BiSPy Documentation

Release 1.0

Julien Flamant

Jun 04, 2023

CONTENTS

BiSPy is an open-source python framework for processing bivariate signals. It supports our papers on time-frequency analysis [1], spectral analysis [2] and linear time-invariant filtering [3] of bivariate signals.

[1] Julien Flamant, Nicolas Le Bihan, Pierre Chainais: "Time-frequency analysis of bivariate signals", In press, Applied and Computational Harmonic Analysis, 2017; arXiv:1609.0246, doi:10.1016/j.acha.2017.05.007

[2] Julien Flamant, Nicolas Le Bihan, Pierre Chainais: "Spectral analysis of stationary random bivariate signals", 2017, IEEE Transactions on Signal Processing; arXiv:1703.06417, doi:10.1109/TSP.2017.2736494

[3] Julien Flamant, Pierre Chainais, Nicolas Le Bihan: "A complete framework for linear filtering of bivariate signals", 2018; Accepted for publication in IEEE Transactions on Signal Processing; arXiv:1802.02469

These papers contains theoretical results and several applications that can be reproduced with this toolbox.

This python toolbox is currently under development and is hosted on GitHub. If you encounter a bug or something unexpected please let me know by raising an issue on the project page.

CHAPTER

ONE

INSTALL FROM PYPI

Due to name conflict the available version on PyPi is named ``bispy-polar''. To install from PyPi, simply type

```
pip install bispy-polar
```

It will automatically install dependencies (see also below).

To get started, simply use

```
import bispy as bsp
```

CHAPTER

TWO

REQUIREMENTS

BiSPy works with python 3.5+.

Dependencies:

- NumPy
- SciPy
- Matplotlib
- numpy-quaternion

To install dependencies:

```
pip install numpy scipy matplotlib numpy-quaternion
```

numpy-quaternion add quaternion dtype support to numpy. Implementation by [moble]. Since this python toolbox relies extensively on this module, you can check out first the nice introduction here.

CHAPTER

THREE

LICENSE

This software is distributed under the CC-BY 4.0 license.

CHAPTER

FOUR

CITE THIS WORK

If you use this package for your own work, please consider citing it with this piece of BibTeX:

```
@misc{BiSPy,
    title =   {{BiSPy: an Open-Source Python project for processing bivariate signals}},
    author =  {Julien Flamant},
    year =    {2018},
    url =     {https://github.com/jflamant/bispy/},
    howpublished = {Online at: \url{github.com/jflamant/bispy/}},
    note =    {Code at https://github.com/jflamant/bispy/, documentation at https://
↪bispy.readthedocs.io/}
}
```

CHAPTER

FIVE

DOCUMENTATION CONTENTS

5.1 Tutorials

5.1.1 Time-Frequency-Polarization analysis: tutorial

This tutorial aims at demonstrating different tools available within the `timefrequency` module of `BiSPy`. The examples provided here come along with the paper

- Julien Flamant, Nicolas Le Bihan, Pierre Chainais: "Time-frequency analysis of bivariate signals", In press, Applied and Computational Harmonic Analysis, 2017; arXiv:1609.0246, doi:10.1016/j.acha.2017.05.007.

The paper contains theoretical results and several applications that can be reproduced with the following tutorial. A Jupyter notebook version can be downloaded here.

Load `bispy` and necessary modules

```
import numpy as np
import matplotlib.pyplot as plt
import quaternion  # load the quaternion module
import bispy as bsp
```

Quaternion Short-Term Fourier Transform (Q-STFT) example

To illustrate the behaviour of the Q-STFT, we construct a simple signal made of two linear chirps, each having its own instantaneous polarization properties.

First, define some constants:

```
N = 1024 # length of the signal

# linear chirps constants
a = 250*np.pi
b = 50*np.pi
c = 150*np.pi
```

Then define the instantaneous amplitudes, orientation, ellipticity and phase of each linear chirp. The amplitudes are taken equal - just a Hanning window.

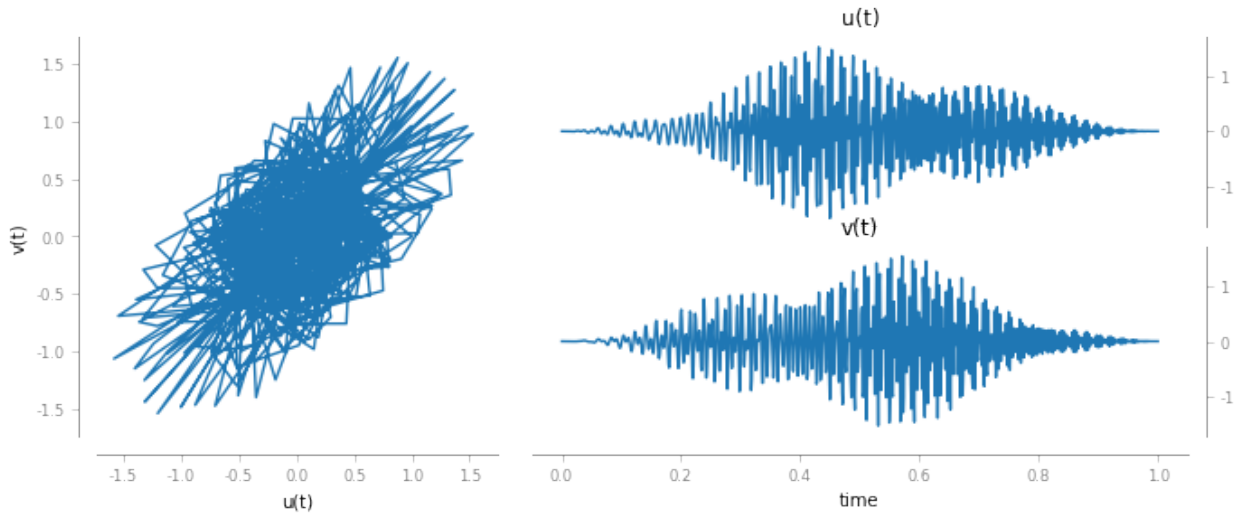
```
# time vector
t = np.linspace(0, 1, N)

# first chirp
theta1 = np.pi/4 # constant orientation
chi1 = np.pi/6-t # reversing ellipticity
phi1 = b*t+a*t**2 # linear chirp

# second chirp
theta2 = np.pi/4*10*t # rotating orientation
chi2 = 0 # constant null ellipticity
phi2 = c*t+a*t**2 # linear chirp

# common amplitude -- simply a window
env = bsp.utils.windows.hanning(N)
```

We can now construct the two components and sum it. To do so, we use the function `signals.bivariateAMFM` to compute directly the quaternion embeddings of each linear chirp.

```
# define chirps x1 and x2
x1 = bsp.signals.bivariateAMFM(env, theta1, chi1, phi1)
x2 = bsp.signals.bivariateAMFM(env, theta2, chi2, phi2)

# sum it
x = x1 + x2
```

Let us have a look at the signal `x[t]`

```
fig, ax = bsp.utils.visual.plot2D(t, x)
```



Now we can compute the Q-STFT. First initialize the object Q-STFT

```
S = bsp.timefrequency.QSTFT(x, t)
```

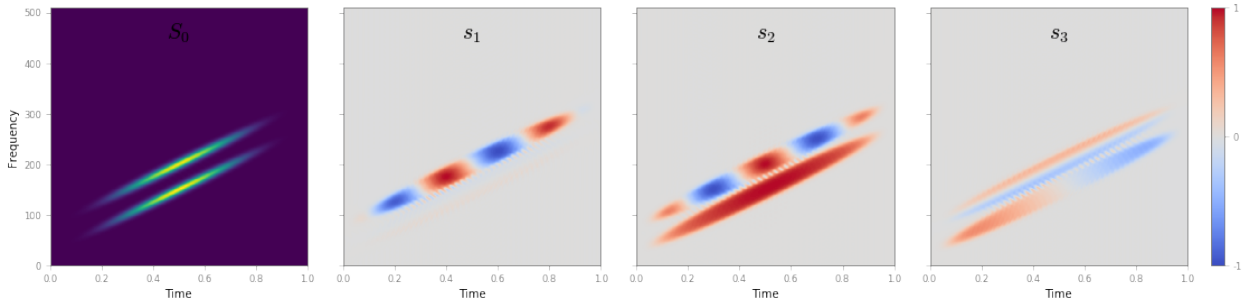
And compute:

```
S.compute(window='hamming', nperseg=101, noverlap=100, nfft=N)
```

```
Computing Time-Frequency Stokes parameters
```

Let us have a look at Time-Frequency Stokes parameters S1, S2 and S3

```
fig, ax = S.plotStokes()
```



Alternatively, we can compute the instantaneous polarization properties from the ridges of the Q-STFT.

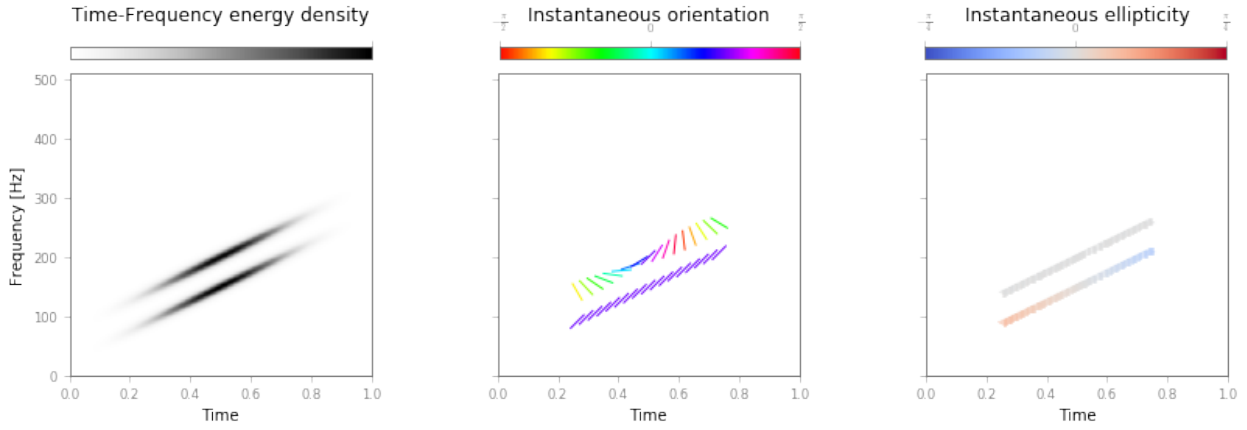
Extract the ridges:

```
S.extractRidges()
```

```
Extracting ridges
Ridge added
Ridge added
2 ridges were recovered.
```

And plot (`quivertdecim` controls the time-decimation of the quiver plot, for a cleaner view):

```
fig, ax = S.plotRidges(quivertdecim=30)
```



The two representations are equivalent and provide the same information: time, frequency and polarization properties of the bivariate signal. A direct inspection shows that instantaneous parameters of each components are recovered by both representations.

