PyWavelets Documentation

Release 0.4.0

The PyWavelets Developers

December 28, 2015

Contents

PyWavelets is free and Open Source wavelet transform software for the Python programming language. It combines a simple high level interface with low level C and Cython performance.

PyWavelets is very easy to use and get started with. Just install the package, open the Python interactive shell and type:

```
>>> import pywt
>>> cA, cD = pywt.dwt([1, 2, 3, 4], 'db1')
```

Voilà! Computing wavelet transforms has never been so simple :)

CHAPTER 1

Main features

The main features of PyWavelets are:

- 1D, 2D and nD Forward and Inverse Discrete Wavelet Transform (DWT and IDWT)
- 1D, 2D and nD Multilevel DWT and IDWT
- 1D and 2D Stationary Wavelet Transform (Undecimated Wavelet Transform)
- 1D and 2D Wavelet Packet decomposition and reconstruction
- Approximating wavelet and scaling functions
- Over seventy built-in wavelet filters and custom wavelets supported
- Single and double precision calculations
- Real and complex calculations
- Results compatible with Matlab Wavelet Toolbox (TM)

CHAPTER 2

Requirements

PyWavelets is a package for the Python programming language. It requires:

- Python 2.6, 2.7 or >=3.3
- Numpy >= 1.6.2

CHAPTER 3

Download

The most recent *development* version can be found on GitHub at https://github.com/PyWavelets/pywt.

Latest release, including source and binary package for Windows, is available for download from the Python Package Index or on the Releases Page.

CHAPTER 4

Install

In order to build PyWavelets from source, a working C compiler (GCC or MSVC) and a recent version of Cython is required.

- Install PyWavelets with `pip install PyWavelets`.
- To build and install from source, navigate to downloaded PyWavelets source code directory and type `python setup.py install`.

Prebuilt Windows binaries and source code packages are also available from Python Package Index.

Binary packages for several Linux distributors are maintained by Open Source community contributors. Query your Linux package manager tool for *python-pywavelets*, *python-wavelets*, *python-pywt* or similar package name.

See also:

Development notes section contains more information on building and installing from source code.

CHAPTER 5

Documentation

Documentation with detailed examples and links to more resources is available online at http://pywavelets.readthedocs.org.

For more usage examples see the demo directory in the source package.

State of development & Contributing

PyWavelets started in 2006 as an academic project for a masters thesis on *Analysis and Classification of Medical Signals using Wavelet Transforms* and was maintained until 2012 by its original developer. In 2013 maintenance was taken over in a new repo) by a larger development team - a move supported by the original developer. The repo move doesn't mean that this is a fork - the package continues to be developed under the name "PyWavelets", and released on PyPi and Github (see this issue for the discussion where that was decided).

All contributions including bug reports, bug fixes, new feature implementations and documentation improvements are welcome. Moreover, developers with an interest in PyWavelets are very welcome to join the development team!

CHAPTER 7

Python 3

Python 3.x is fully supported from release v0.3.0 on.

CHAPTER 8

Contact

Use GitHub Issues or the PyWavelets discussions group to post your comments or questions.

CHAPTER 9

License

PyWavelets is a free Open Source software released under the MIT license.

CHAPTER 10

Contents

10.1 API Reference

10.1.1 Wavelets

Wavelet `families()`

`pywt.families(short=True)`

Returns a list of available built-in wavelet families.

Currently the built-in families are:

- Haar (`haar`)
- Daubechies (`db`)
- Symlets (`sym`)
- Coiflets (`coif`)
- Biorthogonal (`bior`)
- Reverse biorthogonal (`rbio`)
- *“Discrete”* FIR approximation of Meyer wavelet (`dmey`)

Parameters short : bool, optional

Use short names (default: True).

Returns families : list

List of available wavelet families.

Examples

```
>>> import pywt
>>> pywt.families()
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey']
>>> pywt.families(short=False)
['Haar', 'Daubechies', 'Symlets', 'Coiflets', 'Biorthogonal', 'Reverse biorthogonal', 'Discrete
```

Built-in wavelets - `wavelist()`

`pywt.wavelist`(*family=None*)

Returns list of available wavelet names for the given family name.

Parameters **family** : {'haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey'}

Short family name. If the family name is None (default) then names of all the built-in wavelets are returned. Otherwise the function returns names of wavelets that belong to the given family.

Returns **wavelist** : list

List of available wavelet names

Examples

```
>>> import pywt
>>> pywt.wavelist('coif')
['coif1', 'coif2', 'coif3', 'coif4', 'coif5']
```

Custom user wavelets are also supported through the `Wavelet` object constructor as described below.

`Wavelet` object

class `pywt.Wavelet`(*name*[, *filter_bank=None*])

Describes properties of a wavelet identified by the specified wavelet *name*. In order to use a built-in wavelet the *name* parameter must be a valid wavelet name from the `pywt.wavelist()` list.

Custom Wavelet objects can be created by passing a user-defined filters set with the *filter_bank* parameter.

Parameters

- **name** – Wavelet name
- **filter_bank** – Use a user supplied filter bank instead of a built-in `Wavelet`.

The filter bank object can be a list of four filters coefficients or an object with `filter_bank` attribute, which returns a list of such filters in the following order:

```
[dec_lo, dec_hi, rec_lo, rec_hi]
```

Wavelet objects can also be used as a base filter banks. See section on *using custom wavelets* for more information.

Example:

```
>>> import pywt
>>> wavelet = pywt.Wavelet('db1')
```

name
Wavelet name.

short_name
Short wavelet name.

dec_lo
Decomposition filter values.

dec_hi
Decomposition filter values.

rec_lo
Reconstruction filter values.

rec_hi
Reconstruction filter values.

dec_len
Decomposition filter length.

rec_len
Reconstruction filter length.

filter_bank
Returns filters list for the current wavelet in the following order:

```
[dec_lo, dec_hi, rec_lo, rec_hi]
```

inverse_filter_bank
Returns list of reverse wavelet filters coefficients. The mapping from the *filter_coeffs* list is as follows:

```
[rec_lo[::-1], rec_hi[::-1], dec_lo[::-1], dec_hi[::-1]]
```

short_family_name
Wavelet short family name

family_name
Wavelet family name

orthogonal
Set if wavelet is orthogonal

biorthogonal
Set if wavelet is biorthogonal

symmetry
`asymmetric`, `near symmetric`, `symmetric`

vanishing_moments_psi
Number of vanishing moments for the wavelet function

vanishing_moments_phi
Number of vanishing moments for the scaling function

Example:

```
>>> def format_array(arr):
...     return "[%s]" % ", ".join(["%.14f" % x for x in arr])

>>> import pywt
>>> wavelet = pywt.Wavelet('db1')
>>> print(wavelet)
Wavelet db1
  Family name:    Daubechies
  Short name:     db
  Filters length: 2
  Orthogonal:     True
  Biorthogonal:   True
  Symmetry:       asymmetric
>>> print(format_array(wavelet.dec_lo), format_array(wavelet.dec_hi))
[0.70710678118655, 0.70710678118655] [-0.70710678118655, 0.70710678118655]
```

```
>>> print(format_array(wavelet.rec_lo), format_array(wavelet.rec_hi))
[0.70710678118655, 0.70710678118655] [0.70710678118655, -0.70710678118655]
```

Approximating wavelet and scaling functions - `Wavelet.wavefun()`

`Wavelet.`**`wavefun`**(*level*)

Changed in version 0.2: The time (space) localisation of approximation function points was added.

The `wavefun()` method can be used to calculate approximations of scaling function (*phi*) and wavelet function (*psi*) at the given level of refinement.

For `orthogonal` wavelets returns approximations of scaling function and wavelet function with corresponding x-grid coordinates:

```
[phi, psi, x] = wavelet.wavefun(level)
```

Example:

```
>>> import pywt
>>> wavelet = pywt.Wavelet('db2')
>>> phi, psi, x = wavelet.wavefun(level=5)
```

For other (`biorthogonal` but not `orthogonal`) wavelets returns approximations of scaling and wavelet function both for decomposition and reconstruction and corresponding x-grid coordinates:

```
[phi_d, psi_d, phi_r, psi_r, x] = wavelet.wavefun(level)
```

Example:

```
>>> import pywt
>>> wavelet = pywt.Wavelet('bior3.5')
>>> phi_d, psi_d, phi_r, psi_r, x = wavelet.wavefun(level=5)
```

See also:

You can find live examples of `wavefun()` usage and images of all the built-in wavelets on the Wavelet Properties Browser page.

Using custom wavelets

PyWavelets comes with a `long list` of the most popular wavelets built-in and ready to use. If you need to use a specific wavelet which is not included in the list it is very easy to do so. Just pass a list of four filters or an object with a `filter_bank` attribute as a *filter_bank* argument to the `Wavelet` constructor.

The filters list, either in a form of a simple Python list or returned via the `filter_bank` attribute, must be in the following order:

- lowpass decomposition filter
- highpass decomposition filter
- lowpass reconstruction filter
- highpass reconstruction filter

just as for the `filter_bank` attribute of the `Wavelet` class.

The Wavelet object created in this way is a standard `Wavelet` instance.

The following example illustrates the way of creating custom Wavelet objects from plain Python lists of filter coefficients and a *filter bank-like* objects.

Example:

```
>>> import pywt, math
>>> c = math.sqrt(2)/2
>>> dec_lo, dec_hi, rec_lo, rec_hi = [c, c], [-c, c], [c, c], [c, -c]
>>> filter_bank = [dec_lo, dec_hi, rec_lo, rec_hi]
>>> myWavelet = pywt.Wavelet(name="myHaarWavelet", filter_bank=filter_bank)
>>>
>>> class HaarFilterBank(object):
...     @property
...     def filter_bank(self):
...         c = math.sqrt(2)/2
...         dec_lo, dec_hi, rec_lo, rec_hi = [c, c], [-c, c], [c, c], [c, -c]
...         return [dec_lo, dec_hi, rec_lo, rec_hi]
>>> filter_bank = HaarFilterBank()
>>> myOtherWavelet = pywt.Wavelet(name="myHaarWavelet", filter_bank=filter_bank)
```

10.1.2 Signal extension modes

Because the most common and practical way of representing digital signals in computer science is with finite arrays of values, some extrapolation of the input data has to be performed in order to extend the signal before computing the *Discrete Wavelet Transform* using the cascading filter banks algorithm.

Depending on the extrapolation method, significant artifacts at the signal's borders can be introduced during that process, which in turn may lead to inaccurate computations of the *DWT* at the signal's ends.

PyWavelets provides several methods of signal extrapolation that can be used to minimize this negative effect:

- `zero` - **zero-padding** - signal is extended by adding zero samples:

```
... 0  0 | x1 x2 ... xn | 0  0 ...
```

- `constant` - **constant-padding** - border values are replicated:

```
... x1 x1 | x1 x2 ... xn | xn xn ...
```

- `symmetric` - **symmetric-padding** - signal is extended by *mirroring* samples:

```
... x2 x1 | x1 x2 ... xn | xn xn-1 ...
```

- `periodic` - **periodic-padding** - signal is treated as a periodic one:

```
... xn-1 xn | x1 x2 ... xn | x1 x2 ...
```

- `smooth` - **smooth-padding** - signal is extended according to the first derivatives calculated on the edges (straight line)

DWT performed for these extension modes is slightly redundant, but ensures perfect reconstruction. To receive the smallest possible number of coefficients, computations can be performed with the *periodization* mode:

- `periodization` - **periodization** - is like *periodic-padding* but gives the smallest possible number of decomposition coefficients. *IDWT* must be performed with the same mode.

Example:

```
    >>> import pywt
    >>> print pywt.Modes.modes
    ['zero', 'constant', 'symmetric', 'periodic', 'smooth', 'periodization']
```

Notice that you can use any of the following ways of passing wavelet and mode parameters:

```
>>> import pywt
>>> (a, d) = pywt.dwt([1,2,3,4,5,6], 'db2', 'smooth')
>>> (a, d) = pywt.dwt([1,2,3,4,5,6], pywt.Wavelet('db2'), pywt.Modes.smooth)
```

Note: Extending data in context of PyWavelets does not mean reallocation of the data in computer's physical memory and copying values, but rather computing the extra values only when they are needed. This feature saves extra memory and CPU resources and helps to avoid page swapping when handling relatively big data arrays on computers with low physical memory.

10.1.3 Discrete Wavelet Transform (DWT)

Wavelet transform has recently become a very popular when it comes to analysis, de-noising and compression of signals and images. This section describes functions used to perform single- and multilevel Discrete Wavelet Transforms.

Single level `dwt`

`pywt.`**`dwt`**`(`*data*, *wavelet*, *mode='symmetric'*, *axis=-1*`)`
Single level Discrete Wavelet Transform.

