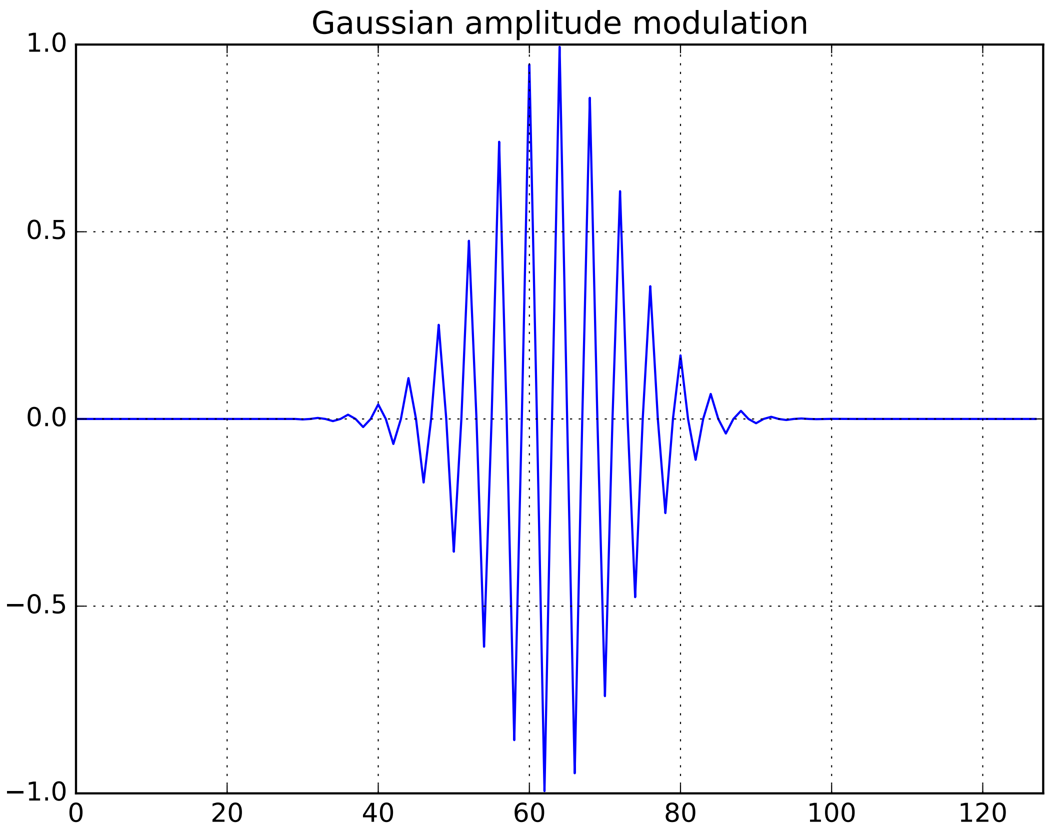
Heisenberg - Gabor Inequality

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.

1.2.2 2. Instantaneous Frequency

As a workaround to the limitations imposed by the Uncertainty principle, we can define a new measure of signal characteristics called the instantaneous frequency. The definition of instantaneous frequency has remained highly controversial ever since its inception, and it is easy to see why. When something is instantaneous it is localized in time. Since time and frequency are inverse quantities, localizing frequency in time can be highly ambiguous. However, a practical definition of instantaneous frequencies is provided in[2], and is discussed in the next section.

[1] http://www.amazon.com/Time-Frequency-Analysis-Theory-Applications/dp/0135945321
[2] http://tftb.nongnu.org/tutorial.pdf

2.1 Analytic Signals and Instantaneous Frequencies

In order to define instantaneous frequencies we must first introduce the concept of analytic signals. For any real valued signal $x(t)$ we associate a complex valued signal $x_a(t)$ defined as:

$$x_a(t) = x(t) + j\widehat{x(t)}$$

where $\widehat{x(t)}$ is the Hilbert transform of $x(t)$. Then the instantaneous frequency can be defined as:

$$\nu_{inst} = \frac{1}{2\pi}\frac{d}{dt}\arctan[x_a(t)]$$

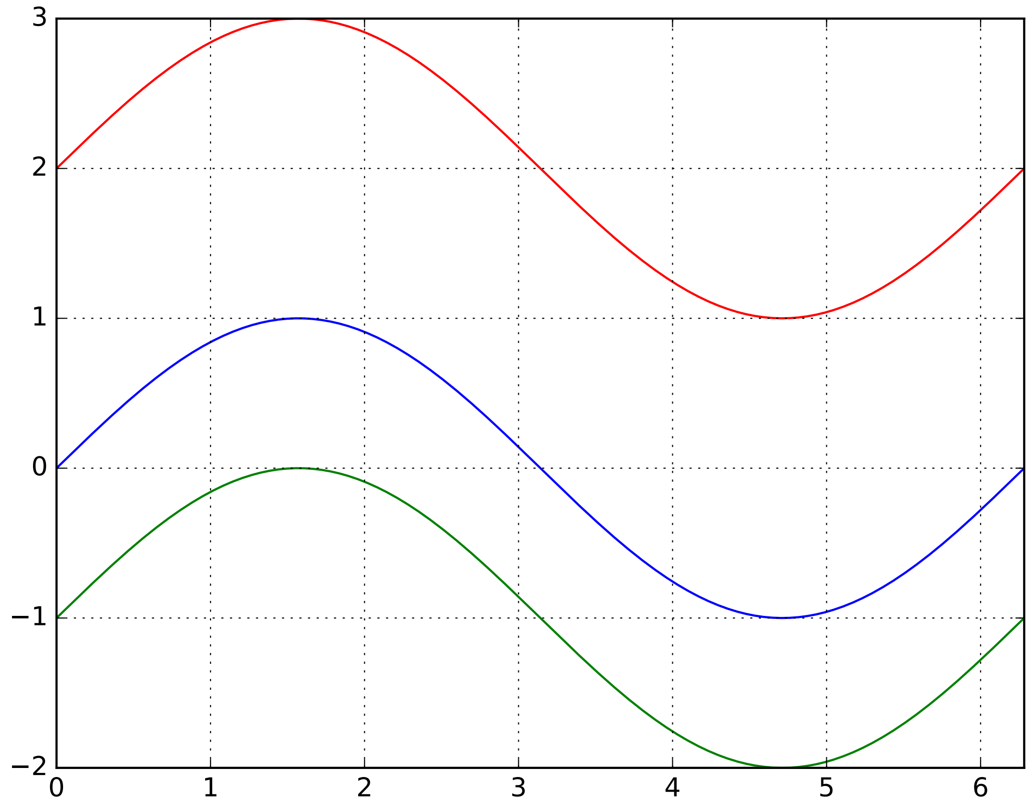
2.2 Instantaneous Frequencies from HHT

The real innovation of the HHT is an iterative algorithm called the Empirical Mode Decomposition (EMD) which breaks a signal down into so-called Intrinsic Mode Functions (IMFs) which are characterized by being narrowband, nearly monocomponent and having a large time-bandwidth product. This allows the IMFs to have well-defined Hilbert transforms and consequently, physically meaningful instantaneous frequencies. In the next couple of sections we briefly describe IMFs and the algorithm, EMD, used to obtain them.