Quaternion Continuous Wavelet Transform (Q-CWT) example

The Q-STFT method has the same limitations as the usual STFT, that is not the ideal tool to analyze signals spanning a wide range of frequencies over short time scales. We revisit here the classic two chirps example in its bivariate (polarized) version.

As before, let us first define some constants:

```
N = 1024 # length of the signal

# hyperbolic chirps parameters
alpha = 15*np.pi
beta = 5*np.pi
tup = 0.8 # set blow-up time value
```

Now, let us define the instantaneous amplitudes, orientation, ellipticity and phase of each linear chirp. The chirps are also windowed.

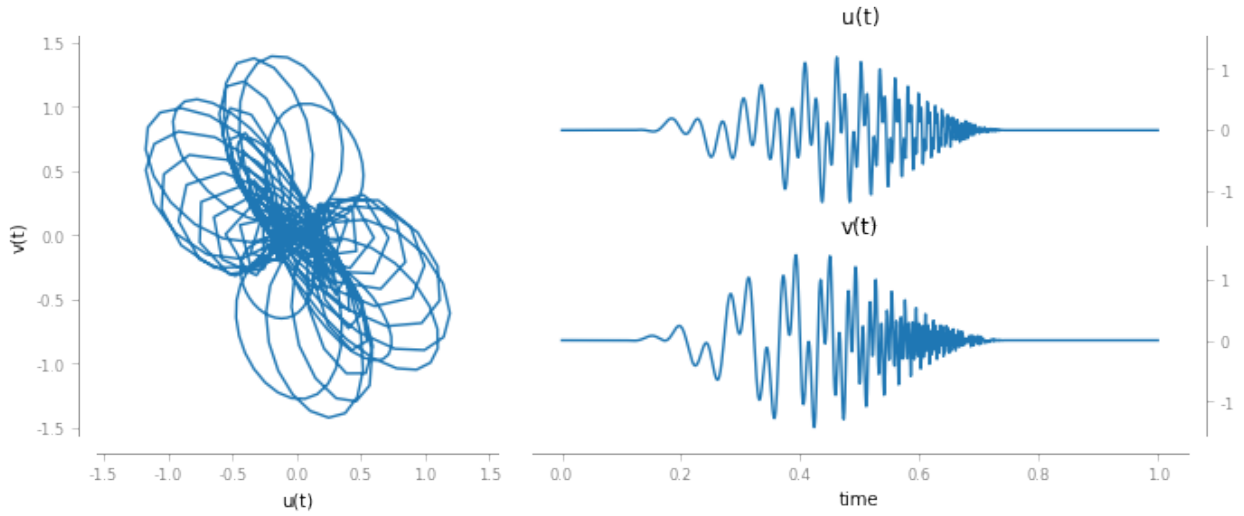
```
t = np.linspace(0, 1, N) # time vector

# chirp 1 parameters
theta1 = -np.pi/3 # constant orientation
chi1 = np.pi/6 # constant ellipticity
phi1 = alpha/(.8-t) # hyperbolic chirp

# chirp 2 parameters
theta2 = 5*t # rotating orientation
chi2 = -np.pi/10 # constant ellipticity
phi2 = beta/(.8-t) # hyperbolic chirp

# envelope
env = np.zeros(N)
Nmin = int(0.1*N) # minimum value of N such that x is nonzero
Nmax = int(0.75*N) # maximum value of N such that x is nonzero

env[Nmin:Nmax] = bsp.utils.windows.hanning(Nmax-Nmin)
```

Construct the two components and sum it. Again we use the function `utils.bivariateAMFM` to compute directly the quaternion embeddings of each linear chirp.

```
x1  = bsp.signals.bivariateAMFM(env, theta1, chi1, phi1)
x2  = bsp.signals.bivariateAMFM(env, theta2, chi2, phi2)

x = x1 + x2
```

Let us visualize the resulting signal, `x[t]`

```
fig, ax = bsp.utils.visual.plot2D(t, x)
```



Now, we can compute its Q-CWT. First define the wavelet parameters and initialize the QCWT object:

```
waveletParams = dict(type='Morse', beta=12, gamma=3)
S = bsp.timefrequency.QCWT(x, t)
```

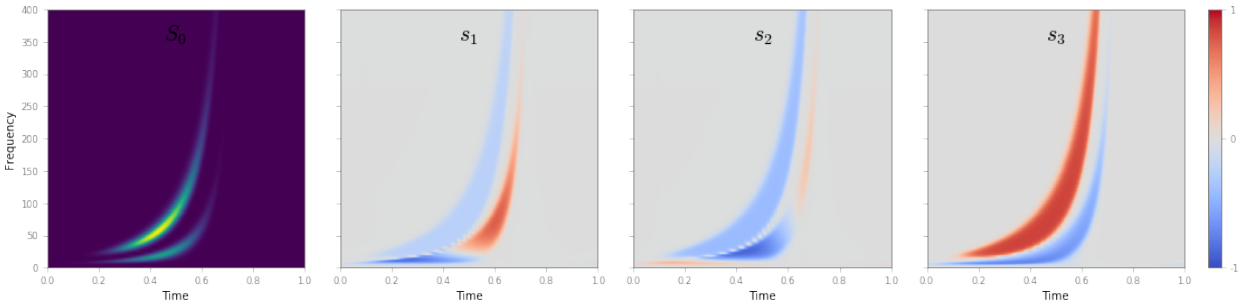
And compute:

```
fmin = 0.01
fmax = 400
S.compute(fmin, fmax, waveletParams, N)
```

```
Computing Time-Frequency Stokes parameters
```

Let us have a look at Time-Scale Stokes parameters S1, S2 and S3

```
fig, ax = S.plotStokes()
```



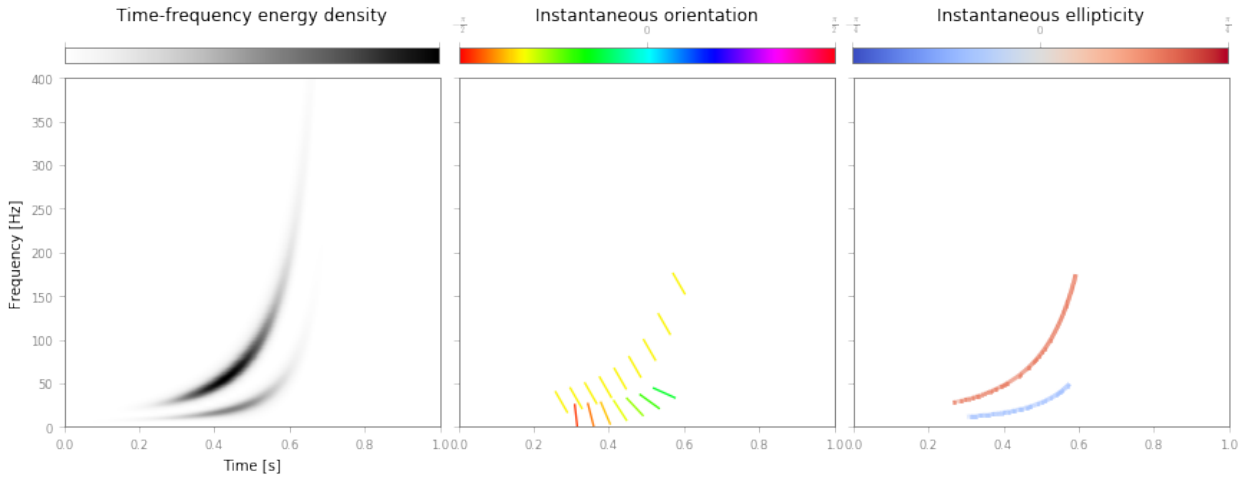
Similarly we can compute the instantaneous polarization attributes from the ridges of the Q-CWT.

```
S.extractRidges()
```

```
Extracting ridges
Ridge added
Ridge added
2 ridges were recovered.
```

And plot the results

```
fig, ax = S.plotRidges(quivertdecim=40)
```



Again, both representations are equivalent and provide the same information: time, scale and polarization properties of the bivariate signal. A direct inspection shows that instantaneous parameters of each components are recovered by both representations.

5.1.2 Spectral analysis of bivariate signals: tutorial

This tutorial aims at demonstrating different tools available within the `spectral` module of `BiSPy`. The examples provided here come along with the paper

- Julien Flamant, Nicolas Le Bihan, Pierre Chainais: "Spectral analysis of stationary random bivariate signals", IEEE Transactions on Signal Processing, 2017; arXiv:1703.06417, doi:10.1109/TSP.2017.2736494

The paper contains theoretical results and several applications that can be reproduced with the following tutorial. A completementary notebook version is available here.

Load `bispy` and necessary modules

```
import numpy as np
import matplotlib.pyplot as plt
import quaternion  # load the quaternion module
import bispy as bsp
```

Synthetic examples

The following examples are presented in the aforementioned paper. The module `bispy.signals` gives useful functions to generate the synthetic signals presented.