Parameters **data** : array_like

Input signal

wavelet : Wavelet object or name

Wavelet to use

mode : str, optional

Signal extension mode, see Modes

axis: int, optional

Axis over which to compute the DWT. If not given, the last axis is used.

Returns **(cA, cD)** : tuple

Approximation and detail coefficients.

Notes

Length of coefficients arrays depends on the selected mode. For all modes except periodization:

```
len(cA) == len(cD) == floor((len(data) + wavelet.dec_len - 1) / 2)
```

For periodization mode ("per"):

```
len(cA) == len(cD) == ceil(len(data) / 2)
```

Examples

```
>>> import pywt
>>> (cA, cD) = pywt.dwt([1, 2, 3, 4, 5, 6], 'db1')
>>> cA
array([ 2.12132034,  4.94974747,  7.77817459])
>>> cD
array([-0.70710678, -0.70710678, -0.70710678])
```

See the *signal extension modes* section for the list of available options and the `dwt_coeff_len()` function for information on getting the expected result length.

The transform can be performed over one axis of multi-dimensional data. By default this is the last axis. For multi-dimensional transforms see the *2D transforms* section.

Multilevel decomposition using `wavedec`

`pywt.`**`wavedec`**(*data*, *wavelet*, *mode='symmetric'*, *level=None*)

Multilevel 1D Discrete Wavelet Transform of data.

Parameters data: array_like

Input data

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

level : int, optional

Decomposition level (must be >= 0). If level is None (default) then it will be calculated using the `dwt_max_level` function.

Returns [cA_n, cD_n, cD_n-1, ..., cD2, cD1] : list

Ordered list of coefficients arrays where *n* denotes the level of decomposition. The first element (*cA_n*) of the result is approximation coefficients array and the following elements (*cD_n - cD_1*) are details coefficients arrays.

Examples

```
>>> from pywt import wavedec
>>> coeffs = wavedec([1,2,3,4,5,6,7,8], 'db1', level=2)
>>> cA2, cD2, cD1 = coeffs
>>> cD1
array([-0.70710678, -0.70710678, -0.70710678, -0.70710678])
>>> cD2
array([-2., -2.])
>>> cA2
array([  5.,  13.])
```

Partial Discrete Wavelet Transform data decomposition `downcoef`

`pywt.downcoef(part, data, wavelet, mode='symmetric', level=1)`

Partial Discrete Wavelet Transform data decomposition.

Similar to *pywt.dwt*, but computes only one set of coefficients. Useful when you need only approximation or only details at the given level.

Parameters **part** : str

Coefficients type:

- 'a' - approximations reconstruction is performed
- 'd' - details reconstruction is performed

data : array_like

Input signal.

wavelet : Wavelet object or name

Wavelet to use

mode : str, optional

Signal extension mode, see *Modes*. Default is 'symmetric'.

level : int, optional

Decomposition level. Default is 1.

Returns **coeffs** : ndarray

1-D array of coefficients.

See also:

upcoef

Maximum decomposition level - `dwt_max_level`

`pywt.dwt_max_level(data_len, filter_len)`

Compute the maximum useful level of decomposition.

Parameters **data_len** : int

Input data length.

filter_len : int

Wavelet filter length.

Returns **max_level** : int

Maximum level.

Examples

```
>>> import pywt
>>> w = pywt.Wavelet('sym5')
>>> pywt.dwt_max_level(data_len=1000, filter_len=w.dec_len)
6
```

```
>>> pywt.dwt_max_level(1000, w)
6
```

Result coefficients length - `dwt_coeff_len`

`pywt.dwt_coeff_len`(*data_len*, *filter_len*, *mode='symmetric'*)
Returns length of dwt output for given data length, filter length and mode

Parameters **data_len** : int

Data length.

filter_len : int

Filter length.

mode : str, optional (default: 'symmetric')

Signal extension mode, see Modes

Returns **len** : int

Length of dwt output.

Notes

For all modes except periodization:

```
len(cA) == len(cD) == floor((len(data) + wavelet.dec_len - 1) / 2)
```

for periodization mode ("per"):

```
len(cA) == len(cD) == ceil(len(data) / 2)
```

Based on the given *input data length*, Wavelet *decomposition filter length* and *signal extension mode*, the `dwt_coeff_len()` function calculates the length of the resulting coefficients arrays that would be created while performing `dwt()` transform.

filter_len can be either an *int* or `Wavelet` object for convenience.

10.1.4 Inverse Discrete Wavelet Transform (IDWT)

Single level `idwt`

`pywt.idwt`(*cA*, *cD*, *wavelet*, *mode='symmetric'*, *axis=-1*)
Single level Inverse Discrete Wavelet Transform.

Parameters **cA** : array_like or None

Approximation coefficients. If None, will be set to array of zeros with same shape as *cD*.

cD : array_like or None

Detail coefficients. If None, will be set to array of zeros with same shape as *cA*.

wavelet : Wavelet object or name

Wavelet to use

mode : str, optional (default: 'symmetric')

Signal extension mode, see Modes

axis: int, optional

Axis over which to compute the inverse DWT. If not given, the last axis is used.

Returns rec: array_like

Single level reconstruction of signal from given coefficients.

Example:

```
>>> import pywt
>>> (cA, cD) = pywt.dwt([1,2,3,4,5,6], 'db2', 'smooth')
>>> print pywt.idwt(cA, cD, 'db2', 'smooth')
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

One of the neat features of `idwt()` is that one of the *cA* and *cD* arguments can be set to `None`. In that situation the reconstruction will be performed using only the other one. Mathematically speaking, this is equivalent to passing a zero-filled array as one of the arguments.

Example:

```
>>> import pywt
>>> (cA, cD) = pywt.dwt([1,2,3,4,5,6], 'db2', 'smooth')
>>> A = pywt.idwt(cA, None, 'db2', 'smooth')
>>> D = pywt.idwt(None, cD, 'db2', 'smooth')
>>> print A + D
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

Multilevel reconstruction using `waverec`

`pywt.`**`waverec`**(*coeffs*, *wavelet*, *mode='symmetric'*)

Multilevel 1D Inverse Discrete Wavelet Transform.

Parameters **coeffs** : array_like

Coefficients list [cAn, cDn, cDn-1, ..., cD2, cD1]

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

Examples

```
>>> import pywt
>>> coeffs = pywt.wavedec([1,2,3,4,5,6,7,8], 'db1', level=2)
>>> pywt.waverec(coeffs, 'db1')
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
```

Direct reconstruction with `upcoef`

`pywt.upcoef`(*part*, *coeffs*, *wavelet*, *level=1*, *take=0*)

Direct reconstruction from coefficients.

Parameters **part** : str

Coefficients type: * 'a' - approximations reconstruction is performed * 'd' - details reconstruction is performed

coeffs : array_like

Coefficients array to recontruct

wavelet : Wavelet object or name

Wavelet to use

level : int, optional

Multilevel reconstruction level. Default is 1.

take : int, optional

Take central part of length equal to 'take' from the result. Default is 0.

Returns **rec** : ndarray

1-D array with reconstructed data from coefficients.

See also:

downcoef

Examples

```
>>> import pywt
>>> data = [1,2,3,4,5,6]
>>> (cA, cD) = pywt.dwt(data, 'db2', 'smooth')
>>> pywt.upcoef('a', cA, 'db2') + pywt.upcoef('d', cD, 'db2')
array([-0.25       , -0.4330127 ,  1.        ,  2.        ,  3.        ,
        4.        ,  5.        ,  6.        ,  1.78589838, -1.03108891])
>>> n = len(data)
>>> pywt.upcoef('a', cA, 'db2', take=n) + pywt.upcoef('d', cD, 'db2', take=n)
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

10.1.5 2D Forward and Inverse Discrete Wavelet Transform

Single level `dwt2`

`pywt.dwt2`(*data*, *wavelet*, *mode='symmetric'*, *axes=(-2, -1)*)

2D Discrete Wavelet Transform.

Parameters **data** : ndarray

2D array with input data

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

axes : 2-tuple of ints, optional

Axes over which to compute the DWT. Repeated elements mean the DWT will be performed multiple times along these axes.

Returns (cA, (cH, cV, cD)) : tuple

Approximation, horizontal detail, vertical detail and diagonal detail coefficients respectively. Horizontal refers to array axis 0.

Examples

```
>>> import numpy as np
>>> import pywt
>>> data = np.ones((4,4), dtype=np.float64)
>>> coeffs = pywt.dwt2(data, 'haar')
>>> cA, (cH, cV, cD) = coeffs
>>> cA
array([[ 2.,  2.],
       [ 2.,  2.]])
>>> cV
array([[ 0.,  0.],
       [ 0.,  0.]])
```

The relation to the other common data layout where all the approximation and details coefficients are stored in one big 2D array is as follows:

```
                             -------------------
                             |        |        |
                             | cA(LL) | cH(LH) |
                             |        |        |
(cA, (cH, cV, cD))   <--->   -------------------
                             |        |        |
                             | cV(HL) | cD(HH) |
                             |        |        |
                             -------------------
```

PyWavelets does not follow this pattern because of pure practical reasons of simple access to particular type of the output coefficients.

Single level `idwt2`

`pywt.`**`idwt2`**(*coeffs*, *wavelet*, *mode='symmetric'*, *axes=(-2, -1)*)

2-D Inverse Discrete Wavelet Transform.

Reconstructs data from coefficient arrays.

Parameters coeffs : tuple

(cA, (cH, cV, cD)) A tuple with approximation coefficients and three details coefficients 2D arrays like from *dwt2()*

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

axes : 2-tuple of ints, optional

Axes over which to compute the IDWT. Repeated elements mean the IDWT will be performed multiple times along these axes.

Examples

```
>>> import numpy as np
>>> import pywt
>>> data = np.array([[1,2], [3,4]], dtype=np.float64)
>>> coeffs = pywt.dwt2(data, 'haar')
>>> pywt.idwt2(coeffs, 'haar')
array([[ 1.,  2.],
       [ 3.,  4.]])
```

2D multilevel decomposition using `wavedec2`

`pywt.`**`wavedec2`**(*data*, *wavelet*, *mode='symmetric'*, *level=None*)

Multilevel 2D Discrete Wavelet Transform.

Parameters **data** : ndarray

2D input data

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

level : int, optional

Decomposition level (must be >= 0). If level is None (default) then it will be calculated using the `dwt_max_level` function.

Returns **[cAn, (cHn, cVn, cDn), ... (cH1, cV1, cD1)]** : list

Coefficients list

Examples

```
>>> import pywt
>>> import numpy as np
>>> coeffs = pywt.wavedec2(np.ones((4,4)), 'db1')
>>> # Levels:
>>> len(coeffs)-1
2
>>> pywt.waverec2(coeffs, 'db1')
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

2D multilevel reconstruction using `waverec2`

`pywt.waverec2(`*coeffs*, *wavelet*, *mode='symmetric'*`)`

Multilevel 2D Inverse Discrete Wavelet Transform.

coeffs [list or tuple] Coefficients list [cAn, (cHn, cVn, cDn), ... (cH1, cV1, cD1)]

wavelet [Wavelet object or name string] Wavelet to use

mode [str, optional] Signal extension mode, see Modes (default: 'symmetric')

Returns 2D array of reconstructed data.