2.3 Intrinsic Mode Functions

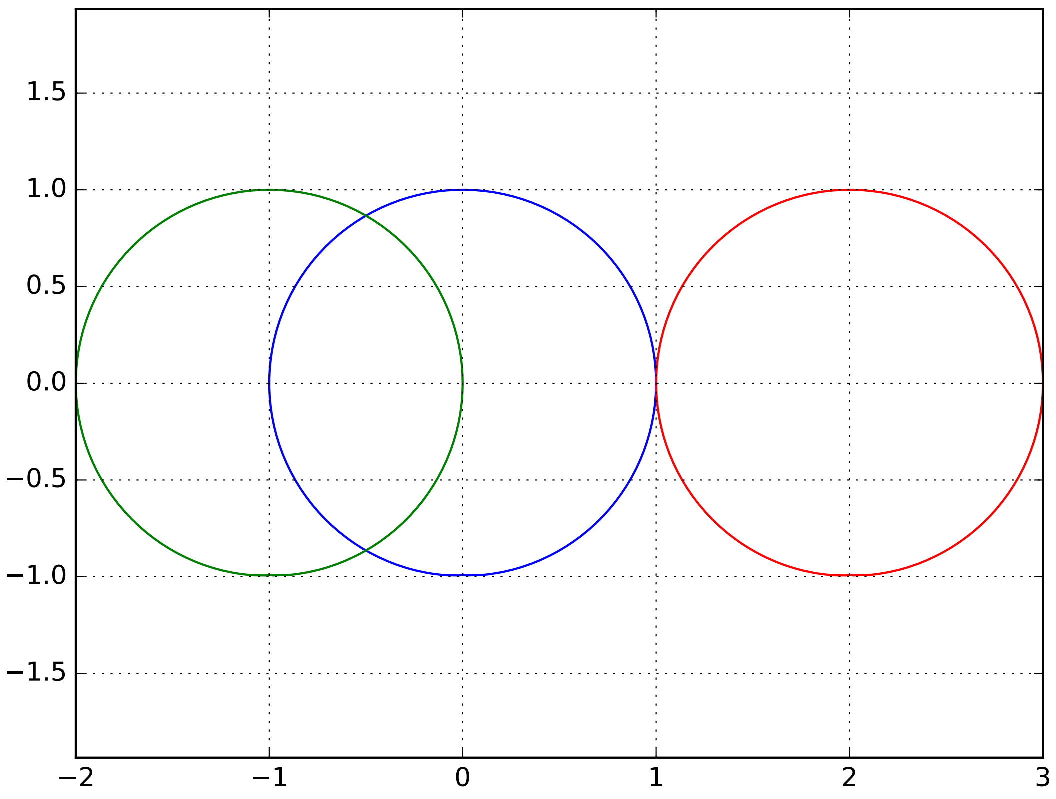
Consider the three sinusoidal signals obtained as follows:

```
>>> x = np.linspace(0, 2 * np.pi, 1000)
>>> s1 = np.sin(x)
>>> s2 = np.sin(x) - 1
>>> s3 = np.sin(x) + 2
>>> plt.plot(x, s1, 'b', x, s2, 'g', x, s3, 'r')
```



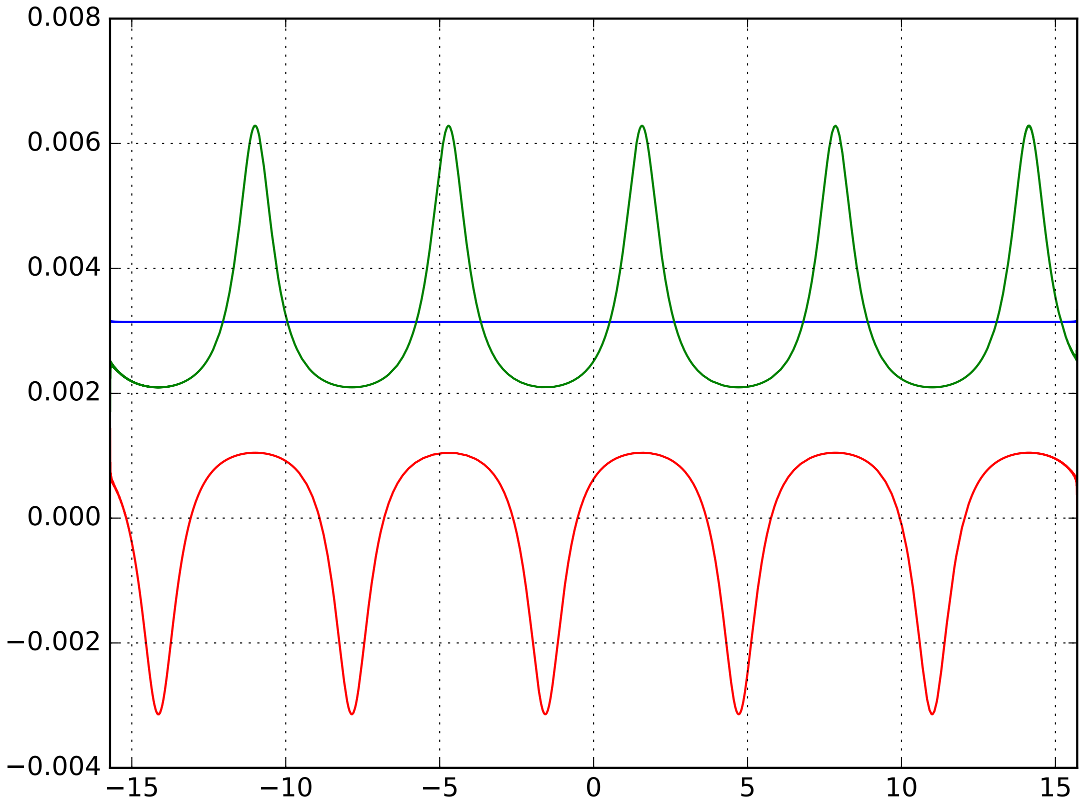
All of them are identical, except that two of them have a nonzero DC component. Since the Hilbert transform of sine is cosine, the analytic signals of these sinusoids should represent unit circles in the complex plane:

```
>>> from scipy.signal import hilbert
>>> hs1 = hilbert(s1)
>>> hs2 = hilbert(s2)
>>> hs3 = hilbert(s3)
>>> plt.plot(np.real(hs1), np.imag(hs1), 'b')
>>> plt.plot(np.real(hs2), np.imag(hs2), 'g')
>>> plt.plot(np.real(hs3), np.imag(hs3), 'r')
```