Example 1: Bivariate white noise only

First let us define the constants defining the polarization properties of the bivariate white gaussian noise.

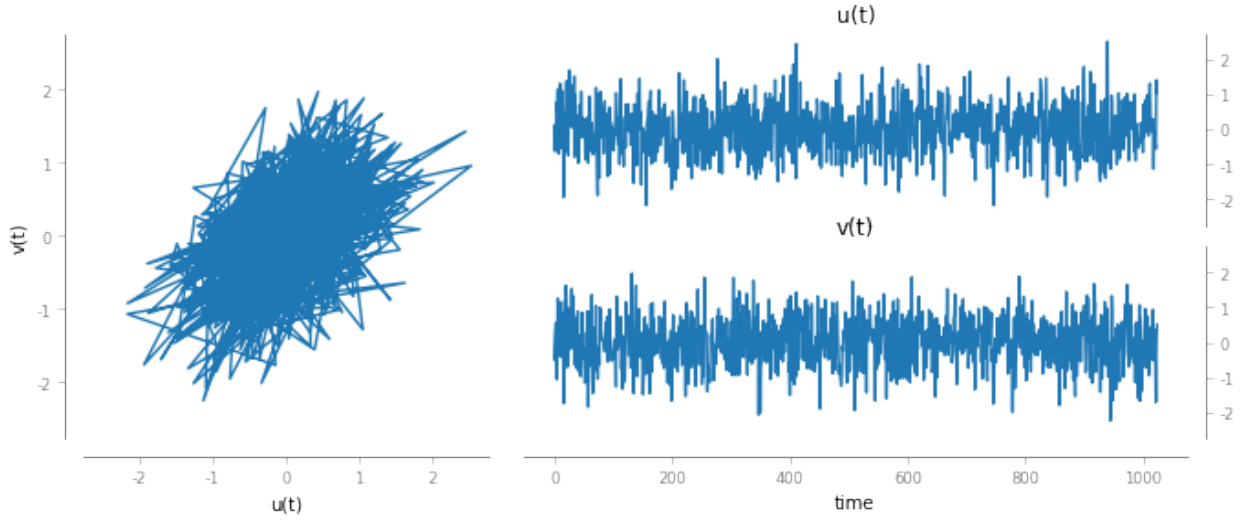
```
N = 1024 # length of the signal
S0 = 1 # power of the bivariate WGN
P0 = .5 # degree of polarization
theta0 = np.pi/4 # angle of linear polarization

t = np.arange(0, N) # time vector
```

First simulate a realization of this bivariate WGN:

```
w = bsp.signals.bivariatewhiteNoise(N, S0, P=P0, theta=theta0)
```

Now, display this signal

```
fig, ax = bsp.utils.visual.plot2D(t, w)
```



The goal is now to compare 2 spectral density estimation methods:

- an averaged polarization periodogram
- an averaged multitaper estimate using Slepian tapers.

To do so, we simulate `M` independent realization of this bivariate WGN, and average across realizations each method output.

```
M = 10 # number of independent realization of the WGN
```

The periodogram and multitaper estimates are computed like:

```
w = bsp.signals.bivariatewhiteNoise(N, S0, P=P0, theta=theta0)
# compute spectral estimates
per = bsp.spectral.Periodogram(t, w)
multi = bsp.spectral.Multitaper(t, w)

# loop accros realizations
```

(continues on next page)

(continued from previous page)

```
for k in range(1, M):
    w = bsp.signals.bivariatewhiteNoise(N, S0,  P=P0, theta=theta0)

    per2 = bsp.spectral.Periodogram(t, w)
    multi2 = bsp.spectral.Multitaper(t, w)
    per = per + per2
    multi = multi + multi2

# normalize by M
per = 1./M * per
multi = 1./M * multi
```

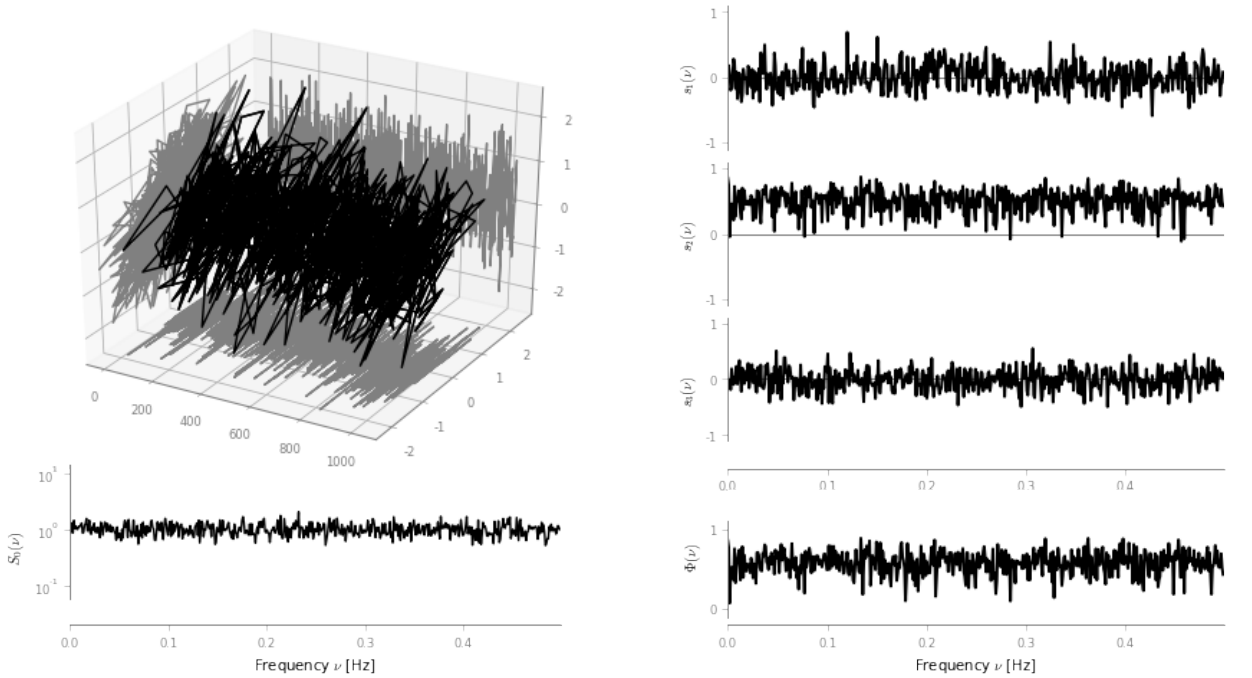
By default, the `Multitaper` class assumes a bandwidth `bw` of 2.5 frequency samples, giving 4 Slepian tapers.

The next step is to normalize the Stokes parameters S_1, S_2, S_3 by the intensity Stokes parameter S_0

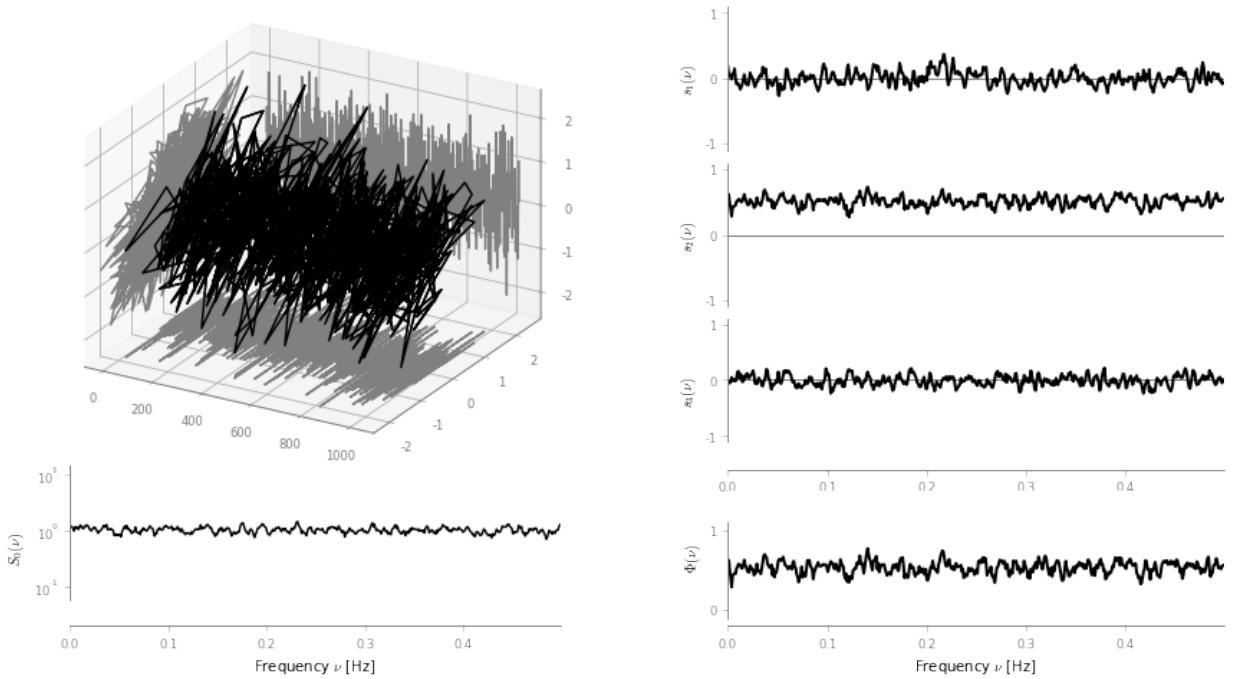
```
per.normalize()
multi.normalize()
```

We can now display the results for both methods

```
fig, axes = per.plot()
```




```
fig, ax = multi.plot()
```



Both estimates permit to recover the main features of the bivariate WGN: power, degree of polarization and polarization state are recovered.