Examples

```
>>> import pywt
>>> import numpy as np
>>> coeffs = pywt.wavedec2(np.ones((4,4)), 'db1')
>>> # Levels:
>>> len(coeffs)-1
2
>>> pywt.waverec2(coeffs, 'db1')
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

10.1.6 nD Forward and Inverse Discrete Wavelet Transform

Single level - `dwtn`

`pywt.dwtn(`*data*, *wavelet*, *mode='symmetric'*, *axes=None*`)`

Single-level n-dimensional Discrete Wavelet Transform.

Parameters **data** : ndarray

n-dimensional array with input data.

wavelet : Wavelet object or name string

Wavelet to use.

mode : str, optional

Signal extension mode, see *Modes*. Default is 'symmetric'.

axes : sequence of ints, optional

Axes over which to compute the DWT. Repeated elements mean the DWT will be performed multiple times along these axes. A value of *None* (the default) selects all axes.

Axes may be repeated, but information about the original size may be lost if it is not divisible by *2 ** nrepeats*. The reconstruction will be larger, with additional values derived according to the *mode* parameter. *pywt.wavedecn* should be used for multilevel decomposition.

Returns **coeffs** : dict

Results are arranged in a dictionary, where key specifies the transform type on each dimension and value is a n-dimensional coefficients array.

For example, for a 2D case the result will look something like this:

```
{'aa': <coeffs>  # A(LL) - approx. on 1st dim, approx. on 2nd dim
 'ad': <coeffs>  # V(LH) - approx. on 1st dim, det. on 2nd dim
 'da': <coeffs>  # H(HL) - det. on 1st dim, approx. on 2nd dim
 'dd': <coeffs>  # D(HH) - det. on 1st dim, det. on 2nd dim
}
```

Single level - `idwtn`

`pywt.`**`idwtn`**(*coeffs*, *wavelet*, *mode='symmetric'*, *axes=None*)

Single-level n-dimensional Inverse Discrete Wavelet Transform.

Parameters **coeffs: dict**

Dictionary as in output of *dwtn*. Missing or None items will be treated as zeroes.

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode used in the decomposition, see Modes (default: 'symmetric').

axes : sequence of ints, optional

Axes over which to compute the IDWT. Repeated elements mean the IDWT will be performed multiple times along these axes. A value of *None* (the default) selects all axes.

For the most accurate reconstruction, the axes should be provided in the same order as they were provided to *dwtn*.

Returns data: ndarray

Original signal reconstructed from input data.

Multilevel decomposition - `wavedecn`

`pywt.`**`wavedecn`**(*data*, *wavelet*, *mode='symmetric'*, *level=None*)

Multilevel nD Discrete Wavelet Transform.

Parameters **data** : ndarray

nD input data

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

level : int, optional

Dxecomposition level (must be >= 0). If level is None (default) then it will be calculated using the `dwt_max_level` function.

Returns [cAn, {details_level_n}, ... {details_level_1}] : list

Coefficients list

Examples

```
>>> import numpy as np
>>> from pywt import wavedecn, waverecn
>>> coeffs = wavedecn(np.ones((4, 4, 4)), 'db1')
>>> # Levels:
>>> len(coeffs)-1
2
>>> waverecn(coeffs, 'db1')
array([[[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]]])
```

Multilevel reconstruction - `waverecn`

`pywt.`**`waverecn`**(*coeffs*, *wavelet*, *mode='symmetric'*)

Multilevel nD Inverse Discrete Wavelet Transform.

coeffs [array_like] Coefficients list [cAn, {details_level_n}, ... {details_level_1}]

wavelet [Wavelet object or name string] Wavelet to use

mode [str, optional] Signal extension mode, see Modes (default: 'symmetric')

Returns nD array of reconstructed data.

Examples

```
>>> import numpy as np
>>> from pywt import wavedecn, waverecn
>>> coeffs = wavedecn(np.ones((4, 4, 4)), 'db1')
>>> # Levels:
>>> len(coeffs)-1
2
>>> waverecn(coeffs, 'db1')
array([[[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]]]])
```

10.1.7 Stationary Wavelet Transform

Stationary Wavelet Transform (SWT), also known as *Undecimated wavelet transform* or *Algorithme à trous* is a translation-invariance modification of the *Discrete Wavelet Transform* that does not decimate coefficients at every transformation level.

Multilevel `swt`

`pywt.`**`swt`**(*data*, *wavelet*, *level=None*, *start_level=0*)

Performs multilevel Stationary Wavelet Transform.

Parameters **data :**

Input signal

wavelet :

Wavelet to use (Wavelet object or name)

level : int, optional

Transform level.

start_level : int, optional

The level at which the decomposition will begin (it allows one to skip a given number of transform steps and compute coefficients starting from start_level) (default: 0)

Returns **coeffs** : list

List of approximation and details coefficients pairs in order similar to wavedec function:

```
[(cAn, cDn), ..., (cA2, cD2), (cA1, cD1)]
```

where `n` equals input parameter *level*.

If *m* = *start_level* is given, then the beginning *m* steps are skipped:

```
[(cAm+n, cDm+n), ..., (cAm+1, cDm+1), (cAm, cDm)]
```

Multilevel `swt2`

`pywt.swt2(data, wavelet, level, start_level=0)`

2D Stationary Wavelet Transform.

Parameters **data** : ndarray

2D array with input data

wavelet : Wavelet object or name string

Wavelet to use

level : int

How many decomposition steps to perform

start_level : int, optional

The level at which the decomposition will start (default: 0)

Returns **coeffs** : list

Approximation and details coefficients:

```
[
    (cA_n,
        (cH_n, cV_n, cD_n)
    ),
    (cA_n+1,
        (cH_n+1, cV_n+1, cD_n+1)
    ),
    ...,
    (cA_n+level,
        (cH_n+level, cV_n+level, cD_n+level)
    )
]
```

where cA is approximation, cH is horizontal details, cV is vertical details, cD is diagonal details and n is start_level.

Maximum decomposition level - `swt_max_level`

`pywt.swt_max_level(input_len)`

Calculates the maximum level of Stationary Wavelet Transform for data of given length.

Parameters **input_len** : int

Input data length.

Returns **max_level** : int

Maximum level of Stationary Wavelet Transform for data of given length.

10.1.8 Wavelet Packets

New in version 0.2.

Version *0.2* of PyWavelets includes many new features and improvements. One of such new feature is a two-dimensional wavelet packet transform structure that is almost completely sharing programming interface with the one-dimensional tree structure.

In order to achieve this simplification, a new inheritance scheme was used in which a `BaseNode` base node class is a superclass for both `Node` and `Node2D` node classes.

The node classes are used as data wrappers and can be organized in trees (binary trees for 1D transform case and quad-trees for the 2D one). They are also superclasses to the `WaveletPacket` class and `WaveletPacket2D` class that are used as the decomposition tree roots and contain a couple additional methods.

The below diagram illustrates the inheritance tree:

- `BaseNode` - common interface for 1D and 2D nodes:
  - `Node` - data carrier node in a 1D decomposition tree
    - `WaveletPacket` - 1D decomposition tree root node
  - `Node2D` - data carrier node in a 2D decomposition tree
    - `WaveletPacket2D` - 2D decomposition tree root node

BaseNode - a common interface of WaveletPacket and WaveletPacket2D

class `pywt.`**`BaseNode`**
class `pywt.`**`Node`**`(`*BaseNode*`)`
class `pywt.`**`WaveletPacket`**`(`*Node*`)`
class `pywt.`**`Node2D`**`(`*BaseNode*`)`
class `pywt.`**`WaveletPacket2D`**`(`*Node2D*`)`

> **Note:** The BaseNode is a base class for `Node` and `Node2D`. It should not be used directly unless creating a new transformation type. It is included here to document the common interface of 1D and 2D node an wavelet packet transform classes.

`__init__``(`*parent*, *data*, *node_name*`)`

Parameters

- **`parent`** – parent node. If parent is `None` then the node is considered detached.
- **`data`** – data associated with the node. 1D or 2D numeric array, depending on the transform type.
- **`node_name`** – a name identifying the coefficients type. See `Node.node_name` and `Node2D.node_name` for information on the accepted subnodes names.

`data`
Data associated with the node. 1D or 2D numeric array (depends on the transform type).

`parent`
Parent node. Used in tree navigation. `None` for root node.

`wavelet`
`Wavelet` used for decomposition and reconstruction. Inherited from parent node.

`mode`
Signal extension *mode* for the `dwt()` (`dwt2()`) and `idwt()` (`idwt2()`) decomposition and reconstruction functions. Inherited from parent node.

`level`
Decomposition level of the current node. `0` for root (original data), `1` for the first decomposition level, etc.

`path`
Path string defining position of the node in the decomposition tree.

node_name
Node name describing `data` coefficients type of the current subnode.

See `Node.node_name` and `Node2D.node_name`.

maxlevel
Maximum allowed level of decomposition. Evaluated from parent or child nodes.

is_empty
Checks if `data` attribute is `None`.

has_any_subnode
Checks if node has any subnodes (is not a leaf node).

decompose()
Performs Discrete Wavelet Transform on the `data` and returns transform coefficients.

reconstruct([*update=False*])
Performs Inverse Discrete Wavelet Transform on subnodes coefficients and returns reconstructed data for the current level.

Parameters **update** – If set, the `data` attribute will be updated with the reconstructed value.

Note: Descends to subnodes and recursively calls `reconstruct()` on them.

get_subnode(*part*[, *decompose=True*])
Returns subnode or None (see *decomposition* flag description).

Parameters

- **part** – Subnode name
- **decompose** – If True and subnode does not exist, it will be created using coefficients from the DWT decomposition of the current node.

__getitem__(*path*)
Used to access nodes in the decomposition tree by string *path*.

Parameters **path** – Path string composed from valid node names. See `Node.node_name` and `Node2D.node_name` for node naming convention.

Similar to `get_subnode()` method with *decompose=True*, but can access nodes on any level in the decomposition tree.

If node does not exist yet, it will be created by decomposition of its parent node.

__setitem__(*path*, *data*)
Used to set node or node's data in the decomposition tree. Nodes are identified by string *path*.

Parameters

- **path** – Path string composed from valid node names. See `Node.node_name` and `Node2D.node_name` for node naming convention.
- **data** – numeric array or `BaseNode` subclass.

__delitem__(*path*)
Used to delete node from the decomposition tree.

Parameters **path** – Path string composed from valid node names. See `Node.node_name` and `Node2D.node_name` for node naming convention.

get_leaf_nodes([*decompose=False*])
Traverses through the decomposition tree and collects leaf nodes (nodes without any subnodes).

Parameters **decompose** – If *decompose* is `True`, the method will try to decompose the tree up to the `maximum level`.

walk (*self*, *func*[, *args=()*[, *kwargs={}*[, *decompose=True*]]])
Traverses the decomposition tree and calls `func(node, *args, **kwargs)` on every node. If *func* returns `True`, descending to subnodes will continue.

Parameters

- **func** – callable accepting `BaseNode` as the first param and optional positional and keyword arguments:

```
func(node, *args, **kwargs)
```

- **decompose** – If *decompose* is `True` (default), the method will also try to decompose the tree up to the `maximum level`.

Args arguments to pass to the *func*

Kwargs keyword arguments to pass to the *func*

walk_depth (*self*, *func*[, *args=()*[, *kwargs={}*[, *decompose=False*]]])
Similar to `walk()` but traverses the tree in depth-first order.

Parameters

- **func** – callable accepting `BaseNode` as the first param and optional positional and keyword arguments:

```
func(node, *args, **kwargs)
```

- **decompose** – If *decompose* is `True`, the method will also try to decompose the tree up to the `maximum level`.

Args arguments to pass to the *func*

Kwargs keyword arguments to pass to the *func*

WaveletPacket and WaveletPacket tree Node

class pywt.**Node** (*BaseNode*)
class pywt.**WaveletPacket** (*Node*)

node_name
Node name describing `data` coefficients type of the current subnode.

For `WaveletPacket` case it is just as in `dwt()`:

- `a` - approximation coefficients
- `d` - details coefficients

decompose ()

See also:

- `dwt()` for 1D Discrete Wavelet Transform output coefficients.

class `pywt.`**`WaveletPacket`**(*Node*)

`__init__`(*data*, *wavelet*[, *mode='symmetric'*[, *maxlevel=None*]])

Parameters

- **`data`** – data associated with the node. 1D numeric array.
- **`wavelet`** – Wavelet to use in the transform. This can be a name of the wavelet from the *`wavelist()`* list or a *`Wavelet`* object instance.
- **`mode`** – Signal extension *mode* for the *`dwt()`* and *`idwt()`* decomposition and reconstruction functions.
- **`maxlevel`** – Maximum allowed level of decomposition. If not specified it will be calculated based on the *wavelet* and *data* length using *`pywt.dwt_max_level()`*.

`get_level`(*level*[, *order="natural"*[, *decompose=True*]])

Collects nodes from the given level of decomposition.

Parameters

- **`level`** – Specifies decomposition *level* from which the nodes will be collected.
- **`order`** – Specifies nodes order - natural (`natural`) or frequency (`freq`).
- **`decompose`** – If set then the method will try to decompose the data up to the specified *level*.

If nodes at the given level are missing (i.e. the tree is partially decomposed) and the *decompose* is set to `False`, only existing nodes will be returned.

WaveletPacket2D and WaveletPacket2D tree Node2D

class `pywt.`**`Node2D`**(*BaseNode*)

class `pywt.`**`WaveletPacket2D`**(*Node2D*)

`node_name`

For *`WaveletPacket2D`* case it is just as in *`dwt2()`*:

- `a` - approximation coefficients (*LL*)
- `h` - horizontal detail coefficients (*LH*)
- `v` - vertical detail coefficients (*HL*)
- `d` - diagonal detail coefficients (*HH*)

`decompose`()

See also:

`dwt2()` for 2D Discrete Wavelet Transform output coefficients.

`expand_2d_path(self, path):`

class `pywt.`**`WaveletPacket2D`**(*Node2D*)

`__init__`(*data*, *wavelet*[, *mode='symmetric'*[, *maxlevel=None*]])

Parameters

- **data** – data associated with the node. 2D numeric array.
- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the `wavelist()` list or a `Wavelet` object instance.
- **mode** – Signal extension *mode* for the `dwt()` and `idwt()` decomposition and reconstruction functions.
- **maxlevel** – Maximum allowed level of decomposition. If not specified it will be calculated based on the *wavelet* and *data* length using `pywt.dwt_max_level()`.

get_level(*level*[, *order="natural"*[, *decompose=True*]])

Collects nodes from the given level of decomposition.

Parameters

- **level** – Specifies decomposition *level* from which the nodes will be collected.
- **order** – Specifies nodes order - natural (`natural`) or frequency (`freq`).
- **decompose** – If set then the method will try to decompose the data up to the specified *level*.

If nodes at the given level are missing (i.e. the tree is partially decomposed) and the *decompose* is set to `False`, only existing nodes will be returned.

10.1.9 Thresholding functions

The `thresholding` helper module implements the most popular signal thresholding functions.

Thresholding

pywt.**threshold**(*data*, *value*, *mode='soft'*, *substitute=0*)

Thresholds the input data depending on the mode argument.

In `soft` thresholding, the data values where their absolute value is less than the value param are replaced with substitute. From the data values with absolute value greater or equal to the thresholding value, a quantity of `(signum * value)` is subtracted.

In `hard` thresholding, the data values where their absolute value is less than the value param are replaced with substitute. Data values with absolute value greater or equal to the thresholding value stay untouched.

In `greater` thresholding, the data is replaced with substitute where data is below the thresholding value. Greater data values pass untouched.

In `less` thresholding, the data is replaced with substitute where data is above the thresholding value. Less data values pass untouched.

Parameters **data** : array_like

Numeric data.

value : scalar

Thresholding value.

mode : {'soft', 'hard', 'greater', 'less'}

Decides the type of thresholding to be applied on input data. Default is 'soft'.

substitute : float, optional

Substitute value (default: 0).

Returns output : array

Thresholded array.

Examples