Imagine that each circle is traced out by a phasor rotating anticlockwise, which is centered at the origin in the figure above. The angle that the phasor rotates through in an infinitesimally small time period represents the instantaneous phase of the signal, and its time differential is the instantaneous frequency. Using this interpretation, let's try to compute the instantaneous frequencies of the three signals:

```
>>> from scipy import angle, unwrap
>>> omega_s1 = unwrap(angle(hs1))  # unwrapped instantaneous phase
>>> omega_s2 = unwrap(angle(hs2))
>>> omega_s3 = unwrap(angle(hs3))
>>> f_inst_s1 = np.diff(omega_s1)  # instantaneous frequency
>>> f_inst_s2 = np.diff(omega_s2)
>>> f_inst_s3 = np.diff(omega_s3)
>>> plt.plot(x[1:], f_inst_s1, "b")
>>> plt.plot(x[1:], f_inst_s2, "g")
>>> plt.plot(x[1:], f_inst_s3, "r")
>>> plt.show()
```



As shown in the figure, only one sinusoid presents an instantaneous frequency that is constant and corresponds to the true frequency of the waves. This wave is the one which has its analytical signal centered around the origin, thereby allowing the phasor to rotate through a total angle of 2π in one period. This is the wave that has a zero DC component and is symmetrical around the time axis.

The fact that true instantaneous frequencies are reproduced only when the signal is symmetric about the X-axis motivates the definition of an IMF.

Intrinsic Mode Functions

A function is called an intrinsic mode function when:

1. The number of its extrema and zero-crossings differ at most by unity.
2. The mean of the local envelopes defined by its local maxima and that defined by its local minima should be zero at all times.

Condition 1 ensures that there are no localized oscillations in the signal and it crosses the X-axis at least once before it goes from one extremum to another, which makes it adaptive. Condition 2 ensures meaningful instantaneous frequencies, as explained in the previous example. The next section explains the algorithm for extracting IMFs out of a signal.

2.4 Empirical Mode Decomposition

The EMD is an iterative algorithm which breaks a signal down into IMFs. The process is performed as follows:

1. Find all local extrema in the signal.
2. Join all the local maxima with a cubic spline, creating an upper envelope. Repeat for local minima and create a lower envelope.
3. Calculate the mean of the envelopes.
4. Subtract mean from original signals.
5. Repeat steps 1-4 until result is an IMF.
6. Subtract this IMF from the original signal.
7. Repeat steps 1-6 till there are no more IMFs left in the signal.

The next tutorial demonstrates how EMD can be used with PyHHT.

2.5 Properties of Intrinsic Mode Functions

By virtue of the EMD algorithm, the decomposition is complete, in that the sum of the IMFs and the residue subtracted from the input signal leaves behind only a negligible residue. The decomposition is almost orthogonal. Also, as emphasized earlier, the greatest advantage of the IMFs are well-behaved Hilbert transforms, enabling the extraction of physically meaningful instantaneous frequencies.

IMFs have large time-bandwidth products, which indicates that they tend to move away from the lower bound of the Heisenberg-Gabor inequality, thereby avoiding the limitations of the Uncertainty principle, as explained in section 1.

1.2.3 3. Two Views of Nonlinear Phenomena

Despite all its robustness and convenience, the Hilbert-Huang transform is unfortunately just an algorithm, without a well-defined mathematical base. All inferences drawn from it are empirical and can only be corroborated as such. It lacks the mathematical sophistication of the Fourier framework. On the plus side it provides a very realistic insight into data.

Thus here we have room for a tradeoff between the mathematical elegance of the Fourier analysis and the physical significance provided by the Hilbert-Huang transform. Wavelets are the closest thing to the HHT that not only have the ability to analyze nonlinear and nonstationary phenomena, but also a complete mathematical foundation. Unfortunately wavelets are not adaptive and as such might suffer from problems like uncertainty principle, leakages, Gibb's phenomenon, harmonics, etc - like most of the decomposition techniques that use a priori basis functions. On the other hand, the basis functions of the HHT are IMFs which are adaptive and empirical. But EMD is not a perfect algorithm. For many signals it does not converge down to a set of finite IMFs. Some experts even believe that there is an inherent contradiction between the way IMFs are defined and the way EMD is executed. This means that we can possibly use wavelets as a 'handle' for the appropriate extraction of IMFs, and conversely, use IMFs to establish the physical relevance of wavelet decomposition.

Thus the Hilbert-Huang transform is a alternate view of nonlinear and nonstationary phenomena, one that is unencumbered by mathematical jargon. This lack of mathematical sophistication allows researchers to be very flexible and versatile with its use.

1.2.4 4. Conclusion

Consider a dark room with a photosensitive device. Suppose a light flashes upon the device at a given instant. The Fourier interpretation of this phenomenon would be to consider a number of (ideally infinitely many) of frequencies which are in phase exactly at the time when the light is flashed. The frequencies interfere constructively at that instant to produce the flash of light and cancel each other out at all the other times. The truth of the matter remains that there are not so many frequency 'events' to speak of. But the Fourier interpretation is mathematically so elegant that sometimes it drives the physical significance out of the model.

The Hilbert-Huang transform, on the other hand, gives prevalence only to physically meaningful events. The extraction of instantaneous frequencies does not depend on convolution (as in the Fourier model), but on time derivatives. The bases are not chosen *a priori*, but are adaptive. A complementary use of these two paradigms to analyze nonlinear and nonstationary phenomena has great research potential.

The next tutorial is a comprehensive guide to PyHHT, and provides a detailed overview of how different aspects of the HHT can be harnessed with the module.