Then the usual discussion between periodogram and multitaper estimates apply: the multitaper estimate exhibits reduced leakage bias and less variance than the periodogram estimate.

Example 2: bivariate monochromatic signal in white noise

We proceed similarly. First define the different parameters:

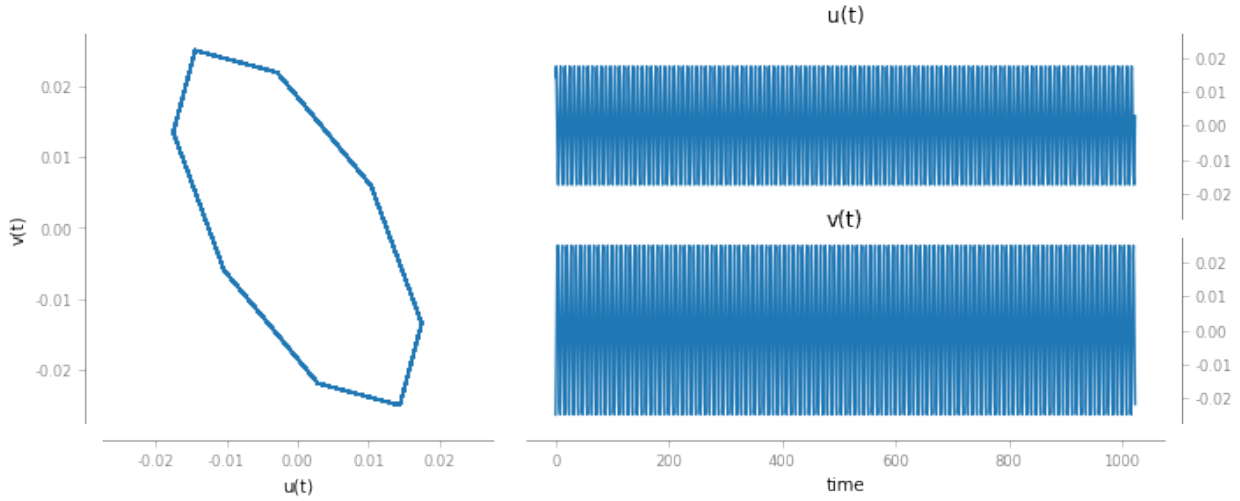
```
N = 1024 # length of the signal

t = np.arange(0, N) # time vector
dt = (t[1]-t[0])

# bivariate monochromatic signal parameters
a = 1/np.sqrt(N*dt) # amplitude = 1
theta = -np.pi/3 # polarization angle
chi = np.pi/8 # ellipticity parameter
f0 = 128/N/dt # frequency

# bivariate WGN noise paramerters
S0_w = 10**(-2) # power of the bivariate WGN
Phi_w = .2 # degree of polarization
theta_w = np.pi/8 # angle of linear polarization
```

Now, simulate a bivariate monochromatic signal (note the use of the argument `complexOutput` which provides a complex output (useful for plots), rather than a quaternion-valued output (useful for computations)

```
x = bsp.signals.bivariateAMFM(a, theta, chi, 2*np.pi*f0*t)
```

Let us have a look at the bivariate signal itself

```
fig, ax = bsp.utils.visual.plot2D(t, x)
```



Again, we compare 2 spectral density estimation methods:

- an averaged polarization periodogram
- an averaged multitaper estimate using Slepian tapers.

To do so, we simulate `M` independent realization of this bivariate WGN, and average across realizations each method output.

```
M = 20 # number of realizations
y = np.zeros((N, M), dtype='quaternion')

# generate the data
for k in range(M):
    phi = 2*np.pi*np.random.rand() # random initial phase term
    x = bsp.signals.bivariateAMFM(a, theta, chi, 2*np.pi*f0*t+phi) # bivariate␣
↪monochromatic signal
    w = bsp.signals.bivariatewhiteNoise(N, S0_w, Phi_w, theta_w) # bivariate WGN
    y[:, k] = x + w

# compute spectral estimates
per = bsp.spectral.Periodogram(t, y[:, 0])
multi = bsp.spectral.Multitaper(t, y[:, 0], bw=3)
for k in range(1, M):
    per2 = bsp.spectral.Periodogram(t, y[:, k])
    multi2 = bsp.spectral.Multitaper(t, y[:, k], bw=3)

    per = per + per2
    multi = multi + multi2


per = 1./M * per
multi = 1/M * multi
```

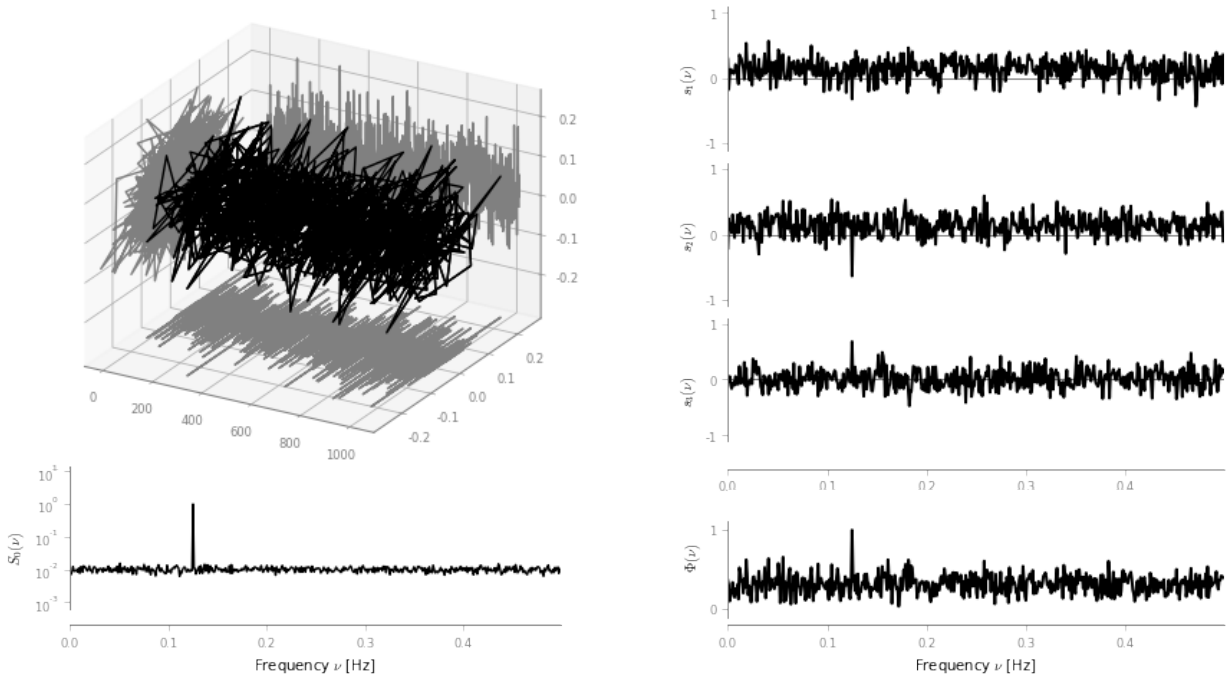
Here the multitaper class is computed with a bandwidth `bw = 3` frequency samples, giving 5 Slepian tapers.

The next step is to normalize the Stokes parameters S_1, S_2, S_3 by the intensity Stokes parameter S_0

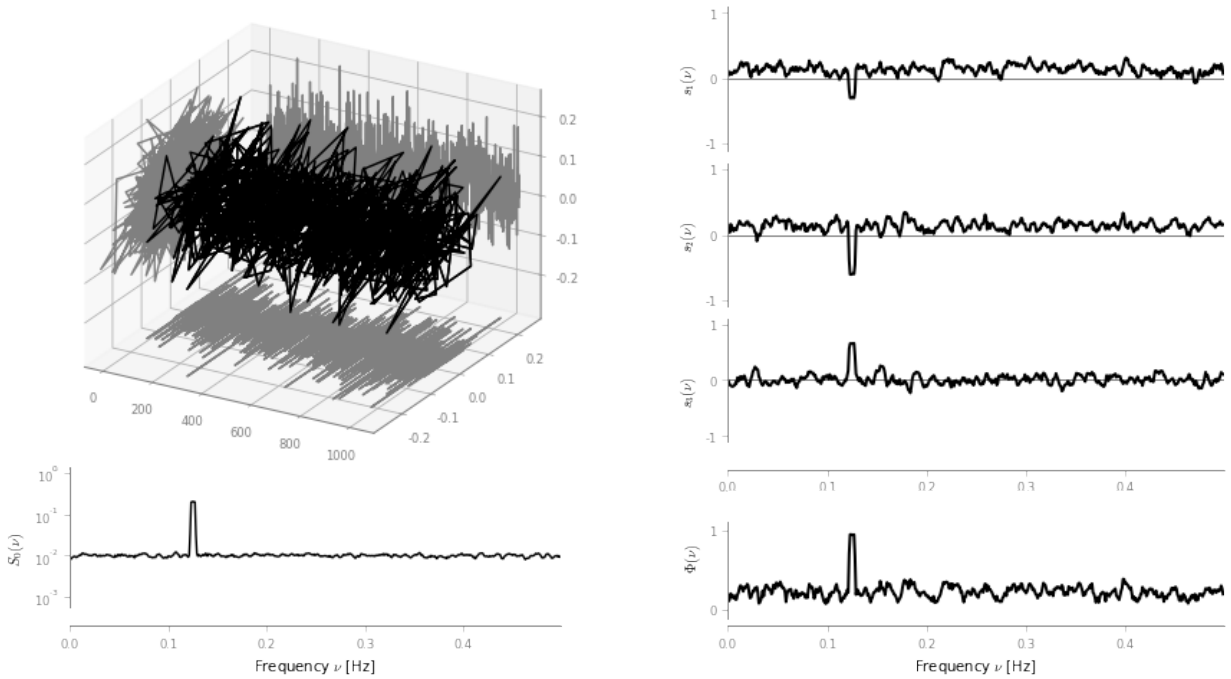
```
per.normalize()
multi.normalize()
```

We can now display the results for both methods

```
fig, ax = per.plot()
```




```
fig, ax = multi.plot()
```



A real case example: spectral analysis of wind measurements

We turn to a real-life example to illustrate the general relevance of the method.