```
>>> import numpy as np
>>> import pywt
>>> data = np.linspace(1, 4, 7)
>>> data
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ])
>>> pywt.threshold(data, 2, 'soft')
array([ 0. ,  0. ,  0. ,  0.5,  1. ,  1.5,  2. ])
>>> pywt.threshold(data, 2, 'hard')
array([ 0. ,  0. ,  2. ,  2.5,  3. ,  3.5,  4. ])
>>> pywt.threshold(data, 2, 'greater')
array([ 0. ,  0. ,  2. ,  2.5,  3. ,  3.5,  4. ])
>>> pywt.threshold(data, 2, 'less')
array([ 1. ,  1.5,  2. ,  0. ,  0. ,  0. ,  0. ])
```

10.1.10 Other functions

Integrating wavelet functions

pywt.**integrate_wavelet**(*wavelet*, *precision=8*)

Integrate *psi* wavelet function from -Inf to x using the rectangle integration method.

Parameters wavelet : Wavelet instance or str

Wavelet to integrate. If a string, should be the name of a wavelet.

precision : int, optional

Precision that will be used for wavelet function approximation computed with the wavefun(level=precision) Wavelet's method (default: 8).

Returns [int_psi, x] :

for orthogonal wavelets

[int_psi_d, int_psi_r, x] :

for other wavelets

Examples

```
>>> from pywt import Wavelet, integrate_wavelet
>>> wavelet1 = Wavelet('db2')
>>> [int_psi, x] = integrate_wavelet(wavelet1, precision=5)
>>> wavelet2 = Wavelet('bior1.3')
>>> [int_psi_d, int_psi_r, x] = integrate_wavelet(wavelet2, precision=5)
```

The result of the call depends on the *wavelet* argument:

- for orthogonal and continuous wavelets - an integral of the wavelet function specified on an x-grid:

```
[int_psi, x_grid] = integrate_wavelet(wavelet, precision)
```

- for other wavelets - integrals of decomposition and reconstruction wavelet functions and a corresponding x-grid:

```
[int_psi_d, int_psi_r, x_grid] = integrate_wavelet(wavelet, precision)
```

Central frequency of *psi* wavelet function

`pywt.`**`central_frequency`**(*wavelet*, *precision=8*)

Computes the central frequency of the *psi* wavelet function.

Parameters **wavelet** : Wavelet instance, str or tuple

Wavelet to integrate. If a string, should be the name of a wavelet.

precision : int, optional

Precision that will be used for wavelet function approximation computed with the wavefun(level=precision) Wavelet's method (default: 8).

Returns scalar

`pywt.`**`scale2frequency`**(*wavelet*, *scale*, *precision=8*)

Parameters **wavelet** : Wavelet instance or str

Wavelet to integrate. If a string, should be the name of a wavelet.

scale : scalar

precision : int, optional

Precision that will be used for wavelet function approximation computed with `wavelet.wavefun(level=precision)`. Default is 8.

Returns **freq** : scalar

Quadrature Mirror Filter

`pywt.`**`qmf`**(*filter*)

Returns the Quadrature Mirror Filter(QMF).

The magnitude response of QMF is mirror image about *pi/2* of that of the input filter.

Parameters **filter** : array_like

Input filter for which QMF needs to be computed.

Returns **qm_filter** : ndarray

Quadrature mirror of the input filter.

Orthogonal Filter Banks

`pywt.`**`orthogonal_filter_bank`**(*scaling_filter*)

Returns the orthogonal filter bank.

The orthogonal filter bank consists of the HPFs and LPFs at decomposition and reconstruction stage for the input scaling filter.

Parameters **scaling_filter** : array_like

Input scaling filter (father wavelet).

Returns orth_filt_bank : tuple of 4 ndarrays

The orthogonal filter bank of the input scaling filter in the order : 1] Decomposition LPF 2] Decomposition HPF 3] Reconstruction LPF 4] Reconstruction HPF

Example Datasets

The following example datasets are available in the module *pywt.data*:

name	description
ecg	ECG waveform (1024 samples)
aero	grayscale image (512x512)
ascent	grayscale image (512x512)
camera	grayscale image (512x512)

Each can be loaded via a function of the same name.

Example: .. sourcecode:: python

```
>>> import pywt
>>> camera = pywt.data.camera()
```

10.2 Usage examples

The following examples are used as doctest regression tests written using reST markup. They are included in the documentation since they contain various useful examples illustrating how to use and how not to use PyWavelets.

10.2.1 The Wavelet object

Wavelet families and builtin Wavelets names

`Wavelet` objects are really a handy carriers of a bunch of DWT-specific data like *quadrature mirror filters* and some general properties associated with them.

At first let's go through the methods of creating a `Wavelet` object. The easiest and the most convenient way is to use builtin named Wavelets.

These wavelets are organized into groups called wavelet families. The most commonly used families are:

```
>>> import pywt
>>> pywt.families()
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey']
```

The `wavelist()` function with family name passed as an argument is used to obtain the list of wavelet names in each family.

```
>>> for family in pywt.families():
...     print("%s family: " % family + ', '.join(pywt.wavelist(family)))
haar family: haar
db family: db1, db2, db3, db4, db5, db6, db7, db8, db9, db10, db11, db12, db13, db14, db15, db16, db1
sym family: sym2, sym3, sym4, sym5, sym6, sym7, sym8, sym9, sym10, sym11, sym12, sym13, sym14, sym15,
coif family: coif1, coif2, coif3, coif4, coif5
bior family: bior1.1, bior1.3, bior1.5, bior2.2, bior2.4, bior2.6, bior2.8, bior3.1, bior3.3, bior3.5
```

```
rbio family: rbio1.1, rbio1.3, rbio1.5, rbio2.2, rbio2.4, rbio2.6, rbio2.8, rbio3.1, rbio3.3, rbio3.5
dmey family: dmey
```

To get the full list of builtin wavelets' names just use the *wavelist()* with no argument. As you can see currently there are 76 builtin wavelets.

```
>>> len(pywt.wavelist())
76
```

Creating Wavelet objects

Now when we know all the names let's finally create a *Wavelet* object:

```
>>> w = pywt.Wavelet('db3')
```

So.. that's it.

Wavelet properties

But what can we do with *Wavelet* objects? Well, they carry some interesting information.

First, let's try printing a *Wavelet* object. This shows a brief information about its name, its family name and some properties like orthogonality and symmetry.

```
>>> print(w)
Wavelet db3
  Family name:    Daubechies
  Short name:     db
  Filters length: 6
  Orthogonal:     True
  Biorthogonal:   True
  Symmetry:       asymmetric
```

But the most important information are the wavelet filters coefficients, which are used in *Discrete Wavelet Transform*. These coefficients can be obtained via the *dec_lo*, *Wavelet.dec_hi*, *rec_lo* and *rec_hi* attributes, which corresponds to lowpass and highpass decomposition filters and lowpass and highpass reconstruction filters respectively:

```
>>> def print_array(arr):
...     print("[%s]" % ", ".join(["%.14f" % x for x in arr]))
```

```
>>> print_array(w.dec_lo)
[0.03522629188210, -0.08544127388224, -0.13501102001039, 0.45987750211933, 0.80689150931334, 0.33267
>>> print_array(w.dec_hi)
[-0.33267055295096, 0.80689150931334, -0.45987750211933, -0.13501102001039, 0.08544127388224, 0.03522
>>> print_array(w.rec_lo)
[0.33267055295096, 0.80689150931334, 0.45987750211933, -0.13501102001039, -0.08544127388224, 0.035226
>>> print_array(w.rec_hi)
[0.03522629188210, 0.08544127388224, -0.13501102001039, -0.45987750211933, 0.80689150931334, -0.3326
```

Another way to get the filters data is to use the *filter_bank* attribute, which returns all four filters in a tuple:

```
>>> w.filter_bank == (w.dec_lo, w.dec_hi, w.rec_lo, w.rec_hi)
True
```

Other Wavelet's properties are:

Wavelet *name*, *short_family_name* and *family_name*:

```
>>> print(w.name)
db3
>>> print(w.short_family_name)
db
>>> print(w.family_name)
Daubechies
```

- Decomposition (`dec_len`) and reconstruction (`rec_len`) filter lengths:

```
>>> int(w.dec_len) # int() is for normalizing longs and ints for doctest
6
>>> int(w.rec_len)
6
```

- Orthogonality (`orthogonal`) and biorthogonality (`biorthogonal`):

```
>>> w.orthogonal
True
>>> w.biorthogonal
True
```

- Symmetry (`symmetry`):

```
>>> print(w.symmetry)
asymmetric
```

- Number of vanishing moments for the scaling function *phi* (`vanishing_moments_phi`) and the wavelet function *psi* (`vanishing_moments_psi`) associated with the filters:

```
>>> w.vanishing_moments_phi
0
>>> w.vanishing_moments_psi
3
```

Now when we know a bit about the builtin Wavelets, let's see how to create *custom Wavelets* objects. These can be done in two ways:

1. Passing the filter bank object that implements the *filter_bank* attribute. The attribute must return four filters coefficients.

```
>>> class MyHaarFilterBank(object):
...     @property
...     def filter_bank(self):
...         from math import sqrt
...         return ([sqrt(2)/2, sqrt(2)/2], [-sqrt(2)/2, sqrt(2)/2],
...                 [sqrt(2)/2, sqrt(2)/2], [sqrt(2)/2, -sqrt(2)/2])
```

```
>>> my_wavelet = pywt.Wavelet('My Haar Wavelet', filter_bank=MyHaarFilterBank())
```

2. Passing the filters coefficients directly as the *filter_bank* parameter.

```
>>> from math import sqrt
>>> my_filter_bank = ([sqrt(2)/2, sqrt(2)/2], [-sqrt(2)/2, sqrt(2)/2],
...                   [sqrt(2)/2, sqrt(2)/2], [sqrt(2)/2, -sqrt(2)/2])
>>> my_wavelet = pywt.Wavelet('My Haar Wavelet', filter_bank=my_filter_bank)
```

Note that such custom wavelets **will not** have all the properties set to correct values:

```
>>> print(my_wavelet)
Wavelet My Haar Wavelet
  Family name:
```

```
  Short name:
  Filters length: 2
  Orthogonal:     False
  Biorthogonal:   False
  Symmetry:       unknown
```

You can however set a few of them on your own:

```
>>> my_wavelet.orthogonal = True
>>> my_wavelet.biorthogonal = True
```

```
>>> print(my_wavelet)
Wavelet My Haar Wavelet
  Family name:
  Short name:
  Filters length: 2
  Orthogonal:     True
  Biorthogonal:   True
  Symmetry:       unknown
```

And now... the *wavefun*!

We all know that the fun with wavelets is in wavelet functions. Now what would be this package without a tool to compute wavelet and scaling functions approximations?

This is the purpose of the `wavefun()` method, which is used to approximate scaling function (*phi*) and wavelet function (*psi*) at the given level of refinement, based on the filters coefficients.

The number of returned values varies depending on the wavelet's orthogonality property. For orthogonal wavelets the result is tuple with scaling function, wavelet function and xgrid coordinates.