1.3 Using PyHHT: EMD and Hilbert Spectral Analysis

CHAPTER 2

pyhht package

2.1 Submodules

2.2 pyhht.emd module

Empirical Mode Decomposition.

`pyhht.emd.`**`EMD`**

alias of `pyhht.emd.EmpiricalModeDecomposition`

`class` `pyhht.emd.`**`EmpiricalModeDecomposition`**(*x, t=None, threshold_1=0.05, threshold_2=0.5, alpha=0.05, ndirs=4, fixe=0, maxiter=2000, fixe_h=0, n_imfs=0, nbsym=2, bivariate_mode='bbox_center'*)

Bases: `object`

The EMD class.

Methods

`decompose`()	Decompose the input signal into IMFs.
`io`()	Compute the index of orthoginality, as defined by:
`keep_decomposing`()	Check whether to continue the sifting operation.
`mean_and_amplitude`(m)	Compute the mean of the envelopes and the mode amplitudes.
`stop_EMD`()	Check if there are enough extrema (3) to continue sifting.
`stop_sifting`(m)	Evaluate the stopping criteria for the current mode.

__init__(*x, t=None, threshold_1=0.05, threshold_2=0.5, alpha=0.05, ndirs=4, fixe=0, maxiter=2000, fixe_h=0, n_imfs=0, nbsym=2, bivariate_mode='bbox_center'*)

Empirical mode decomposition.

Parameters

x [array-like, shape (n_samples,)] The signal on which to perform EMD

t [array-like, shape (n_samples,), optional] The timestamps of the signal.

threshold_1 [float, optional] Threshold for the stopping criterion, corresponding to θ_1 in [3]. Defaults to 0.05.

threshold_2 [float, optional] Threshold for the stopping criterion, corresponding to θ_2 in [3]. Defaults to 0.5.

alpha [float, optional] Tolerance for the stopping criterion, corresponding to α in [3]. Defaults to 0.05.

ndirs [int, optional] Number of directions in which interpolants for envelopes are computed for bivariate EMD. Defaults to 4. This is ignored if the signal is real valued.

fixe [int, optional] Number of sifting iterations to perform for each IMF. By default, the stopping criterion mentioned in [1] is used. If set to a positive integer, each mode is either the result of exactly *fixe* number of sifting iterations, or until a pure IMF is found, whichever is sooner.

maxiter [int, optional] Upper limit of the number of sifting iterations for each mode. Defaults to 2000.

n_imfs [int, optional] Number of IMFs to extract. By default, this is ignored and decomposition is continued until a monotonic trend is left in the residue.

nbsym [int, optional] Number of extrema to use to mirror the signals on each side of their boundaries.

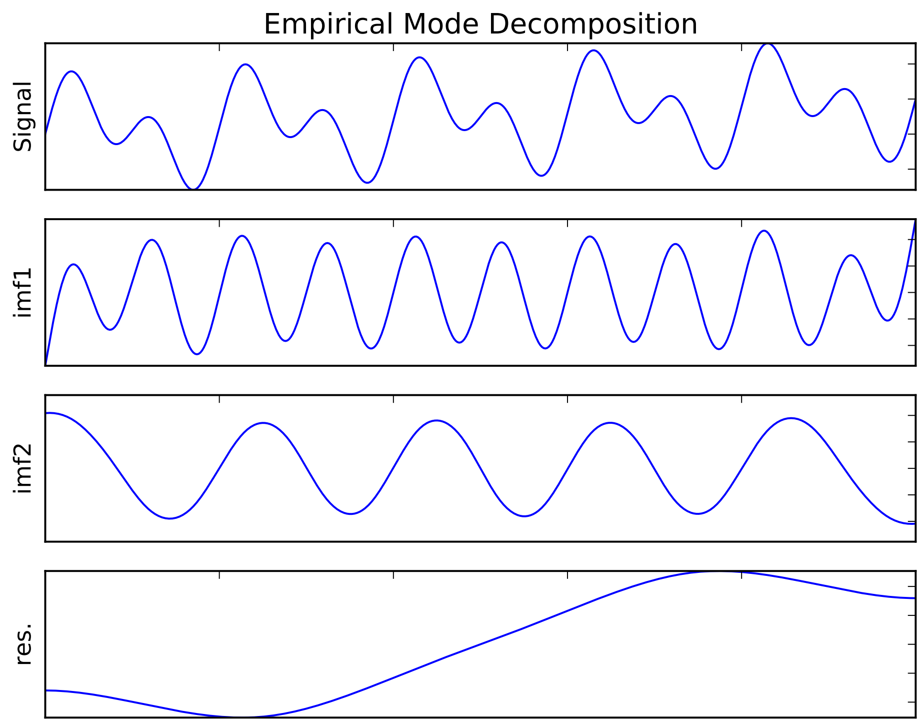
bivariate_mode [str, optional] The algorithm to be used for bivariate EMD as described in [4]. Can be one of 'centroid' or 'bbox_center'. This is ignored if the signal is real valued.

References

[1], *[2]*, *[3]*, *[4]*

Examples

```
>>> from pyhht.visualization import plot_imfs
>>> import numpy as np
>>> t = np.linspace(0, 1, 1000)
>>> modes = np.sin(2 * pi * 5 * t) + np.sin(2 * pi * 10 * t)
>>> x = modes + t
>>> decomposer = EMD(x)
>>> imfs = decomposer.decompose()
>>> plot_imfs(x, imfs, t)
```



Attributes

is_bivariate [bool] Whether the decomposer performs bivariate EMD. This is automatically determined by the input value. This is True if at least one non-zero imaginary component is found in the signal.

nbits [list] List of number of sifting iterations it took to extract each IMF.

decompose()

Decompose the input signal into IMFs.

This function does all the heavy lifting required for sifting, and should ideally be the only public method of this class.

Returns

imfs [array-like, shape (n_imfs, n_samples)] A matrix containing one IMF per row.

Examples

```
>>> from pyhht.visualization import plot_imfs
>>> import numpy as np
>>> t = np.linspace(0, 1, 1000)
>>> modes = np.sin(2 * pi * 5 * t) + np.sin(2 * pi * 10 * t)
>>> x = modes + t
>>> decomposer = EMD(x)
>>> imfs = decomposer.decompose()
```

io()

Compute the index of orthoginality, as defined by:

$$\sum_{i,j=1,i\neq j}^{N} \frac{\|C_i \overline{C_j}\|}{\|x\|^2}$$

Where C_i is the i th IMF.

Returns

float Index of orthogonality. Lower values are better.