We consider a dataset of instantaneous wind measurements (east and northward velocities). The dataset is available for download at http://www.commsp.ee.ic.ac.uk/~mandic/research/WL_Complex_Stuff.htm. This dataset has been used by the authors in several publications, e.g. in

```
S. L. Goh, M. Chen, D. H. Popovic, K. Aihara, D. Obradovic and D. P. Mandic, "Complex-
↪Valued Forecasting of Wind Profile," Renewable Energy, vol. 31, pp. 1733-1750, 2006.
```

Quoting the included Readme: >- Wind data for 'low', 'medium' and 'high' dynamics regions. - Data are recorded using the Gill Instruments WindMaster, the 2D ultrasonic anemometer - Wind was sampled at 32 Hz and resampled at 50Hz, and the two channels correspond to the the "north" and "east" direction - To make a complex-valued wind signal, combine z=v_n + j v_e, where 'v' is wind speed and 'n' and 'e' the north and east directions - Data length = 5000 samples

Setting 1: low-wind

We start by loading the data

```
import scipy.io as scio
windData = scio.loadmat('datasets/wind/low-wind.mat')

u = windData['v_east'][:,0]
v = windData['v_north'][:, 0]

N = np.size(u) # should be 5000
dt = 1./50
```

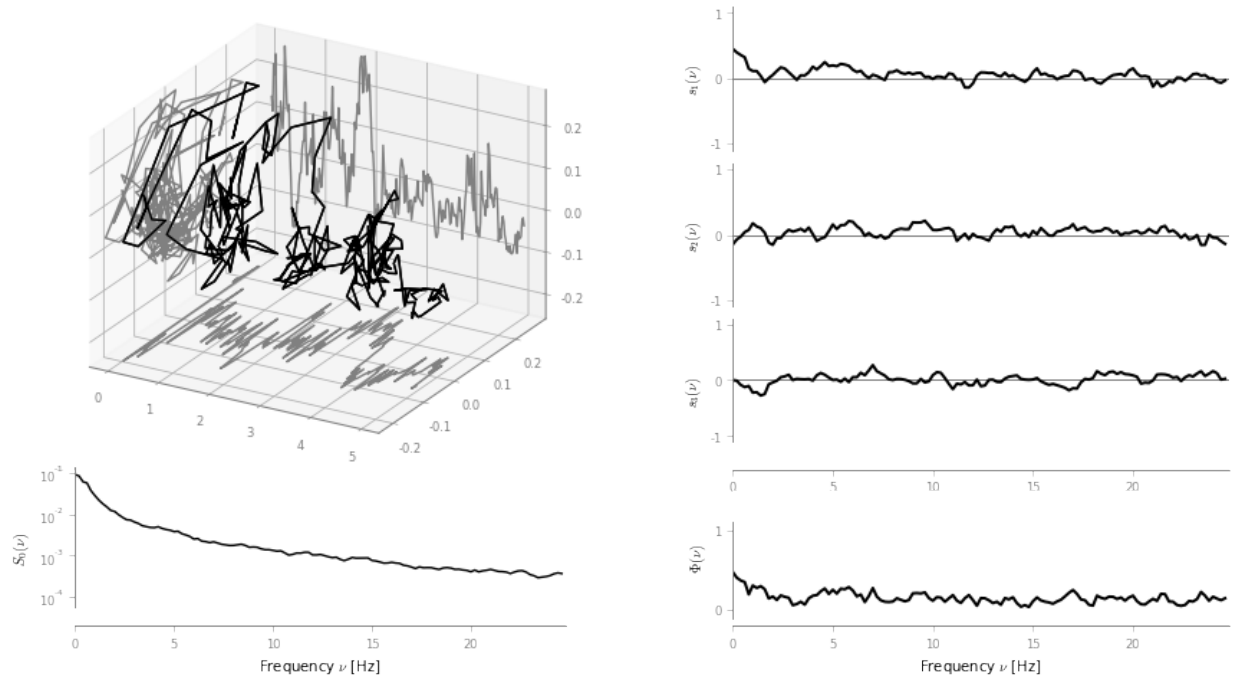
Estimating polarization features in bivariate signals requires ideally multiple measurements/realizations. We will fake this out using an ergodic hypothesis. This thus split the signal into `Nw` subsignals, and compute for each a spectral estimate. By averaging out spectral estimates, one obtains a estimate of the spectral density of the underlying process. (Welch method with no overlap)

Let's define a handy function:

```
def subsignal(u, v, Nx, k):
    '"subsamples u, v components and returns the associated quaternion signal"'
    uk = u[k*Nx:(k+1)*Nx]
    vk = v[k*Nx:(k+1)*Nx]

    # to make it zero-mean
    uk = uk - np.mean(uk)
    vk = vk - np.mean(vk)

    return bsp.utils.sympSynth(uk, vk)
```

Then we compute the averaged multitaper estimate

```
# subsampling parameters
Nw = 20 # number of subsamples
Nx = N // Nw # length of one subsampled signal

# time index for subsampled signals
tx = np.arange(Nx)*dt

xk = subsignal(u, v, Nx, 0)

multi = bsp.spectral.Multitaper(tx, xk)
# loop across subsamples
for k in range(1, Nw):

    xk = subsignal(u, v, Nx, k)
    multi2 = bsp.spectral.Multitaper(tx, xk)
    multi = multi + multi2

# normalize and plot multitaper estimate
multi.normalize()
fig, ax = multi.plot()
```



The total power spectrum $S_0(\nu)$ exhibits a power-law like shape.

Looking at the degree of polarization $\Phi(\nu)$, we see that the signal is almost unpolarized at all frequencies, except for frequencies below 0.5 Hz, where we notice a small increase in the degree of polarization.

Setting 2: moderate wind

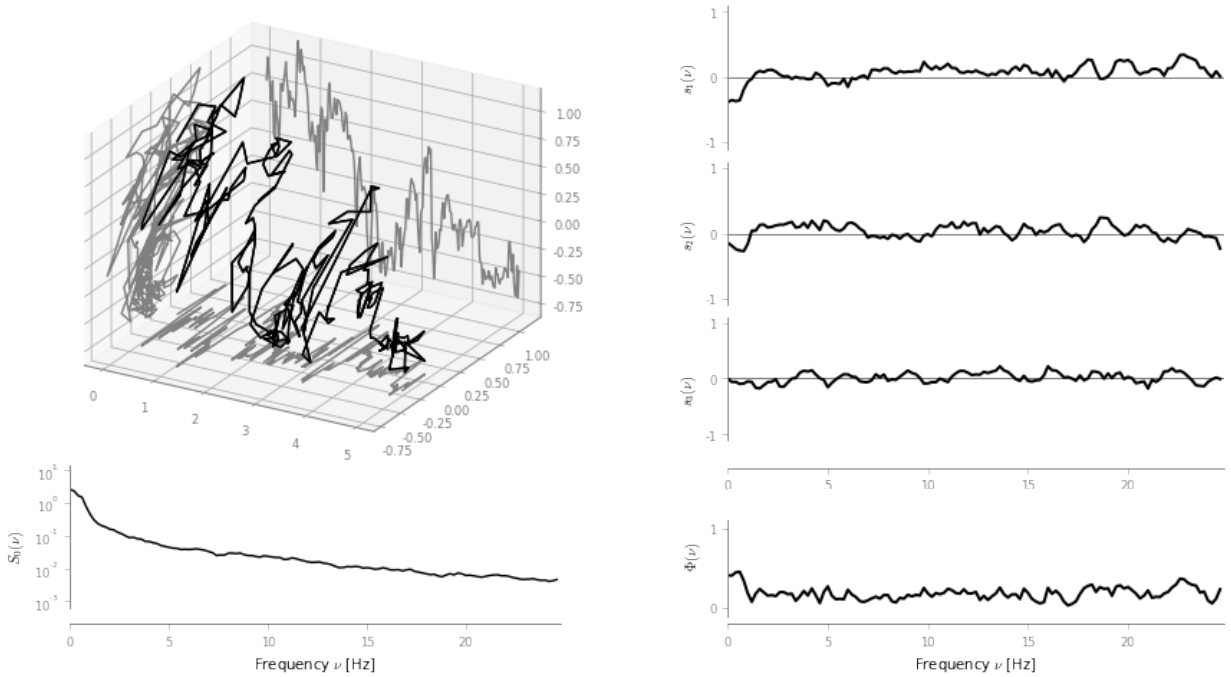
We follow the same procedure as above.

```
# load data
windData = scio.loadmat('datasets/wind/medium-wind.mat')

u = windData['v_east'][:,0]
v = windData['v_north'][:, 0]

N = np.size(u)

# we use an ergodic argument and split the signal into "sub-signals"
Nw = 20
Nx = N // Nw
tx = np.arange(Nx)*dt

xk = subsignal(u, v, Nx, 0)

# compute spectral estimate
multi = bsp.spectral.Multitaper(tx, xk)
for k in range(1, Nw):

    xk = subsignal(u, v, Nx, k)
    multi2 = bsp.spectral.Multitaper(tx, xk)
```

(continues on next page)

(continued from previous page)

```
    multi = multi + multi2

# normalize and plot multitaper estimate
multi.normalize()
fig, ax = multi.plot()
```



We observe again power law - like shape in the total power $S_0(\nu)$. The degree of polarization $\Phi(\nu)$ is close to zero for frequencies above 1 Hz; There is again a small "step" for frequencies below 1 Hz.