```
>>> w = pywt.Wavelet('sym3')
>>> w.orthogonal
True
>>> (phi, psi, x) = w.wavefun(level=5)
```

For biorthogonal (non-orthogonal) wavelets different scaling and wavelet functions are used for decomposition and reconstruction, and thus five elements are returned: decomposition scaling and wavelet functions approximations, reconstruction scaling and wavelet functions approximations, and the xgrid.

```
>>> w = pywt.Wavelet('bior1.3')
>>> w.orthogonal
False
>>> (phi_d, psi_d, phi_r, psi_r, x) = w.wavefun(level=5)
```

See also:

You can find live examples of `wavefun()` usage and images of all the built-in wavelets on the Wavelet Properties Browser page.

10.2.2 Signal Extension Modes

Import `pywt` first

```
>>> import pywt
>>> def format_array(a):
...     """Consistent array representation across different systems"""
...     import numpy
...     a = numpy.where(numpy.abs(a) < 1e-5, 0, a)
...     return numpy.array2string(a, precision=5, separator=' ', suppress_small=True)
```

List of available signal extension *modes*:

```
>>> print(pywt.Modes.modes)
['zero', 'constant', 'symmetric', 'periodic', 'smooth', 'periodization']
```

Test that `dwt()` and `idwt()` can be performed using every mode:

```
>>> x = [1,2,1,5,-1,8,4,6]
>>> for mode in pywt.Modes.modes:
...     cA, cD = pywt.dwt(x, 'db2', mode)
...     print("Mode: %s" % mode)
...     print("cA: " + format_array(cA))
...     print("cD: " + format_array(cD))
...     print("Reconstruction: " + format_array(
...         pywt.idwt(cA, cD, 'db2', mode)))
Mode: zero
cA: [-0.03468  1.73309  3.40612  6.32929  6.95095]
cD: [-0.12941 -2.156   -5.95035 -1.21545 -1.8625 ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: constant
cA: [ 1.2848   1.73309  3.40612  6.32929  7.51936]
cD: [-0.48296 -2.156   -5.95035 -1.21545  0.25882]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: symmetric
cA: [ 1.76777  1.73309  3.40612  6.32929  7.77817]
cD: [-0.61237 -2.156   -5.95035 -1.21545  1.22474]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: periodic
cA: [ 6.91627  1.73309  3.40612  6.32929  6.91627]
cD: [-1.99191 -2.156   -5.95035 -1.21545 -1.99191]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: smooth
cA: [-0.51764  1.73309  3.40612  6.32929  7.45001]
cD: [ 0.      -2.156   -5.95035 -1.21545  0.      ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: periodization
cA: [ 4.05317  3.05257  2.85381  8.42522]
cD: [ 0.18947  4.18258  4.33738  2.60428]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
```

Invalid mode name should rise a `ValueError`:

```
>>> pywt.dwt([1,2,3,4], 'db2', 'invalid')
Traceback (most recent call last):
...
ValueError: Unknown mode name 'invalid'.
```

You can also refer to modes via *Modes* class attributes:

```
>>> for mode_name in ['zero', 'constant', 'symmetric', 'periodic', 'smooth', 'periodization']:
...     mode = getattr(pywt.Modes, mode_name)
...     cA, cD = pywt.dwt([1,2,1,5,-1,8,4,6], 'db2', mode)
...     print("Mode: %d (%s)" % (mode, mode_name))
...     print("cA: " + format_array(cA))
...     print("cD: " + format_array(cD))
...     print("Reconstruction: " + format_array(
...         pywt.idwt(cA, cD, 'db2', mode)))
Mode: 0 (zero)
cA: [-0.03468  1.73309  3.40612  6.32929  6.95095]
cD: [-0.12941 -2.156   -5.95035 -1.21545 -1.8625 ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 2 (constant)
cA: [ 1.2848   1.73309  3.40612  6.32929  7.51936]
cD: [-0.48296 -2.156   -5.95035 -1.21545  0.25882]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 1 (symmetric)
cA: [ 1.76777  1.73309  3.40612  6.32929  7.77817]
cD: [-0.61237 -2.156   -5.95035 -1.21545  1.22474]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 4 (periodic)
cA: [ 6.91627  1.73309  3.40612  6.32929  6.91627]
cD: [-1.99191 -2.156   -5.95035 -1.21545 -1.99191]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 3 (smooth)
cA: [-0.51764  1.73309  3.40612  6.32929  7.45001]
cD: [ 0.      -2.156   -5.95035 -1.21545  0.     ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 5 (periodization)
cA: [ 4.05317  3.05257  2.85381  8.42522]
cD: [ 0.18947  4.18258  4.33738  2.60428]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
```

The default mode is *symmetric*:

```
>>> cA, cD = pywt.dwt(x, 'db2')
>>> print(cA)
[ 1.76776695  1.73309178  3.40612438  6.32928585  7.77817459]
>>> print(cD)
[-0.61237244 -2.15599552 -5.95034847 -1.21545369  1.22474487]
>>> print(pywt.idwt(cA, cD, 'db2'))
[ 1.  2.  1.  5. -1.  8.  4.  6.]
```

And using a keyword argument:

```
>>> cA, cD = pywt.dwt(x, 'db2', mode='symmetric')
>>> print(cA)
[ 1.76776695  1.73309178  3.40612438  6.32928585  7.77817459]
>>> print(cD)
[-0.61237244 -2.15599552 -5.95034847 -1.21545369  1.22474487]
>>> print(pywt.idwt(cA, cD, 'db2'))
[ 1.  2.  1.  5. -1.  8.  4.  6.]
```

10.2.3 DWT and IDWT

Discrete Wavelet Transform

Let's do a `Discrete Wavelet Transform` of a sample data *x* using the `db2` wavelet. It's simple..

```
>>> import pywt
>>> x = [3, 7, 1, 1, -2, 5, 4, 6]
>>> cA, cD = pywt.dwt(x, 'db2')
```

And the approximation and details coefficients are in `cA` and `cD` respectively:

```
>>> print(cA)
[ 5.65685425  7.39923721  0.22414387  3.33677403  7.77817459]
>>> print(cD)
[-2.44948974 -1.60368225 -4.44140056 -0.41361256  1.22474487]
```

Inverse Discrete Wavelet Transform

Now let's do an opposite operation - *`Inverse Discrete Wavelet Transform`*:

```
>>> print(pywt.idwt(cA, cD, 'db2'))
[ 3.  7.  1.  1. -2.  5.  4.  6.]
```

Voilà! That's it!

More Examples

Now let's experiment with the *`dwt()`* some more. For example let's pass a *`Wavelet`* object instead of the wavelet name and specify signal extension mode (the default is *symmetric*) for the border effect handling:

```
>>> w = pywt.Wavelet('sym3')
>>> cA, cD = pywt.dwt(x, wavelet=w, mode='constant')
>>> print(cA)
[ 4.38354585  3.80302657  7.31813271 -0.58565539  4.09727044  7.81994027]
>>> print(cD)
[-1.33068221 -2.78795192 -3.16825651 -0.67715519 -0.09722957 -0.07045258]
```

Note that the output coefficients arrays length depends not only on the input data length but also on the :class:Wavelet type (particularly on its `filters lenght` that are used in the transformation).

To find out what will be the output data size use the *`dwt_coeff_len()`* function:

```
>>> # int() is for normalizing Python integers and long integers for documentation tests
>>> int(pywt.dwt_coeff_len(data_len=len(x), filter_len=w.dec_len, mode='symmetric'))
6
>>> int(pywt.dwt_coeff_len(len(x), w, 'symmetric'))
6
>>> len(cA)
6
```

Looks fine. (And if you expected that the output length would be a half of the input data length, well, that's the trade-off that allows for the perfect reconstruction...).

The third argument of the *`dwt_coeff_len()`* is the already mentioned signal extension mode (please refer to the PyWavelets' documentation for the *modes* description). Currently there are six *extension modes* available:

```
>>> pywt.Modes.modes
['zero', 'constant', 'symmetric', 'periodic', 'smooth', 'periodization']
```

```
>>> [int(pywt.dwt_coeff_len(len(x), w.dec_len, mode)) for mode in pywt.Modes.modes]
[6, 6, 6, 6, 6, 4]
```

As you see in the above example, the *periodization* (periodization) mode is slightly different from the others. It's aim when doing the *`DWT`* transform is to output coefficients arrays that are half of the length of the input data.

Knowing that, you should never mix the periodization mode with other modes when doing *`DWT`* and *`IDWT`*. Otherwise, it will produce **invalid results**:

```
>>> x
[3, 7, 1, 1, -2, 5, 4, 6]
>>> cA, cD = pywt.dwt(x, wavelet=w, mode='periodization')
>>> print(pywt.idwt(cA, cD, 'sym3', 'symmetric')) # invalid mode
[ 1.  1. -2.  5.]
>>> print(pywt.idwt(cA, cD, 'sym3', 'periodization'))
[ 3.  7.  1.  1. -2.  5.  4.  6.]
```

Tips & tricks

Passing `None` instead of coefficients data to `idwt()`

Now some tips & tricks. Passing `None` as one of the coefficient arrays parameters is similar to passing a *zero-filled* array. The results are simply the same:

```
>>> print(pywt.idwt([1,2,0,1], None, 'db2', 'symmetric'))
[ 1.19006969  1.54362308  0.44828774 -0.25881905  0.48296291  0.8365163 ]
```

```
>>> print(pywt.idwt([1, 2, 0, 1], [0, 0, 0, 0], 'db2', 'symmetric'))
[ 1.19006969  1.54362308  0.44828774 -0.25881905  0.48296291  0.8365163 ]
```

```
>>> print(pywt.idwt(None, [1, 2, 0, 1], 'db2', 'symmetric'))
[ 0.57769726 -0.93125065  1.67303261 -0.96592583 -0.12940952 -0.22414387]
```

```
>>> print(pywt.idwt([0, 0, 0, 0], [1, 2, 0, 1], 'db2', 'symmetric'))
[ 0.57769726 -0.93125065  1.67303261 -0.96592583 -0.12940952 -0.22414387]
```

Remember that only one argument at a time can be `None`:

```
>>> print(pywt.idwt(None, None, 'db2', 'symmetric'))
Traceback (most recent call last):
...
ValueError: At least one coefficient parameter must be specified.
```

Coefficients data size in `idwt`

When doing the *IDWT* transform, usually the coefficient arrays must have the same size.

```
>>> print(pywt.idwt([1, 2, 3, 4, 5], [1, 2, 3, 4], 'db2', 'symmetric'))
Traceback (most recent call last):
...
ValueError: Coefficients arrays must have the same size.
```

Not every coefficient array can be used in *IDWT*. In the following example the *idwt()* will fail because the input arrays are invalid - they couldn't be created as a result of *DWT*, because the minimal output length for dwt using `db4` wavelet and the *symmetric* mode is `4`, not `3`:

```
>>> pywt.idwt([1,2,4], [4,1,3], 'db4', 'symmetric')
Traceback (most recent call last):
...
ValueError: Invalid coefficient arrays length for specified wavelet. Wavelet and mode must be the sa
```

```
>>> int(pywt.dwt_coeff_len(1, pywt.Wavelet('db4').dec_len, 'symmetric'))
4
```

10.2.4 Multilevel DWT, IDWT and SWT

Multilevel DWT decomposition

```
>>> import pywt
>>> x = [3, 7, 1, 1, -2, 5, 4, 6]
>>> db1 = pywt.Wavelet('db1')
>>> cA3, cD3, cD2, cD1 = pywt.wavedec(x, db1)
>>> print(cA3)
[ 8.83883476]
>>> print(cD3)
[-0.35355339]
>>> print(cD2)
[ 4.  -3.5]
>>> print(cD1)
[-2.82842712  0.          -4.94974747 -1.41421356]
```

```
>>> pywt.dwt_max_level(len(x), db1)
3
```

```
>>> cA2, cD2, cD1 = pywt.wavedec(x, db1, mode='constant', level=2)
```

Multilevel IDWT reconstruction

```
>>> coeffs = pywt.wavedec(x, db1)
>>> print(pywt.waverec(coeffs, db1))
[ 3.  7.  1.  1. -2.  5.  4.  6.]
```

Multilevel SWT decomposition

```
>>> x = [3, 7, 1, 3, -2, 6, 4, 6]
>>> (cA2, cD2), (cA1, cD1) = pywt.swt(x, db1, level=2)
>>> print(cA1)
[ 7.07106781  5.65685425  2.82842712  0.70710678  2.82842712  7.07106781
  7.07106781  6.36396103]
>>> print(cD1)
[-2.82842712  4.24264069 -1.41421356  3.53553391 -5.65685425  1.41421356
 -1.41421356  2.12132034]
>>> print(cA2)
[  7.    4.5   4.    5.5   7.    9.5  10.    8.5]
>>> print(cD2)
[ 3.   3.5  0.  -4.5 -3.   0.5  0.   0.5]
```

```
>>> [(cA2, cD2)] = pywt.swt(cA1, db1, level=1, start_level=1)
>>> print(cA2)
[  7.    4.5   4.    5.5   7.    9.5  10.    8.5]
>>> print(cD2)
[ 3.   3.5  0.  -4.5 -3.   0.5  0.   0.5]
```

```
>>> coeffs = pywt.swt(x, db1)
>>> len(coeffs)
3
>>> pywt.swt_max_level(len(x))
3
```

```
>>> from __future__ import print_function
```

10.2.5 Wavelet Packets

Import pywt

```
>>> import pywt
```

```
>>> def format_array(a):
...     """Consistent array representation across different systems"""
...     import numpy
...     a = numpy.where(numpy.abs(a) < 1e-5, 0, a)
...     return numpy.array2string(a, precision=5, separator=' ', suppress_small=True)
```

Create Wavelet Packet structure

Ok, let's create a sample `WaveletPacket`:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

The input *data* and decomposition coefficients are stored in the `WaveletPacket.data` attribute:

```
>>> print(wp.data)
[1, 2, 3, 4, 5, 6, 7, 8]
```

`Nodes` are identified by `paths`. For the root node the path is `''` and the decomposition level is `0`.