Examples

```
>>> import numpy as np
>>> t = np.linspace(0, 1, 1000)
>>> modes = np.sin(2 * pi * 5 * t) + np.sin(2 * pi * 10 * t)
>>> x = modes + t
>>> decomposer = EMD(x)
>>> imfs = decomposer.decompose()
>>> print('%.3f' % decomposer.io())
0.017
```

keep_decomposing()

Check whether to continue the sifting operation.

mean_and_amplitude(*m*)

Compute the mean of the envelopes and the mode amplitudes.

Parameters

m [array-like, shape (n_samples,)] The input array or an itermediate value of the sifting process.

Returns

tuple A tuple containing the mean of the envelopes, the number of extrema, the number of zero crosssing and the estimate of the amplitude of themode.

stop_EMD()

Check if there are enough extrema (3) to continue sifting.

Returns

bool Whether to stop further cubic spline interpolation for lack of local extrema.

stop_sifting(*m*)

Evaluate the stopping criteria for the current mode.

Parameters

m [array-like, shape (n_samples,)] The current mode.

Returns

bool Whether to stop sifting. If this evaluates to true, the current mode is interpreted as an IMF.

2.3 pyhht.utils module

Utility functions used to inspect EMD functionality.

`pyhht.utils.`**`boundary_conditions`**`(`*signal*, *time_samples*, *z=None*, *nbsym=2*`)`

Extend a 1D signal by mirroring its extrema on either side.

Parameters

signal [array-like, shape (n_samples,)] The input signal.

time_samples [array-like, shape (n_samples,)] Timestamps of the signal samples

z [array-like, shape (n_samples,), optional] A proxy signal on whose extrema the interpolation is evaluated. Defaults to *signal*.

nbsym [int, optional] The number of extrema to consider on either side of the signal. Defaults to 2

Returns

tuple A tuple of four arrays which represent timestamps of the minima of the extended signal, timestamps of the maxima of the extended signal, minima of the extended signal and maxima of the extended signal. signal, minima of the extended signal and maxima of the extended signal.

Examples

```
>>> from __future__ import print_function
>>> import numpy as np
>>> signal = np.array([-1, 1, -1, 1, -1])
>>> tmin, tmax, vmin, vmax = boundary_conditions(signal, np.arange(5))
>>> tmin
array([-2,  2,  6])
>>> tmax
array([-3, -1,  1,  3,  5,  7])
>>> vmin
array([-1, -1, -1])
>>> vmax
array([1, 1, 1, 1, 1, 1])
```

`pyhht.utils.`**`extr`**`(`*x*`)`

Extract the indices of the extrema and zero crossings.

Parameters

x [array-like, shape (n_samples,)] Input signal.

Returns

tuple A tuple of three arrays representing the minima, maxima and zero crossings of the signal respectively.

Examples

```
>>> from __future__ import print_function
>>> import numpy as np
>>> x = np.array([0, -2, 0, 1, 3, 0.5, 0, -1, -1])
>>> indmin, indmax, indzer = extr(x)
>>> print(indmin)
[1]
>>> print(indmax)
[4]
>>> print(indzer)
[0 2 6]
```

pyhht.utils.**get_envelops**(*x*, *t=None*)

Get the upper and lower envelopes of an array, as defined by its extrema.

Parameters

x [array-like, shape (n_samples,)] The input array.

t [array-like, shape (n_samples,), optional] Timestamps of the signal. Defaults to *np.arange(n_samples,)*

Returns

tuple A tuple of arrays representing the upper and the lower envelopes respectively.

Examples

```
>>> import numpy as np
>>> x = np.random.rand(100,)
>>> upper, lower = get_envelops(x)
```

pyhht.utils.**inst_freq**(*x*, *t=None*)

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

Parameters

x [array-like, shape (n_samples,)] The input analytic signal.

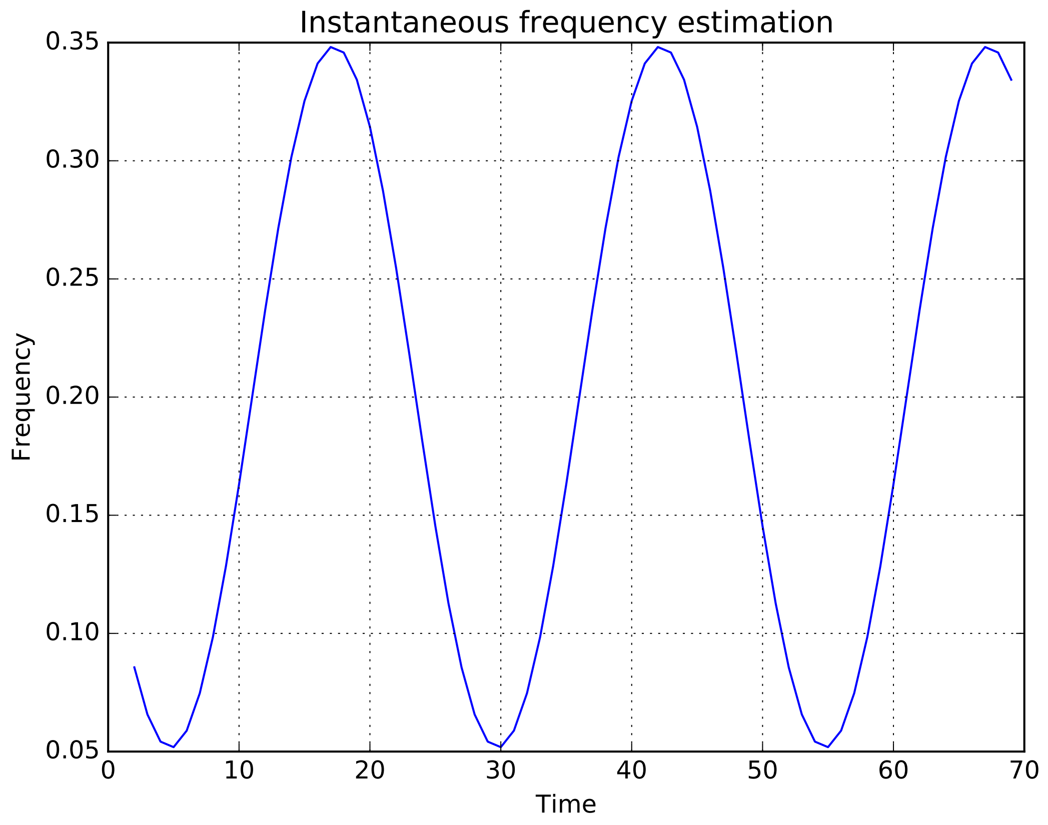
t [array-like, shape (n_samples,), optional] The time instants at which to calculate the instantaneous frequency. Defaults to *np.arange(2, n_samples)*

Returns

array-like Normalized instantaneous frequencies of the input signal

Examples

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



2.4 pyhht.visualization module

Visualization functions for PyHHT.