Setting 3: high-wind

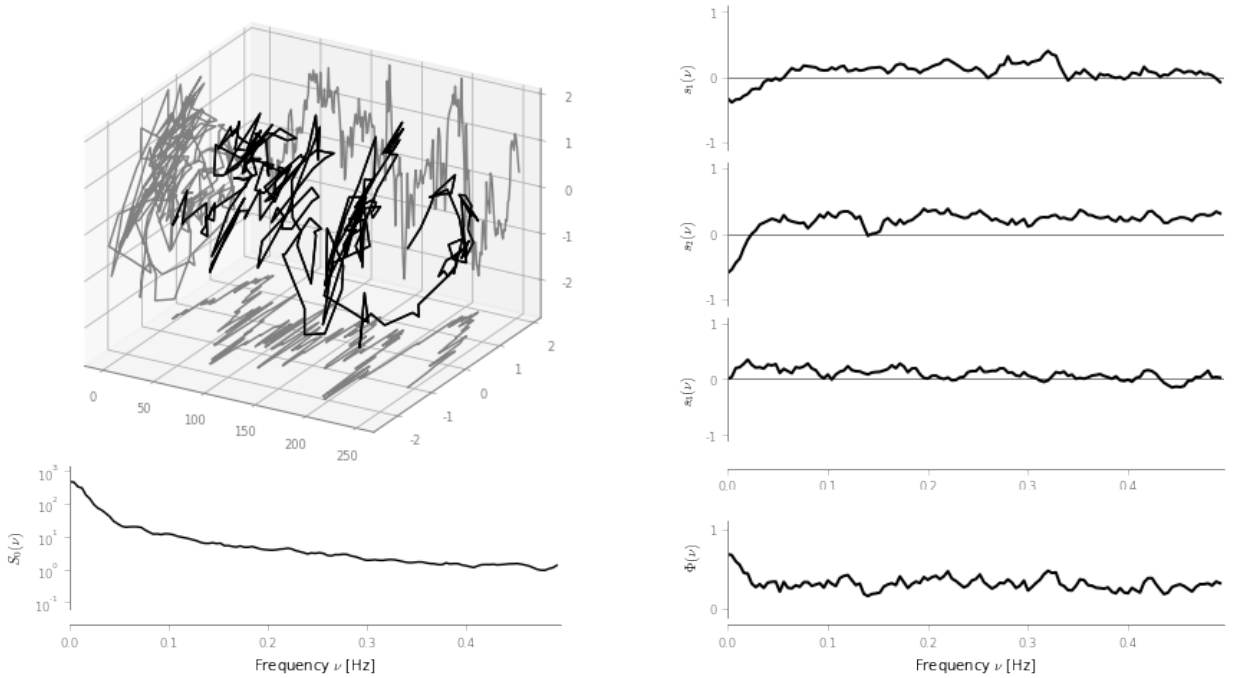
Again, same procedure.

```
# load data
windData = scio.loadmat('datasets/wind/high-wind.mat')

u = windData['v_east'][:,0]
v = windData['v_north'][:, 0]

N = np.size(u)

# we use an ergodic argument and split the signal into "sub-signals"
Nw = 20
Nx = N // Nw
tx = np.arange(Nx)

xk = subsignal(u, v, Nx, 0)
```

(continues on next page)

(continued from previous page)

```
# compute spectral estimate
multi = bsp.spectral.Multitaper(tx, xk)
for k in range(1, Nw):

    xk = subsignal(u, v, Nx, k)
    multi2 = bsp.spectral.Multitaper(tx, xk)

    multi = multi + multi2
# normalize and plot multitaper estimate
multi.normalize()
fig, ax = multi.plot()
```



Again $S_0(\nu)$ exhibits a power law shape. The degree of polarization is overall higher than in the low and moderate wind settings. The signal is strongly polarized ($\Phi(0) \simeq 0.7$) at low frequencies. High frequencies show a relatively constant degree of polarization, around $\Phi(\nu) \simeq 0.3$

5.2 Reference manual

5.2.1 `qfft` - Quaternion Fourier Transforms

General

Qfft(x, **kwargs)	Performs QFT using 2 ffts.
iQfft(X, **kwargs)	Performs inverse QFT.
Qfftshift(X)	Shifts the QFT array
iQfftshift(X)	Unshifts the QFFT array
Qfftfreq(N[, dt])	Return the sampled frequencies, from time spacing dt.

bispy.qfft.Qfft

bispy.qfft.**Qfft**(*x*, ***kwargs*)

Performs QFT using 2 ffts.

Parameters

x
[array_type]

Returns

X
[array_type]

bispy.qfft.iQfft

bispy.qfft.**iQfft**(*X*, ***kwargs*)

Performs inverse QFT.

Parameters

X
[array_type]

Returns

x
[array_type]

bispy.qfft.Qfftshift

bispy.qfft.**Qfftshift**(*X*)

Shifts the QFT array

Parameters

X
[array_type]

Returns

Xshifted
[array_type]

bispy.qfft.iQfftshift

bispy.qfft.**iQfftshift**(*X*)

Unshifts the QFFT array

Parameters

X
[array_type]

Returns

Xunshifted
[array_type]

bispy.qfft.Qfftfreq

bispy.qfft.**Qfftfreq**(*N*, *dt=1.0*)

Return the sampled frequencies, from time spacing dt.

See numpy.fft.fftfreq for further reference.

Parameters

N
[int] length of the signal

dt
[float, optional] time sampling step. Default 1.0

Returns

f
[array_type] sampled frequencies

5.2.2 utils - Utility functions

Quaternion specific operations

sympSplit(q)	Splits a quaternion array into two complex arrays.
sympSynth(q_1, q_2)	Constructs a quaternion array from two complex arrays.
StokesNorm(q)	Return the Stokes-Poincaré norm of a quaternion.
normalizeStokes(S0, S1, S2, S3[, tol])	Normalize Stokes parameters S1, S2, S3 by S0.
Stokes2geo(S0, S1, S2, S3[, tol])	Return geometric parameters from Stokes parameters.
geo2Stokes(a, theta, chi[, Phi])	Compute Stokes parameters from geometric parameters.
quat2euler(q)	Euler polar form of a quaternion array.
euler2quat(a, theta, chi, phi)	Quaternion from Euler polar form.

bispy.utils.sympSplit

bispy.utils.**sympSplit**(*q*)

Splits a quaternion array into two complex arrays.

The decomposition reads:

```
q = q_1 + i q_2
```

where q_1, q_2 are complex (1, 1j) numpy arrays

Parameters

q
[quaternion numpy array]

Returns

q_1, q_2
[complex numpy arrays]

See also:

sympSynth

Examples

```
>>> q
array([[quaternion(0.3, 0.47, -0.86, -0.42),
        quaternion(0.24, -1.07, -2.11, 0.37),
        quaternion(-0.24, -1.36, -1.14, 1.69)],
       [quaternion(0.4, -0.61, 0.04, -0.03),
        quaternion(-1.58, -1.69, -1.18, -1.02),
        quaternion(0.78, -1.06, -1.05, -0.62)]], dtype=quaternion)
>>> q_1, q_2 = sympSplit(q)
>>> q_1
array([[ 0.30-0.86j,  0.24-2.11j, -0.24-1.14j],
   [ 0.40+0.04j, -1.58-1.18j,  0.78-1.05j]])
>>> q_2
array([[ 0.47-0.42j, -1.07+0.37j, -1.36+1.69j],
   [-0.61-0.03j, -1.69-1.02j, -1.06-0.62j]])
```

bispy.utils.sympSynth

bispy.utils.**sympSynth**(*q_1*, *q_2*)

Constructs a quaternion array from two complex arrays.

The decomposition reads:

```
q = q_1 + i q_2
```

where q_1, q_2 are complex (1, 1j) numpy arrays

Parameters

q_1, q_2
[complex numpy arrays]

Returns

q
[quaternion numpy array]

See also:

sympSplit

Examples

```
>>> q_1
array([[ 0.30-0.86j,  0.24-2.11j, -0.24-1.14j],
   [ 0.40+0.04j, -1.58-1.18j,  0.78-1.05j]])
>>> q_2
array([[ 0.47-0.42j, -1.07+0.37j, -1.36+1.69j],
   [-0.61-0.03j, -1.69-1.02j, -1.06-0.62j]])
>>> sympSynth(q_1 q_2)
array([[quaternion(0.3, 0.47, -0.86, -0.42),
        quaternion(0.24, -1.07, -2.11, 0.37),
        quaternion(-0.24, -1.36, -1.14, 1.69)],
       [quaternion(0.4, -0.61, 0.04, -0.03),
        quaternion(-1.58, -1.69, -1.18, -1.02),
        quaternion(0.78, -1.06, -1.05, -0.62)]], dtype=quaternion)
```

bispy.utils.StokesNorm

bispy.utils.**StokesNorm**(*q*)

Return the Stokes-Poincaré norm of a quaternion.

The Stokes-Poincaré norm is defined by:

```
StokesNorm(q) = -q*j*np.conj(q)
```

with j = quaternion(0, 0, 1, 0).

Parameters

q
[quaternion numpy array]

Returns

q*j*np.conj(q)
[Stokes-Poincaré norm of q]

See also:

quat2euler

bispy.utils.normalizeStokes

bispy.utils.**normalizeStokes**(*S0*, *S1*, *S2*, *S3*, *tol=0.0*)

Normalize Stokes parameters S1, S2, S3 by S0.

Normalization can be performed using a soft thresholding-like method, if regularization is needed:

```
Si = Si/(S0 + tol*np.max(S0))
```

where i = 1, 2, 3 and *tol* is the tolerance factor. This function assumes that the maximum value of S0 has a significance for the whole indices of the Si arrays.

Parameters

S0, S1, S2, S3
[array_type]

tol
[float, optional]

Returns

S1n, S2n, S3n
[array_type]

See also:

quat2euler

bispy.utils.Stokes2geo

bispy.utils.**Stokes2geo**(*S0*, *S1*, *S2*, *S3*, *tol=0.0*)

Return geometric parameters from Stokes parameters.