```
>>> print(repr(wp.path))
''
>>> print(wp.level)
0
```

The *maxlevel*, if not given as param in the constructor, is automatically computed:

```
>>> print(wp['ad'].maxlevel)
3
```

Traversing WP tree:

Accessing subnodes:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

First check what is the maximum level of decomposition:

```
>>> print(wp.maxlevel)
3
```

and try accessing subnodes of the WP tree:

- 1st level:

```
>>> print(wp['a'].data)
[  2.12132034   4.94974747   7.77817459  10.60660172]
>>> print(wp['a'].path)
a
```

- 2nd level:

```
>>> print(wp['aa'].data)
[  5.  13.]
>>> print(wp['aa'].path)
aa
```

- 3rd level:

```
>>> print(wp['aaa'].data)
[ 12.72792206]
>>> print(wp['aaa'].path)
aaa
```

Ups, we have reached the maximum level of decomposition and got an `IndexError`:

```
>>> print(wp['aaaa'].data)
Traceback (most recent call last):
...
IndexError: Path length is out of range.
```

Now try some invalid path:

```
>>> print(wp['ac'])
Traceback (most recent call last):
...
ValueError: Subnode name must be in ['a', 'd'], not 'c'.
```

which just yielded a `ValueError`.

Accessing Node's attributes:

`WaveletPacket` object is a tree data structure, which evaluates to a set of *`Node`* objects. *`WaveletPacket`* is just a special subclass of the *`Node`* class (which in turn inherits from the *`BaseNode`*).

Tree nodes can be accessed using the *obj[x]* (`Node.__getitem__()`) operator. Each tree node has a set of attributes: `data`, `path`, *`node_name`*, `parent`, `level`, `maxlevel` and `mode`.

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

```
>>> print(wp['ad'].data)
[-2. -2.]
```

```
>>> print(wp['ad'].path)
ad
```

```
>>> print(wp['ad'].node_name)
d
```

```
>>> print(wp['ad'].parent.path)
a
```

```
>>> print(wp['ad'].level)
2
```

```
>>> print(wp['ad'].maxlevel)
3
```

```
>>> print(wp['ad'].mode)
symmetric
```

Collecting nodes

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

We can get all nodes on the particular level either in `natural` order:

```
>>> print([node.path for node in wp.get_level(3, 'natural')])
['aaa', 'aad', 'ada', 'add', 'daa', 'dad', 'dda', 'ddd']
```

or sorted based on the band frequency (`freq`):

```
>>> print([node.path for node in wp.get_level(3, 'freq')])
['aaa', 'aad', 'add', 'ada', 'dda', 'ddd', 'dad', 'daa']
```

Note that *`WaveletPacket.get_level()`* also performs automatic decomposition until it reaches the specified *level*.

Reconstructing data from Wavelet Packets:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

Now create a new *`Wavelet Packet`* and set its nodes with some data.

```
>>> new_wp = pywt.WaveletPacket(data=None, wavelet='db1', mode='symmetric')
```

```
>>> new_wp['aa'] = wp['aa'].data
>>> new_wp['ad'] = [-2., -2.]
```

For convenience, `Node.data` gets automatically extracted from the *`Node`* object:

```
>>> new_wp['d'] = wp['d']
```

And reconstruct the data from the `aa`, `ad` and `d` packets.

```
>>> print(new_wp.reconstruct(update=False))
[ 1.  2.  3.  4.  5.  6.  7.  8.]
```

If the *update* param in the reconstruct method is set to `False`, the node's `data` will not be updated.

```
>>> print(new_wp.data)
None
```

Otherwise, the `data` attribute will be set to the reconstructed value.

```
>>> print(new_wp.reconstruct(update=True))
[ 1.  2.  3.  4.  5.  6.  7.  8.]
>>> print(new_wp.data)
[ 1.  2.  3.  4.  5.  6.  7.  8.]
```

```
>>> print([n.path for n in new_wp.get_leaf_nodes(False)])
['aa', 'ad', 'd']
```

```
>>> print([n.path for n in new_wp.get_leaf_nodes(True)])
['aaa', 'aad', 'ada', 'add', 'daa', 'dad', 'dda', 'ddd']
```

Removing nodes from Wavelet Packet tree:

Let's create a sample data:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

First, start with a tree decomposition at level 2. Leaf nodes in the tree are:

```
>>> dummy = wp.get_level(2)
>>> for n in wp.get_leaf_nodes(False):
...     print(n.path, format_array(n.data))
aa [  5.  13.]
ad [-2. -2.]
da [-1. -1.]
dd [ 0.  0.]
```

```
>>> node = wp['ad']
>>> print(node)
ad: [-2. -2.]
```

To remove a node from the WP tree, use Python's *del obj[x]* (`Node.__delitem__`):

```
>>> del wp['ad']
```

The leaf nodes that left in the tree are:

```
>>> for n in wp.get_leaf_nodes():
...     print(n.path, format_array(n.data))
aa [  5.  13.]
da [-1. -1.]
dd [ 0.  0.]
```

And the reconstruction is:

```
>>> print(wp.reconstruct())
[ 2.  3.  2.  3.  6.  7.  6.  7.]
```

Now restore the deleted node value.

```
>>> wp['ad'].data = node.data
```

Printing leaf nodes and tree reconstruction confirms the original state of the tree:

```
>>> for n in wp.get_leaf_nodes(False):
...     print(n.path, format_array(n.data))
aa [  5.  13.]
ad [-2. -2.]
```

```
da [-1. -1.]
dd [ 0.  0.]
```

```
>>> print(wp.reconstruct())
[ 1.  2.  3.  4.  5.  6.  7.  8.]
```

Lazy evaluation:

Note: This section is for demonstration of pywt internals purposes only. Do not rely on the attribute access to nodes as presented in this example.

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

1. At first the wp's attribute *a* is None

   ```
   >>> print(wp.a)
   None
   ```

   Remember that you should not rely on the attribute access.

2. At first attempt to access the node it is computed via decomposition of its parent node (the wp object itself).

   ```
   >>> print(wp['a'])
   a: [  2.12132034   4.94974747   7.77817459  10.60660172]
   ```

3. Now the *wp.a* is set to the newly created node:

   ```
   >>> print(wp.a)
   a: [  2.12132034   4.94974747   7.77817459  10.60660172]
   ```

   And so is *wp.d*:

   ```
   >>> print(wp.d)
   d: [-0.70710678 -0.70710678 -0.70710678 -0.70710678]
   ```

10.2.6 2D Wavelet Packets

Import pywt

```
>>> from __future__ import print_function
>>> import pywt
>>> import numpy
```

Create 2D Wavelet Packet structure

Start with preparing test data:

```
>>> x = numpy.array([[1, 2, 3, 4, 5, 6, 7, 8]] * 8, 'd')
>>> print(x)
[[ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
```

```
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]]
```

Now create a `2D Wavelet Packet` object:

```
>>> wp = pywt.WaveletPacket2D(data=x, wavelet='db1', mode='symmetric')
```

The input *data* and decomposition coefficients are stored in the `WaveletPacket2D.data` attribute:

```
>>> print(wp.data)
[[ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]]
```

`Nodes` are identified by paths. For the root node the path is `''` and the decomposition level is `0`.

```
>>> print(repr(wp.path))
''
>>> print(wp.level)
0
```

The `WaveletPacket2D.maxlevel`, if not given in the constructor, is automatically computed based on the data size:

```
>>> print(wp.maxlevel)
3
```

Traversing WP tree:

Wavelet Packet `nodes` are arranged in a tree. Each node in a WP tree is uniquely identified and addressed by a `path` string.

In the 1D `WaveletPacket` case nodes were accessed using `'a'` (approximation) and `'d'` (details) path names (each node has two 1D children).

Because now we deal with a bit more complex structure (each node has four children), we have four basic path names based on the dwt 2D output convention to address the WP2D structure:

- `a` - LL, low-low coefficients
- `h` - LH, low-high coefficients
- `v` - HL, high-low coefficients
- `d` - HH, high-high coefficients

In other words, subnode naming corresponds to the `dwt2()` function output naming convention (as wavelet packet transform is based on the dwt2 transform):

```
                                    -------------------
                                    |        |        |
                                    | cA(LL) | cH(LH) |
                                    |        |        |
(cA, (cH, cV, cD))  <--->           -------------------
```

```
                            |         |         |
                            | cV(HL)  | cD(HH)  |
                            |         |         |
                            -------------------

   (fig.1: DWT 2D output and interpretation)
```

Knowing what the nodes names are, we can now access them using the indexing operator *obj[x]* (`WaveletPacket2D.__getitem__()`):

```
>>> print(wp['a'].data)
[[  3.    7.   11.   15.]
 [  3.    7.   11.   15.]
 [  3.    7.   11.   15.]
 [  3.    7.   11.   15.]]
>>> print(wp['h'].data)
[[ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]]
>>> print(wp['v'].data)
[[-1. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1. -1.]]
>>> print(wp['d'].data)
[[ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]]
```

Similarly, a subnode of a subnode can be accessed by:

```
>>> print(wp['aa'].data)
[[ 10.   26.]
 [ 10.   26.]]
```

Indexing base `WaveletPacket2D` (as well as 1D `WaveletPacket`) using compound path is just the same as indexing WP subnode:

```
>>> node = wp['a']
>>> print(node['a'].data)
[[ 10.   26.]
 [ 10.   26.]]
>>> print(wp['a']['a'].data is wp['aa'].data)
True
```

Following down the decomposition path:

```
>>> print(wp['aaa'].data)
[[ 36.]]
>>> print(wp['aaaa'].data)
Traceback (most recent call last):
...
IndexError: Path length is out of range.
```

Ups, we have reached the maximum level of decomposition for the `'aaaa'` path, which btw. was:

```
>>> print(wp.maxlevel)
3
```

Now try some invalid path:

```
>>> print(wp['f'])
Traceback (most recent call last):
...
ValueError: Subnode name must be in ['a', 'h', 'v', 'd'], not 'f'.
```

Accessing Node2D's attributes:

`WaveletPacket2D` is a tree data structure, which evaluates to a set of `Node2D` objects. `WaveletPacket2D` is just a special subclass of the `Node2D` class (which in turn inherits from a `BaseNode`, just like with `Node` and `WaveletPacket` for the 1D case.).