`pyhht.visualization.`**`plot_imfs`**(*signal*, *imfs*, *time_samples=None*, *fignum=None*, *show=True*)

Plot the signal, IMFs and residue.

Parameters

signal [array-like, shape (n_samples,)] The input signal.

imfs [array-like, shape (n_imfs, n_samples)] Matrix of IMFs as generated with the *EMD.decompose* method.

time_samples [array-like, shape (n_samples), optional] Time instants of the signal samples. (defaults to *np.arange(1, len(signal))*)

fignum [int, optional] Matplotlib figure number (by default a new figure is created)

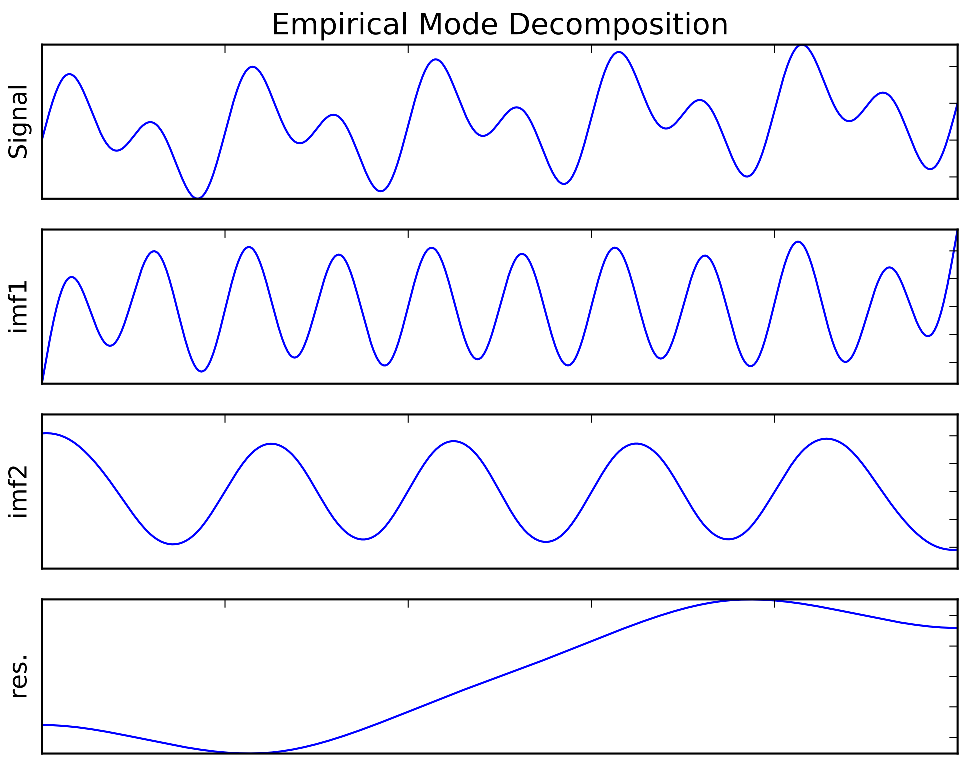
show [bool, optional] Whether to display the plot. Defaults to True, set to False if further plotting needs to be done.

Returns

'matplotlib.figure.Figure' The figure (new or existing) in which the decomposition is plotted.

Examples

```
>>> from pyhht.visualization import plot_imfs
>>> import numpy as np
>>> from pyhht import EMD
>>> t = np.linspace(0, 1, 1000)
>>> modes = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 10 * t)
>>> x = modes + t
>>> decomposer = EMD(x)
>>> imfs = decomposer.decompose()
>>> plot_imfs(x, imfs, t)
```



2.5 Module contents

`pyhht.`**`EMD`**

alias of *`pyhht.emd.EmpiricalModeDecomposition`*

`class` `pyhht.`**`EmpiricalModeDecomposition`**(*x, t=None, threshold_1=0.05, threshold_2=0.5, alpha=0.05, ndirs=4, fixe=0, maxiter=2000, fixe_h=0, n_imfs=0, nbsym=2, bivariate_mode='bbox_center'*)

Bases: `object`

The EMD class.

Methods

decompose()	Decompose the input signal into IMFs.
io()	Compute the index of orthoginality, as defined by:
keep_decomposing()	Check whether to continue the sifting operation.
mean_and_amplitude(m)	Compute the mean of the envelopes and the mode amplitudes.
stop_EMD()	Check if there are enough extrema (3) to continue sifting.
stop_sifting(m)	Evaluate the stopping criteria for the current mode.

__init__(*x, t=None, threshold_1=0.05, threshold_2=0.5, alpha=0.05, ndirs=4, fixe=0, maxiter=2000, fixe_h=0, n_imfs=0, nbsym=2, bivariate_mode='bbox_center'*)

Empirical mode decomposition.

Parameters

x [array-like, shape (n_samples,)] The signal on which to perform EMD

t [array-like, shape (n_samples,), optional] The timestamps of the signal.

threshold_1 [float, optional] Threshold for the stopping criterion, corresponding to θ_1 in [3]. Defaults to 0.05.

threshold_2 [float, optional] Threshold for the stopping criterion, corresponding to θ_2 in [3]. Defaults to 0.5.

alpha [float, optional] Tolerance for the stopping criterion, corresponding to α in [3]. Defaults to 0.05.

ndirs [int, optional] Number of directions in which interpolants for envelopes are computed for bivariate EMD. Defaults to 4. This is ignored if the signal is real valued.

fixe [int, optional] Number of sifting iterations to perform for each IMF. By default, the stopping criterion mentioned in [1] is used. If set to a positive integer, each mode is either the result of exactly *fixe* number of sifting iterations, or until a pure IMF is found, whichever is sooner.

maxiter [int, optional] Upper limit of the number of sifting iterations for each mode. Defaults to 2000.

n_imfs [int, optional] Number of IMFs to extract. By default, this is ignored and decomposition is continued until a monotonic trend is left in the residue.

nbsym [int, optional] Number of extrema to use to mirror the signals on each side of their boundaries.