It returns the decomposition in a, theta, chi and degree of polarization Phi.

Parameters

S0, S1, S2, S3
[array_type]

tol
[float, optional]

Returns

a, theta, chi, Phi
[array_type]

See also:

quat2euler
normalizeStokes
geo2Stokes

bispy.utils.geo2Stokes

bispy.utils.**geo2Stokes**(*a*, *theta*, *chi*, *Phi=1*)

Compute Stokes parameters from geometric parameters.

Parameters

a, theta, chi
[array_type]

Phi
[array_type, optional]

Returns

S0, S1, S2, S3
[array_type]

See also:

quat2euler
Stokes2geo

bispy.utils.quat2euler

bispy.utils.**quat2euler**(*q*)

Euler polar form of a quaternion array.

The decomposition reads:

```
q = a * np.exp(i * theta) * np.exp(-k * chi) * np.exp(j * phi)
```

with a > 0, -pi/2 < theta < pi/2, -pi/4 < chi < pi/4 and -pi < phi < pi .

Parameters

q
[quaternion numpy array]

Returns

a, theta, chi, phi
[array_type]

See also:

euler2quat

bispy.utils.euler2quat

bispy.utils.**euler2quat**(*a*, *theta*, *chi*, *phi*)

Quaternion from Euler polar form.

The decomposition reads:

```
q = a * np.exp(i * theta) * np.exp(-k * chi) * np.exp(j * phi)
```

with a > 0, -pi/2 < theta < pi/2, -pi/4 < chi < pi/4 and -pi < phi < pi .

Parameters

a, theta, chi, phi
[array_type]

Returns

q
[quaternion numpy array]

See also:

quat2euler

Windows functions

class bispy.utils.**windows**

Windows functions static methods.

These window functions are provided for convenience, and are meant to be used with the QSTFT class.

Methods

gaussian(N[, sigma])	Gaussian window
hamming(N)	Hamming window
hanning(N)	Hanning window
rectangle(N)	Rectangle window

Graphical tools

class bispy.utils.**visual**

Static methods for visualization of bivariate signals.

Methods

plot2D(t, q[, labels])	2D plot of a bivariate signal.
plot3D(t, q)	3D plot of a bivariate signal

static plot2D(*t*, *q*, *labels=['u(t)', 'v(t)']*)

2D plot of a bivariate signal.

Plots the 2D trace, and time evolution of each component.

Parameters

t, q
[array_type] time and signal arrays (signal array may be either complex or quaternion type)

labels
[[label1, label2]] list of labels to display.

Returns

fig, ax
[figure and axis handles]

static plot3D(*t*, *q*)

3D plot of a bivariate signal

Parameters

t, q
[array_type] time and signal arrays (signal array may be either complex or quaternion type)

Returns

fig, ax
[figure and axis handles]

5.2.3 `timefrequency` - Time-Frequency representations

Quaternion Embedding

class bispy.timefrequency.**Hembedding**(*q*)

H-embedding class. Computes quaternion embedding of complex-valued signals.

Parameters

q
[array_type] quaternion input signal

Attributes

signal
[array_type] original input signal

Hembedding
[array_type] Quaternion-embedding of the input signal

a
[array_type] instantaneous amplitude

theta
[array_type] instantaneous orientation

chi
[array_type] instantaneous ellipticity

phi
[array_type] instantaneous phase

Q-STFT

class bispy.timefrequency.**QSTFT**(*x*, *t=None*)

Compute the Quaternion-Short Term Fourier Transform for bivariate signals taken as (1, i)-quaternion valued signals.

Parameters

x
[array_type] input signal array

t
[array_type (optional)] time samples array. Default is t = np.arange(x.shape[0])

Attributes

t
[array_type] time samples array

x
[array_type] input signal array

params
[dict] parameters used for the computation of the Q-STFT.

sampled_time
[array_type] sampled times instants

f
[array_type] sampled frequencies

tfpr
[array_type] Q-STFT coefficients array

S0, S1, S2, S3
[array_type] Time-frequency Stokes parameters, non-normalized [w.r.t. S0]

S1n, S2n, S3n
[array_type] normalized time-frequency Stokes parameters [w.r.t. S0] using the tolerance factor *tol*. See *utils.normalizeStokes*.

ridges
[list] List of ridges index and values extracted from the time-frequency energy density S0. Requires call of *extractRidges* for ridges to be added.

Methods

compute([window, nperseg, noverlap, nfft, ...])	Compute the Q-STFT of the signal x.
extractRidges([parThresh, parMinD])	Extracts ridges from the time-frequency energy density S0.
inverse([mask])	Compute inverse Q-STFT
normalizeStokes([tol])	Re-compute normalized Stokes parameters with a different normalization.
plotRidges([quivertdecim])	Plot S0, the orientation and ellipticity recovered from the ridges in time-frequency domain
plotSignal([kind])	Plot the bivariate signal x.
plotStokes([S0_cmap, s_cmap, single_sided])	Time-frequency plot of time-frequency energy map (S0) and time-frequency polarization parameters (normalized Stokes parameters S1n, S2n, S3n)

compute(*window='hamming'*, *nperseg=128*, *noverlap=None*, *nfft=None*, *boundary='zeros'*, *tol=0.01*, *ridges=False*)

Compute the Q-STFT of the signal x.

It takes advantages of the scipy.signal.stft function for greater flexibility.

Parameters

window, nperseg, noverlap, nfft, boundary
[stft parameters] See *scipy.signal.stft*

tol
[float, optional] tolerance factor used in normalization of Stokes parameters. Default to 0.01

ridges: bool, optional
If True, compute also the ridges of the transform. Default to *False*. Ridges can be later computed using *extractRidges()*.

extractRidges(*parThresh=4*, *parMinD=3*)

Extracts ridges from the time-frequency energy density S0.

Parameters

parThresh
[float, optional] Controls the threshold at which local maxima of S0 are accepted or rejected. Larger values of `parThresh`increase the number of eligible points.

parMinD
[float, optional] Ridge smoothness parameter. Controls at which maximal distance can be located two eligible same ridge points. The smaller `parMinD`is the smoother ridges are.

Returns

ridges
[list] list of detected ridges

`inverse`(*mask=None*)

Compute inverse Q-STFT

Parameters

mask: array_type
mask applied to Q-STFT coefficients prior to inversion. If mask=None, no mask is employed.

`plotRidges`(*quivertdecim=10*)

Plot S0, the orientation and ellipticity recovered from the ridges in time-frequency domain

If ridges are not extracted yet, it runs *extractRidges* method first.

Parameters

quivertdecim
[int, optional] time-decimation index (allows faster and cleaner visualization of orientation vector field)

Returns

fig, ax
[figure and axis handles] may be needed to tweak the plot

`plotStokes`(*S0_cmap='viridis'*, *s_cmap='coolwarm'*, *single_sided=True*)

Time-frequency plot of time-frequency energy map (S0) and time-frequency polarization parameters (normalized Stokes parameters S1n, S2n, S3n)

Parameters

S0_cmap
[colormap (sequential)] to use for S0 time-frequency distribution

s_cmap
[colormap (diverging)] to use for normalized Stokes time-frequency distribution

Returns

fig, ax
[figure and axis handles] may be needed to tweak the plot

Q-CWT

`class bispy.timefrequency.QCWT`(*x*, *t=None*)

Methods

compute(fmin, fmax, waveletParams[, ...])	Compute the Q-CWT of x using a specified wavelet.
extractRidges([parThresh, parMinD])	Extracts ridges from the time-scale energy density S0.
normalizeStokes([tol])	Re-compute normalized Stokes parameters with a different normalization.
plotRidges([quivertdecim])	Plot S0, and the orientation and ellipticity recovered from the ridges in time-scale domain
plotSignal([kind])	Plot the bivariate signal x.
plotStokes([S0_cmap, s_cmap])	Time-frequency plot of time-frequency energy map (S0) and time-frequency polarization parameters (normalized Stokes parameters S1n, S2n, S3n)

compute(*fmin*, *fmax*, *waveletParams*, *Nscales=50*, *tol=0.01*, *ridges=False*)

Compute the Q-CWT of x using a specified wavelet.

Parameters

fmin, fmax
[float] min and max frequencies

waveletParams
[dict] dictionary containing wavelet features. Currently 2 types, 'Morlet' and 'Morse' are supported.

Nscales
[int] number of scales to analyze. Controls the size of the sampled_frequencies array.

tol
[float, optional] tolerance factor used in normalization of Stokes parameters. Default to 0.01

ridges: bool, optional
If True, compute also the ridges of the transform. Default to *False*. Ridges can be later computed using *extractRidges()*.

extractRidges(*parThresh=4*, *parMinD=3*)

Extracts ridges from the time-scale energy density S0.

Parameters

parThresh
[float, optional] Controls the threshold at which local maxima of S0 are accepted or rejected. Larger values of `parThresh`increase the number of eligible points.

parMinD
[float, optional] Ridge smoothness parameter. Controls at which maximal distance can be located two eligible same ridge points. The smaller `parMinD`is the smoother ridges are.

Returns

ridges
[list] list of detected ridges

plotRidges(*quivertdecim=10*)

Plot S0, and the orientation and ellipticity recovered from the ridges in time-scale domain

If ridges are not extracted yet, it runs *extractRidges* method first.

Parameters

quivertdecim
[int, optional] time-decimation index (allows faster and cleaner visualization of orientation vector field)

Returns

fig, ax
[figure and axis handles] may be needed to tweak the plot

`plotStokes`(*S0_cmap='viridis'*, *s_cmap='coolwarm'*)

Time-frequency plot of time-frequency energy map (S0) and time-frequency polarization parameters (normalized Stokes parameters S1n, S2n, S3n)

Parameters

S0_cmap
[colormap (sequential)] to use for S0 time-frequency distribution

s_cmap
[colormap (diverging)] to use for normalized Stokes time-frequency distribution

Returns

fig, ax
[figure and axis handles] may be needed to tweak the plot

5.2.4 `spectral` - Spectral analysis routines

Quaternion PSD

`class` bispy.spectral.`quaternionPSD`(*N*, *S0x*, *Phix*, *mux*, *dt=1*)

Quaternion Power Spectral Density constructor of the form: Gamma_xx(nu) = S_0x(nu)*[1 + Phi_x(nu)*mu_x(nu)]

where S_0x is the PSD of the signal x, Phi_x(nu) is the degree of polarization of the signal and mu_x is the polarization axis.