```
>>> print(wp['av'].data)
[[-4. -4.]
 [-4. -4.]]
```

```
>>> print(wp['av'].path)
av
```

```
>>> print(wp['av'].node_name)
v
```

```
>>> print(wp['av'].parent.path)
a
```

```
>>> print(wp['av'].parent.data)
[[  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]]
```

```
>>> print(wp['av'].level)
2
```

```
>>> print(wp['av'].maxlevel)
3
```

```
>>> print(wp['av'].mode)
symmetric
```

Collecting nodes

We can get all nodes on the particular level using the `WaveletPacket2D.get_level()` method:

- 0 level - the root *wp* node:

```
    >>> len(wp.get_level(0))
    1
    >>> print([node.path for node in wp.get_level(0)])
    ['']
```

- 1st level of decomposition:

```
>>> len(wp.get_level(1))
4
>>> print([node.path for node in wp.get_level(1)])
['a', 'h', 'v', 'd']
```

- 2nd level of decomposition:

```
>>> len(wp.get_level(2))
16
>>> paths = [node.path for node in wp.get_level(2)]
>>> for i, path in enumerate(paths):
...     if (i+1) % 4 == 0:
...         print(path)
...     else:
...         print(path, end=' ')
aa ah av ad
ha hh hv hd
va vh vv vd
da dh dv dd
```

- 3rd level of decomposition:

```
>>> print(len(wp.get_level(3)))
64
>>> paths = [node.path for node in wp.get_level(3)]
>>> for i, path in enumerate(paths):
...     if (i+1) % 8 == 0:
...         print(path)
...     else:
...         print(path, end=' ')
aaa aah aav aad aha ahh ahv ahd
ava avh avv avd ada adh adv add
haa hah hav had hha hhh hhv hhd
hva hvh hvv hvd hda hdh hdv hdd
vaa vah vav vad vha vhh vhv vhd
vva vvh vvv vvd vda vdh vdv vdd
daa dah dav dad dha dhh dhv dhd
dva dvh dvv dvd dda ddh ddv ddd
```

Note that `WaveletPacket2D.get_level()` performs automatic decomposition until it reaches the given level.

Reconstructing data from Wavelet Packets:

Let's create a new empty 2D Wavelet Packet structure and set its nodes values with known data from the previous examples:

```
>>> new_wp = pywt.WaveletPacket2D(data=None, wavelet='db1', mode='symmetric')
```

```
>>> new_wp['vh'] = wp['vh'].data # [[0.0, 0.0], [0.0, 0.0]]
>>> new_wp['vv'] = wp['vh'].data # [[0.0, 0.0], [0.0, 0.0]]
>>> new_wp['vd'] = [[0.0, 0.0], [0.0, 0.0]]
```

```
>>> new_wp['a'] = [[3.0, 7.0, 11.0, 15.0], [3.0, 7.0, 11.0, 15.0],
...                [3.0, 7.0, 11.0, 15.0], [3.0, 7.0, 11.0, 15.0]]
>>> new_wp['d'] = [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0],
...                [0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]]
```

For convenience, `Node2D.data` gets automatically extracted from the base `Node2D` object:

```
    >>> new_wp['h'] = wp['h'] # all zeros
```

Note: just remember to not assign to the node.data parameter directly (todo).

And reconstruct the data from the `a`, `d`, `vh`, `vv`, `vd` and `h` packets (Note that `va` node was not set and the WP tree is "not complete" - the `va` branch will be treated as *zero-array*):

```
>>> print(new_wp.reconstruct(update=False))
[[ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]]
```

Now set the `va` node with the known values and do the reconstruction again:

```
>>> new_wp['va'] = wp['va'].data # [[-2.0, -2.0], [-2.0, -2.0]]
>>> print(new_wp.reconstruct(update=False))
[[ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]]
```

which is just the same as the base sample data *x*.

Of course we can go the other way and remove nodes from the tree. If we delete the `va` node, again, we get the "not complete" tree from one of the previous examples:

```
>>> del new_wp['va']
>>> print(new_wp.reconstruct(update=False))
[[ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]]
```

Just restore the node before next examples.

```
>>> new_wp['va'] = wp['va'].data
```

If the *update* param in the `WaveletPacket2D.reconstruct()` method is set to `False`, the node's `Node2D.data` attribute will not be updated.

```
>>> print(new_wp.data)
None
```

Otherwise, the `WaveletPacket2D.data` attribute will be set to the reconstructed value.

```
>>> print(new_wp.reconstruct(update=True))
[[ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]]
>>> print(new_wp.data)
[[ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]]
```

Since we have an interesting WP structure built, it is a good occasion to present the `WaveletPacket2D.get_leaf_nodes()` method, which collects non-zero leaf nodes from the WP tree:

```
>>> print([n.path for n in new_wp.get_leaf_nodes()])
['a', 'h', 'va', 'vh', 'vv', 'vd', 'd']
```

Passing the *decompose=True* parameter to the method will force the WP object to do a full decomposition up to the *maximum level* of decomposition:

```
>>> paths = [n.path for n in new_wp.get_leaf_nodes(decompose=True)]
>>> len(paths)
64
>>> for i, path in enumerate(paths):
...     if (i+1) % 8 == 0:
...         print(path)
...     else:
...         try:
...             print(path, end=' ')
...         except:
...             print(path, end=' ')
aaa aah aav aad aha ahh ahv ahd
ava avh avv avd ada adh adv add
haa hah hav had hha hhh hhv hhd
hva hvh hvv hvd hda hdh hdv hdd
vaa vah vav vad vha vhh vhv vhd
vva vvh vvv vvd vda vdh vdv vdd
daa dah dav dad dha dhh dhv dhd
dva dvh dvv dvd dda ddh ddv ddd
```

Lazy evaluation:

Note: This section is for demonstration of pywt internals purposes only. Do not rely on the attribute access to nodes as presented in this example.

```
>>> x = numpy.array([[1, 2, 3, 4, 5, 6, 7, 8]] * 8)
>>> wp = pywt.WaveletPacket2D(data=x, wavelet='db1', mode='symmetric')
```

1. At first the wp's attribute *a* is `None`

```
>>> print(wp.a)
None
```

Remember that you should not rely on the attribute access.

2. During the first attempt to access the node it is computed via decomposition of its parent node (the wp object itself).

```
>>> print(wp['a'])
a: [[  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]]
```

3. Now the *a* is set to the newly created node:

```
>>> print(wp.a)
a: [[  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]]
```

And so is *wp.d*:

```
>>> print(wp.d)
d: [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

10.2.7 Gotchas

PyWavelets utilizes `NumPy` under the hood. That's why handling the data containing `None` values can be surprising. `None` values are converted to 'not a number' (`numpy.NaN`) values:

```
>>> import numpy, pywt
>>> x = [None, None]
>>> mode = 'symmetric'
>>> wavelet = 'db1'
>>> cA, cD = pywt.dwt(x, wavelet, mode)
>>> numpy.all(numpy.isnan(cA))
True
>>> numpy.all(numpy.isnan(cD))
True
>>> rec = pywt.idwt(cA, cD, wavelet, mode)
>>> numpy.all(numpy.isnan(rec))
True
```

10.3 Development notes

This section contains information on building and installing PyWavelets from source code as well as instructions for preparing the build environment on Windows and Linux.

10.3.1 Preparing Windows build environment

To start developing PyWavelets code on Windows you will have to install a C compiler and prepare the build environment.

Installing Windows SDK C/C++ compiler

Depending on your Python version, a different version of the Microsoft Visual C++ compiler will be required to build extensions. The same compiler that was used to build Python itself should be used.

For official binary builds of Python 2.6 to 3.2, this will be VS 2008. Python 3.3 and 3.4 were compiled with VS 2010, and for Python 3.5 it will be MSVC 2015.

The MSVC version should be printed when starting a Python REPL, and can be checked against the note below:

Note: For reference:

- the *MSC v.1500* in the Python version string is Microsoft Visual C++ 2008 (Microsoft Visual Studio 9.0 with msvcr90.dll runtime)
- *MSC v.1600* is MSVC 2010 (10.0 with msvcr100.dll runtime)
- *MSC v.1700* is MSVC 2012 (11.0)
- *MSC v.1800* is MSVC 2013 (12.0)
- *MSC v.1900* is MSVC 2015 (14.0)

```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win32
Python 3.2 (r32:88445, Feb 20 2011, 21:30:00) [MSC v.1500 64 bit (AMD64)] on win32
```

To get started first download, extract and install *Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1* from http://www.microsoft.com/downloads/en/details.aspx?familyid=71DEB800-C591-4F97-A900-BEA146E4FAE1&displaylang=en.

There are several ISO images on the site, so just grab the one that is suitable for your platform:

- `GRMSDK_EN_DVD.iso` for 32-bit x86 platform
- `GRMSDKX_EN_DVD.iso` for 64-bit AMD64 platform (AMD64 is the codename for 64-bit CPU architecture, not the processor manufacturer)

After installing the SDK and before compiling the extension you have to configure some environment variables.

For 32-bit build execute the `util/setenv_build32.bat` script in the cmd window:

```
    rem Configure the environment for 32-bit builds.
    rem Use "vcvars32.bat" for a 32-bit build.
    "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat"
    rem Convince setup.py to use the SDK tools.
    set MSSdk=1
    setenv /x86 /release
    set DISTUTILS_USE_SDK=1
```

For 64-bit use `util/setenv_build64.bat`:

```
    rem Configure the environment for 64-bit builds.
    rem Use "vcvars32.bat" for a 32-bit build.
    "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\vcvars64.bat"
    rem Convince setup.py to use the SDK tools.
    set MSSdk=1
    setenv /x64 /release
    set DISTUTILS_USE_SDK=1
```

See also http://wiki.cython.org/64BitCythonExtensionsOnWindows.

MinGW C/C++ compiler

MinGW distribution can be downloaded from http://sourceforge.net/projects/mingwbuilds/.

In order to change the settings and use MinGW as the default compiler, edit or create a Distutils configuration file `c:\Python2*\Lib\distutils\distutils.cfg` and place the following entry in it:

```
[build]
compiler = mingw32
```

You can also take a look at Cython's "Installing MinGW on Windows" page at http://wiki.cython.org/InstallingOnWindows for more info.

Note: Python 2.7/3.2 distutils package is incompatible with the current version (4.7+) of MinGW (MinGW dropped the `-mno-cygwin` flag, which is still passed by distutils).

To use MinGW to compile Python extensions you have to patch the `distutils/cygwinccompiler.py` library module and remove every occurrence of `-mno-cygwin`.

See http://bugs.python.org/issue12641 bug report for more information on the issue.

Next steps

After completing these steps continue with *Installing build dependencies*.

10.3.2 Preparing Linux build environment

There is a good chance that you already have a working build environment. Just skip steps that you don't need to execute.

Installing basic build tools

Note that the example below uses `aptitude` package manager, which is specific to Debian and Ubuntu Linux distributions. Use your favourite package manager to install these packages on your OS.

```
aptitude install build-essential gcc python-dev git-core
```

Next steps

After completing these steps continue with *Installing build dependencies*.

10.3.3 Installing build dependencies

Setting up Python virtual environment

A good practice is to create a separate Python virtual environment for each project. If you don't have virtualenv yet, install and activate it using:

```
curl -O https://raw.github.com/pypa/virtualenv/master/virtualenv.py
python virtualenv.py <name_of_the_venv>
. <name_of_the_venv>/bin/activate
```

Installing Cython

Use `pip` (http://pypi.python.org/pypi/pip) to install Cython:

```
pip install Cython>=0.16
```

Installing numpy

Use `pip` to install numpy:

```
pip install numpy
```

It takes some time to compile numpy, so it might be more convenient to install it from a binary release.

Note: Installing numpy in a virtual environment on Windows is not straightforward.

It is recommended to download a suitable binary `.exe` release from http://www.scipy.org/Download/ and install it using `easy_install` (i.e. `easy_install numpy-1.6.2-win32-superpack-python2.7.exe`).

Note: You can find binaries for 64-bit Windows on http://www.lfd.uci.edu/~gohlke/pythonlibs/.

Installing Sphinx

Sphinx is a documentation tool that converts reStructuredText files into nicely looking html documentation. Install it with:

```
pip install Sphinx
```

numpydoc is used to format the API docmentation appropriately. Install it via:

```
pip install numpydoc
```

10.3.4 Building and installing PyWavelets

Installing from source code

Go to https://github.com/PyWavelets/pywt GitHub project page, fork and clone the repository or use the upstream repository to get the source code:

```
git clone https://github.com/PyWavelets/pywt.git PyWavelets
```

Activate your Python virtual environment, go to the cloned source directory and type the following commands to build and install the package:

```
python setup.py build
python setup.py install
```

To verify the installation run the following command:

```
python setup.py test
```

To build docs:

```
cd doc
make html
```

Installing a development version

You can also install directly from the source repository:

```
pip install -e git+https://github.com/PyWavelets/pywt.git#egg=PyWavelets
```

or:

```
pip install PyWavelets==dev
```

Installing a regular release from PyPi

A regular release can be installed with pip or easy_install:

```
pip install PyWavelets
```

10.3.5 Testing

Continous integration with Travis-CI

The project is using Travis-CI service for continous integration and testing.

Current build status is: If you are submitting a patch or pull request please make sure it does not break the build.

Running tests locally

Tests are implemented with nose, so use one of:

$ nosetests pywt

```
    >>> pywt.test()
```

Running tests with Tox

There's also a config file for running tests with Tox (`pip install tox`). To for example run tests for Python 2.7 and Python 3.4 use:

```
tox -e py27,py34
```

For more information see the Tox documentation.

10.3.6 Something not working?

If these instructions are not clear or you need help setting up your development environment, go ahead and ask on the PyWavelets discussion group at http://groups.google.com/group/pywavelets or open a ticket on GitHub.