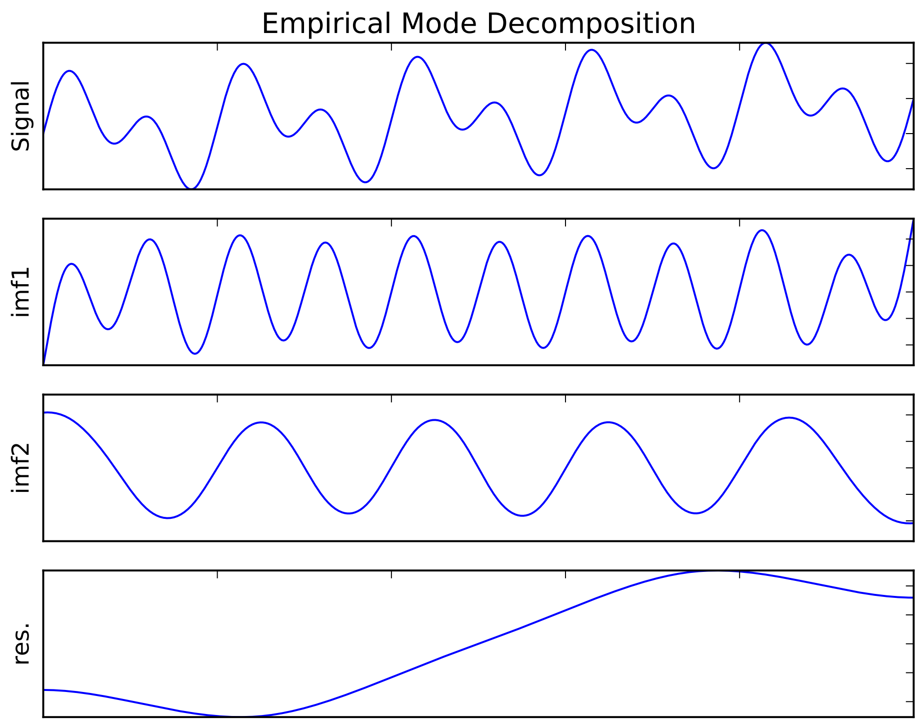
bivariate_mode [str, optional] The algorithm to be used for bivariate EMD as described in [4]. Can be one of 'centroid' or 'bbox_center'. This is ignored if the signal is real valued.

References

[1], *[2]*, *[3]*, *[4]*

Examples

```
>>> from pyhht.visualization import plot_imfs
>>> import numpy as np
>>> t = np.linspace(0, 1, 1000)
>>> modes = np.sin(2 * pi * 5 * t) + np.sin(2 * pi * 10 * t)
>>> x = modes + t
>>> decomposer = EMD(x)
>>> imfs = decomposer.decompose()
>>> plot_imfs(x, imfs, t)
```



Attributes

is_bivariate [bool] Whether the decomposer performs bivariate EMD. This is automatically determined by the input value. This is True if at least one non-zero imaginary component is found in the signal.

nbits [list] List of number of sifting iterations it took to extract each IMF.

decompose()

Decompose the input signal into IMFs.

This function does all the heavy lifting required for sifting, and should ideally be the only public method of this class.

Returns

imfs [array-like, shape (n_imfs, n_samples)] A matrix containing one IMF per row.

Examples

```
>>> from pyhht.visualization import plot_imfs
>>> import numpy as np
>>> t = np.linspace(0, 1, 1000)
>>> modes = np.sin(2 * pi * 5 * t) + np.sin(2 * pi * 10 * t)
>>> x = modes + t
>>> decomposer = EMD(x)
>>> imfs = decomposer.decompose()
```

io()

Compute the index of orthoginality, as defined by:

$$\sum_{i,j=1,i\neq j}^{N} \frac{\|C_i\overline{C_j}\|}{\|x\|^2}$$

Where C_i is the i th IMF.

Returns

float Index of orthogonality. Lower values are better.

Examples

```
>>> import numpy as np
>>> t = np.linspace(0, 1, 1000)
>>> modes = np.sin(2 * pi * 5 * t) + np.sin(2 * pi * 10 * t)
>>> x = modes + t
>>> decomposer = EMD(x)
>>> imfs = decomposer.decompose()
>>> print('%.3f' % decomposer.io())
0.017
```

keep_decomposing()

Check whether to continue the sifting operation.

mean_and_amplitude(*m*)

Compute the mean of the envelopes and the mode amplitudes.

Parameters

m [array-like, shape (n_samples,)] The input array or an itermediate value of the sifting process.

Returns

tuple A tuple containing the mean of the envelopes, the number of extrema, the number of zero crosssing and the estimate of the amplitude of themode.

stop_EMD()

Check if there are enough extrema (3) to continue sifting.

Returns

bool Whether to stop further cubic spline interpolation for lack of local extrema.

stop_sifting(*m*)

Evaluate the stopping criteria for the current mode.

Parameters

m [array-like, shape (n_samples,)] The current mode.

Returns

bool Whether to stop sifting. If this evaluates to true, the current mode is interpreted as an IMF.

`pyhht.plot_imfs`(*signal*, *imfs*, *time_samples=None*, *fignum=None*, *show=True*)

Plot the signal, IMFs and residue.

Parameters

signal [array-like, shape (n_samples,)] The input signal.

imfs [array-like, shape (n_imfs, n_samples)] Matrix of IMFs as generated with the *EMD.decompose* method.

time_samples [array-like, shape (n_samples), optional] Time instants of the signal samples. (defaults to *np.arange(1, len(signal))*)

fignum [int, optional] Matplotlib figure number (by default a new figure is created)