Parameters

N
[int] size of the frequencies array

S0x
[array_type] PSD array of the signal

Phix
[array_type] degree of polarization array

mux
[array_type (quaternion)] polarization axis array

dt
[float (optional)] time sampling size step

Attributes

S0, S1, S2, S3
[array_type] Stokes Parameters array

density
[array_type] quaternion PSD

f
[array_type] sampled frequencies array

Phi
[array_type] degree of polarization

mu
[array_type] polarization axis

Methods

plot([single_sided])	Displays the quaternion PSD
plotStokes([single_sided])	Displays Stokes Parameters S0, S1, S2, S3

plot(*single_sided=True*)
Displays the quaternion PSD

plotStokes(*single_sided=True*)
Displays Stokes Parameters S0, S1, S2, S3

Periodogram

class bispy.spectral.**Periodogram**(*t*, *x*, *computeFlag=True*)
Compute the periodogram of bivariate signals taken as (1, i)-quaternion valued signals.

Parameters

t
[array_type] time samples array

x
[array_type] input signal array (has to be of quaternion dtype)

compute
[bool, optional] Flag activating computation of the estimate. Default is true. If False one has to run the compute() method manually.

Attributes

t
[array_type] time samples array

signal
[array_type] input signal array

f
[array_type] sampled frequencies array

density
[array_type] spectral density quaternion array

S0, S1, S2, S3
[array_type] Stokes parameters, non-normalized [w.r.t. S0]

S1n, S2n, S3n
[array_type] normalized Stokes parameters [w.r.t. S0] using the tolerance factor *tol*. They are not computed by default. See *normalize*.

Phi
[array_type] Degree of polarization. Not computed by default; See *normalize*.

Methods

compute()	Low-level function.
normalize([tol])	Normalize Stokes parameters wrt S0.
plot()	Generic plot of spectral estimates

compute()
Low-level function. Compute Periodogram estimate

normalize(*tol=0.0*)
Normalize Stokes parameters wrt S0. In addition, compute the degree of polarization Phi.

Parameters

tol
[float, optional] tolerance factor used in Stokes parameters normalization. Default is 0.0

Returns

self.S1n, self.S2n, self.S3n
[array_type] normalized Stokes parameters

self.Phi
[array_type] degree of polarization

See also:

utils.normalizeStokes

plot()
Generic plot of spectral estimates

Multitaper

class bispy.spectral.**Multitaper**(*t*, *x*, *bw=2.5*, *computeFlag=True*)
Compute a multitaper spectral estimate of the spectrum of bivariate signals taken as (1, i)-quaternion valued signals. The data tapers are chosen as discrete-prolate spheroidal sequences (dpss or Slepian tapers).

Parameters

t
[array_type] time samples array

x
[array_type] input signal array (has to be of quaternion dtype)

bw
[float, optional] spectral bandwidth. Default is 2.5

computeFlag
[bool, optional] Flag activating computation of the estimate. Default is true. If False one has to run the compute() method manually.

Attributes

t
[array_type] time samples array

signal
[array_type] input signal array

f
[array_type] sampled frequencies array

densities
[array_type] spectral density quaternion array for each taper

density
[array_type] spectral density quaternion array

dpss
[array_type] data tapers used

S0, S1, S2, S3
[array_type] Stokes parameters, non-normalized [w.r.t. S0]

S1n, S2n, S3n
[array_type] normalized Stokes parameters [w.r.t. S0] using the tolerance factor *tol*. They are not computed by default. See *normalize*.

Phi
[array_type] Degree of polarization. Not computed by default; See *normalize*.

Methods

compute([bw])	Low-level method that computes the multitaper estimate
normalize([tol])	Normalize Stokes parameters wrt S0.
plot()	Generic plot of spectral estimates

compute(*bw=2.5*)

Low-level method that computes the multitaper estimate

normalize(*tol=0.0*)

Normalize Stokes parameters wrt S0. In addition, compute the degree of polarization Phi.

Parameters

tol
[float, optional] tolerance factor used in Stokes parameters normalization. Default is 0.0

Returns

self.S1n, self.S2n, self.S3n
[array_type] normalized Stokes parameters

self.Phi
[array_type] degree of polarization

See also:

`utils.normalizeStokes`

`plot()`
Generic plot of spectral estimates

5.2.5 `filters` - LTI filters for bivariate signals

Unitary Filter

`class bispy.filters.UnitaryFilter(N, mu, alpha, phi, dt=1.0)`

Unitary filter for bivariate signals. The Unitary filtering relation reads in the QFT spectral domain:

Y(nu) = exp(mu(nu)*alpha(nu) / 2)*X(nu)exp(1j*phi(nu))

where phi is phase delay of the filter, mu its axis and alpha is the birefringence angle.

Parameters

N
[int] length of the filter

mu
[array_type (quaternion)] birefringence axis quaternion array (should be of size N and of dtype quaternion).

alpha
[array_type] birefringence angle array (should be of size N). If alpha is a float, then alpha is assumed constant throughout frequencies.

phi
[array_type or float] phase delay array (should be of size N). If phi is a float, then a constant phase delay is assumed throughout frequencies.

dt
[float (optional)] time sampling step (default 1)

Attributes

N
[int] length of the filter

f
[array_type] sampled frequencies

dt
[float] time sampling step (default 1)

mu, alpha, phi
[array_types] filter parameters

Methods

output(x)	returns the output of the filter given an input signal x

output(*x*)

returns the output of the filter given an input signal x

HermitianFilter

class bispy.filters.**HermitianFilter**(*N*, *K*, *eta*, *mu*, *dt=1.0*)

Hermitian filter for bivariate signals. The Hermitian filtering relations reads in the QFT spectral domain:

Y(nu) = K(nu)*[X(nu) - eta(nu)*mu(nu)*X(nu)*qj]

where K is the homogeneous gain of the filter, eta is the polarizing power and mu the axis of the filter.

Parameters

N
[int] length of the filter

K
[array_type or float] homogeneous gain array (should be of size N). If K is a float, then a constant gain is assumed throughout frequencies.

eta
[array_type or float] polarizing power array (should be of size N). If eta is a float, then a constant polarizing is assumed throughout frequencies.

mu
[array_type (quaternion) or quaternion] diattenuation axis quaternion array (should be of size N and of dtype quaternion).

dt
[float (optional)] time sampling step (default 1)

Attributes

N
[int] length of the filter

f
[array_type] sampled frequencies

dt
[float] time sampling step (default 1)

K, eta, mu
[array_types] filter parameters

Methods

output(x)	returns the output of the filter given an input signal x

output(*x*)

returns the output of the filter given an input signal x

5.2.6 signals - generating bivariate signals

Prototype signals

bivariateAMFM(a, theta, chi, phi[, ...])	Construct a bivariate AM-FM model with specified parameters.
bivariatewhiteNoise(N, S0[, P, theta, ...])	Generates a bivariate white noise with prescribed polarization properties using the Unpolarized/Polarized part decomposition.

bispy.signals.bivariateAMFM

bispy.signals.**bivariateAMFM**(*a*, *theta*, *chi*, *phi*, *Hembedding=True*, *complexOutput=False*)

Construct a bivariate AM-FM model with specified parameters.

The output x[n] is constructed as:

```
x[n] = a[n] * np.exp(i * theta[n]) * np.exp(-k * chi[n]) * np.exp(j *␣
↪phi[n])
```

Parameters

a, theta, chi, phi
[array_type] These are instantaneous geometrical and phase parameters.

Hembedding
[bool, optional] If *True*, returns the H-embedding signal of x, otherwise returns x (1, i)-complex (as a quaternion array). Default is *True*.

complexOutput
[bool, optional] If *True*, output is a complex numpy array. Otherwise output is a quaternion numpy array. Default is *False*.

Returns

x
[array_type]

See also:

euler2quat

bispy.signals.bivariatewhiteNoise

bispy.signals.**bivariatewhiteNoise**(*N*, *S0*, *P=0*, *theta=0*, *complexOutput=False*)

Generates a bivariate white noise with prescribed polarization properties using the Unpolarized/Polarized part decomposition.

Parameters

N
[int] length of the signal

S0
[float] white noise power

P
[float, optional] degree of polarization, must be 0 <= P <= 1. Default is 0

theta
[float, optional] angle of linear polarization. Default is 0

complexOutput: bool, optional
If *True*, output is a complex numpy array. Otherwise output is a quaternion numpy array. Default is *False*.

returns
———-
w
[array_type] bivariate white noise signal

class bispy.signals.**stationaryBivariate**(*targetPSD*)

Simulates realizations of a stationary Gaussian random bivariate sequence with specficied quaternion PSD.

The simulation method relies on spectral synthesis and is approximate. The quality of the approximation increases with the size N (aka the number of frequency bins).

Parameters

targetPSD: quaternionPSD object
target PSD of the signal to sample from

Attributes

simulation
[array_type] array of size (M, N) where M is the number of independent realizations of the signal and N is the length of the simulated sequence.

Methods

output(x)	returns the output of the filter given an input signal x
simulate(M)	Simulate realizations of the stationary Gaussian random bivariate signal with specified quaternion PSD.

simulate(*M*)

Simulate realizations of the stationary Gaussian random bivariate signal with specified quaternion PSD.

Parameters

M
[int] number of independent realizations to simulate

PYTHON MODULE INDEX

INDEX

W