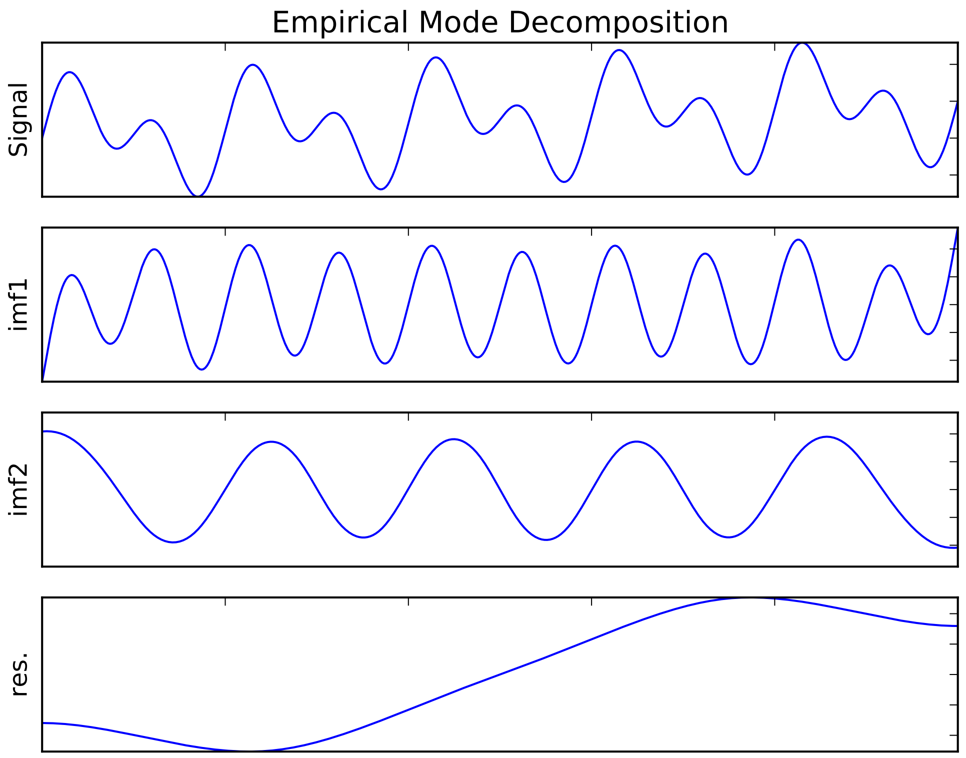
show [bool, optional] Whether to display the plot. Defaults to True, set to False if further plotting needs to be done.

Returns

'matplotlib.figure.Figure' The figure (new or existing) in which the decomposition is plotted.

Examples

```
>>> from pyhht.visualization import plot_imfs
>>> import numpy as np
>>> from pyhht import EMD
>>> t = np.linspace(0, 1, 1000)
>>> modes = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 10 * t)
>>> x = modes + t
>>> decomposer = EMD(x)
>>> imfs = decomposer.decompose()
>>> plot_imfs(x, imfs, t)
```


Empirical Mode Decomposition
Signal
imf1
imf2
res.

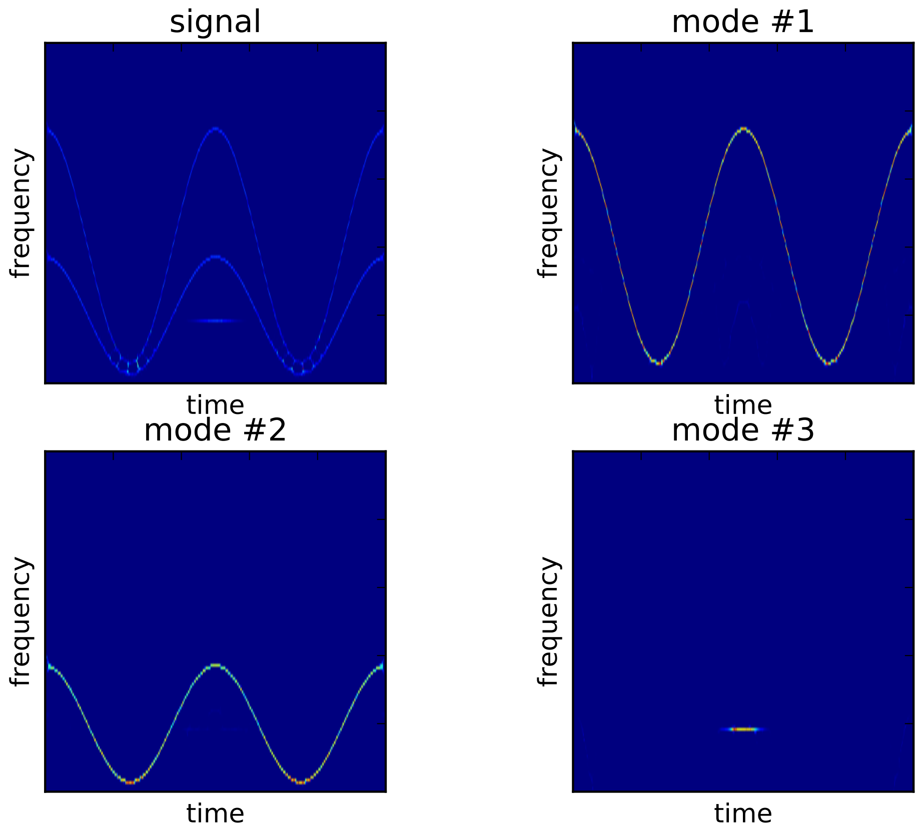
CHAPTER 3

pyhht

CHAPTER 4

PyHHT Examples

- Reassigned spectrogram of a signal having sinusoidal frequency modulation



CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] Huang H. et al. 1998 ‘The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis.’ Procedings of the Royal Society 454, 903-995

[2] Zhao J., Huang D. 2001 ‘Mirror extending and circular spline function for empirical mode decomposition method’. Journal of Zhejiang University (Science) V.2, No.3, 247-252

[3] Gabriel Rilling, Patrick Flandrin, Paulo Gonçalves, June 2003: ‘On Empirical Mode Decomposition and its Algorithms’, IEEE-EURASIP Workshop on Nonlinear Signal and Image Processing NSIP-03

[4] Gabriel Rilling, Patrick Flandrin, Paulo Gonçalves, Jonathan M. Lilly. Bivariate Empirical Mode Decomposition. 10 pages, 3 figures. Submitted to Signal Processing Letters, IEEE. Matlab/C codes and additional .. 2007. <ensl-00137611>

[1] Huang H. et al. 1998 ‘The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis.’ Procedings of the Royal Society 454, 903-995

[2] Zhao J., Huang D. 2001 ‘Mirror extending and circular spline function for empirical mode decomposition method’. Journal of Zhejiang University (Science) V.2, No.3, 247-252

[3] Gabriel Rilling, Patrick Flandrin, Paulo Gonçalves, June 2003: ‘On Empirical Mode Decomposition and its Algorithms’, IEEE-EURASIP Workshop on Nonlinear Signal and Image Processing NSIP-03

[4] Gabriel Rilling, Patrick Flandrin, Paulo Gonçalves, Jonathan M. Lilly. Bivariate Empirical Mode Decomposition. 10 pages, 3 figures. Submitted to Signal Processing Letters, IEEE. Matlab/C codes and additional .. 2007. <ensl-00137611>

Python Module Index

Index