10.4 Resources

10.4.1 Code

The GitHub repository is now the main code repository.

If you are using the Mercurial repository at Bitbucket, please switch to Git/GitHub and follow for development updates.

10.4.2 Questions and bug reports

Use GitHub Issues or PyWavelets discussions group to post questions and open tickets.

10.4.3 Wavelet Properties Browser

Browse properties and graphs of wavelets included in PyWavelets on wavelets.pybytes.com.

10.4.4 Articles

- Denoising: wavelet thresholding
- Wavelet Regression in Python

10.5 PyWavelets

10.5.1 API Reference

Wavelets

Wavelet `families()`

`pywt.`**`families`**`(`*short=True*`)`

Returns a list of available built-in wavelet families.

Currently the built-in families are:

- Haar (`haar`)
- Daubechies (`db`)
- Symlets (`sym`)
- Coiflets (`coif`)
- Biorthogonal (`bior`)
- Reverse biorthogonal (`rbio`)

- *“Discrete”* FIR approximation of Meyer wavelet (`dmey`)

Parameters **short** : bool, optional

Use short names (default: True).

Returns **families** : list

List of available wavelet families.

Examples

```
>>> import pywt
>>> pywt.families()
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey']
>>> pywt.families(short=False)
['Haar', 'Daubechies', 'Symlets', 'Coiflets', 'Biorthogonal', 'Reverse biorthogonal', 'Discrete
```

Built-in wavelets - `wavelist()`

`pywt.`**`wavelist`**(*family=None*)

Returns list of available wavelet names for the given family name.

Parameters **family** : {‘haar’, ‘db’, ‘sym’, ‘coif’, ‘bior’, ‘rbio’, ‘dmey’}

Short family name. If the family name is None (default) then names of all the built-in wavelets are returned. Otherwise the function returns names of wavelets that belong to the given family.

Returns **wavelist** : list

List of available wavelet names

Examples

```
>>> import pywt
>>> pywt.wavelist('coif')
['coif1', 'coif2', 'coif3', 'coif4', 'coif5']
```

Custom user wavelets are also supported through the `Wavelet` object constructor as described below.

`Wavelet` object

class `pywt.`**`Wavelet`**(*name*[, *filter_bank=None*])

Describes properties of a wavelet identified by the specified wavelet *name*. In order to use a built-in wavelet the *name* parameter must be a valid wavelet name from the `pywt.wavelist()` list.

Custom Wavelet objects can be created by passing a user-defined filters set with the *filter_bank* parameter.

Parameters

- **`name`** – Wavelet name
- **`filter_bank`** – Use a user supplied filter bank instead of a built-in `Wavelet`.

The filter bank object can be a list of four filters coefficients or an object with `filter_bank` attribute, which returns a list of such filters in the following order:

```
[dec_lo, dec_hi, rec_lo, rec_hi]
```

Wavelet objects can also be used as a base filter banks. See section on *using custom wavelets* for more information.

Example:

```
>>> import pywt
>>> wavelet = pywt.Wavelet('db1')
```

`name`
Wavelet name.

`short_name`
Short wavelet name.

`dec_lo`
Decomposition filter values.

`dec_hi`
Decomposition filter values.

`rec_lo`
Reconstruction filter values.

`rec_hi`
Reconstruction filter values.

`dec_len`
Decomposition filter length.

`rec_len`
Reconstruction filter length.

`filter_bank`
Returns filters list for the current wavelet in the following order:

```
[dec_lo, dec_hi, rec_lo, rec_hi]
```

`inverse_filter_bank`
Returns list of reverse wavelet filters coefficients. The mapping from the *filter_coeffs* list is as follows:

```
[rec_lo[::-1], rec_hi[::-1], dec_lo[::-1], dec_hi[::-1]]
```

`short_family_name`
Wavelet short family name

`family_name`
Wavelet family name

`orthogonal`
Set if wavelet is orthogonal

`biorthogonal`
Set if wavelet is biorthogonal

`symmetry`
`asymmetric`, `near symmetric`, `symmetric`

`vanishing_moments_psi`
Number of vanishing moments for the wavelet function

vanishing_moments_phi
Number of vanishing moments for the scaling function

Example:

```
>>> def format_array(arr):
...     return "[%s]" % ", ".join(["%.14f" % x for x in arr])

>>> import pywt
>>> wavelet = pywt.Wavelet('db1')
>>> print(wavelet)
Wavelet db1
  Family name:    Daubechies
  Short name:     db
  Filters length: 2
  Orthogonal:     True
  Biorthogonal:   True
  Symmetry:       asymmetric
>>> print(format_array(wavelet.dec_lo), format_array(wavelet.dec_hi))
[0.70710678118655, 0.70710678118655] [-0.70710678118655, 0.70710678118655]
>>> print(format_array(wavelet.rec_lo), format_array(wavelet.rec_hi))
[0.70710678118655, 0.70710678118655] [0.70710678118655, -0.70710678118655]
```

Approximating wavelet and scaling functions - `Wavelet.wavefun()`

`Wavelet.`**`wavefun`**`(`*level*`)`

Changed in version 0.2: The time (space) localisation of approximation function points was added.

The *`wavefun()`* method can be used to calculate approximations of scaling function (*phi*) and wavelet function (*psi*) at the given level of refinement.

For *`orthogonal`* wavelets returns approximations of scaling function and wavelet function with corresponding x-grid coordinates:

```
[phi, psi, x] = wavelet.wavefun(level)
```

Example:

```
>>> import pywt
>>> wavelet = pywt.Wavelet('db2')
>>> phi, psi, x = wavelet.wavefun(level=5)
```

For other (*`biorthogonal`* but not *`orthogonal`*) wavelets returns approximations of scaling and wavelet function both for decomposition and reconstruction and corresponding x-grid coordinates:

```
[phi_d, psi_d, phi_r, psi_r, x] = wavelet.wavefun(level)
```

Example:

```
>>> import pywt
>>> wavelet = pywt.Wavelet('bior3.5')
>>> phi_d, psi_d, phi_r, psi_r, x = wavelet.wavefun(level=5)
```

See also:

You can find live examples of *`wavefun()`* usage and images of all the built-in wavelets on the Wavelet Properties Browser page.

Using custom wavelets

PyWavelets comes with a `long list` of the most popular wavelets built-in and ready to use. If you need to use a specific wavelet which is not included in the list it is very easy to do so. Just pass a list of four filters or an object with a `filter_bank` attribute as a *filter_bank* argument to the `Wavelet` constructor.

The filters list, either in a form of a simple Python list or returned via the `filter_bank` attribute, must be in the following order:

- lowpass decomposition filter
- highpass decomposition filter
- lowpass reconstruction filter
- highpass reconstruction filter

just as for the `filter_bank` attribute of the `Wavelet` class.

The Wavelet object created in this way is a standard `Wavelet` instance.

The following example illustrates the way of creating custom Wavelet objects from plain Python lists of filter coefficients and a *filter bank-like* objects.

Example:

```
>>> import pywt, math
>>> c = math.sqrt(2)/2
>>> dec_lo, dec_hi, rec_lo, rec_hi = [c, c], [-c, c], [c, c], [c, -c]
>>> filter_bank = [dec_lo, dec_hi, rec_lo, rec_hi]
>>> myWavelet = pywt.Wavelet(name="myHaarWavelet", filter_bank=filter_bank)
>>>
>>> class HaarFilterBank(object):
...     @property
...     def filter_bank(self):
...         c = math.sqrt(2)/2
...         dec_lo, dec_hi, rec_lo, rec_hi = [c, c], [-c, c], [c, c], [c, -c]
...         return [dec_lo, dec_hi, rec_lo, rec_hi]
>>> filter_bank = HaarFilterBank()
>>> myOtherWavelet = pywt.Wavelet(name="myHaarWavelet", filter_bank=filter_bank)
```

Signal extension modes

Because the most common and practical way of representing digital signals in computer science is with finite arrays of values, some extrapolation of the input data has to be performed in order to extend the signal before computing the *Discrete Wavelet Transform* using the cascading filter banks algorithm.

Depending on the extrapolation method, significant artifacts at the signal's borders can be introduced during that process, which in turn may lead to inaccurate computations of the *DWT* at the signal's ends.

PyWavelets provides several methods of signal extrapolation that can be used to minimize this negative effect:

- `zero` - **zero-padding** - signal is extended by adding zero samples:

```
... 0  0 | x1 x2 ... xn | 0  0 ...
```

- `constant` - **constant-padding** - border values are replicated:

```
... x1 x1 | x1 x2 ... xn | xn xn ...
```

- `symmetric` - **symmetric-padding** - signal is extended by *mirroring* samples:

```
... x2 x1 | x1 x2 ... xn | xn xn-1 ...
```

- `periodic` - **periodic-padding** - signal is treated as a periodic one:

```
... xn-1 xn | x1 x2 ... xn | x1 x2 ...
```

- `smooth` - **smooth-padding** - signal is extended according to the first derivatives calculated on the edges (straight line)

DWT performed for these extension modes is slightly redundant, but ensures perfect reconstruction. To receive the smallest possible number of coefficients, computations can be performed with the *periodization* mode:

- `periodization` - **periodization** - is like *periodic-padding* but gives the smallest possible number of decomposition coefficients. *IDWT* must be performed with the same mode.

Example:

```
>>> import pywt
>>> print pywt.Modes.modes
['zero', 'constant', 'symmetric', 'periodic', 'smooth', 'periodization']
```

Notice that you can use any of the following ways of passing wavelet and mode parameters:

```
>>> import pywt
>>> (a, d) = pywt.dwt([1,2,3,4,5,6], 'db2', 'smooth')
>>> (a, d) = pywt.dwt([1,2,3,4,5,6], pywt.Wavelet('db2'), pywt.Modes.smooth)
```

Note: Extending data in context of PyWavelets does not mean reallocation of the data in computer's physical memory and copying values, but rather computing the extra values only when they are needed. This feature saves extra memory and CPU resources and helps to avoid page swapping when handling relatively big data arrays on computers with low physical memory.

Discrete Wavelet Transform (DWT)

Wavelet transform has recently become a very popular when it comes to analysis, de-noising and compression of signals and images. This section describes functions used to perform single- and multilevel Discrete Wavelet Transforms.

Single level `dwt`

`pywt.dwt`(*data*, *wavelet*, *mode='symmetric'*, *axis=-1*)

Single level Discrete Wavelet Transform.

Parameters **data** : array_like

Input signal

wavelet : Wavelet object or name

Wavelet to use

mode : str, optional

Signal extension mode, see Modes

axis: int, optional

Axis over which to compute the DWT. If not given, the last axis is used.

Returns **(cA, cD)** : tuple

Approximation and detail coefficients.

Notes

Length of coefficients arrays depends on the selected mode. For all modes except periodization:

```
len(cA) == len(cD) == floor((len(data) + wavelet.dec_len - 1) / 2)
```

For periodization mode ("per"):

```
len(cA) == len(cD) == ceil(len(data) / 2)
```

Examples

```
>>> import pywt
>>> (cA, cD) = pywt.dwt([1, 2, 3, 4, 5, 6], 'db1')
>>> cA
array([ 2.12132034,  4.94974747,  7.77817459])
>>> cD
array([-0.70710678, -0.70710678, -0.70710678])
```

See the *signal extension modes* section for the list of available options and the `dwt_coeff_len()` function for information on getting the expected result length.

The transform can be performed over one axis of multi-dimensional data. By default this is the last axis. For multi-dimensional transforms see the *2D transforms* section.

Multilevel decomposition using `wavedec`

`pywt.`**`wavedec`**(*data*, *wavelet*, *mode='symmetric'*, *level=None*)

Multilevel 1D Discrete Wavelet Transform of data.

Parameters data: array_like

Input data

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

level : int, optional

Decomposition level (must be >= 0). If level is None (default) then it will be calculated using the `dwt_max_level` function.

Returns [cA_n, cD_n, cD_n-1, ..., cD2, cD1] : list

Ordered list of coefficients arrays where *n* denotes the level of decomposition. The first element (*cA_n*) of the result is approximation coefficients array and the following elements (*cD_n - cD_1*) are details coefficients arrays.

Examples

```
>>> from pywt import wavedec
>>> coeffs = wavedec([1,2,3,4,5,6,7,8], 'db1', level=2)
>>> cA2, cD2, cD1 = coeffs
>>> cD1
array([-0.70710678, -0.70710678, -0.70710678, -0.70710678])
>>> cD2
array([-2., -2.])
>>> cA2
array([  5.,  13.])
```

Partial Discrete Wavelet Transform data decomposition `downcoef`

`pywt.`**`downcoef`**(*part*, *data*, *wavelet*, *mode='symmetric'*, *level=1*)

Partial Discrete Wavelet Transform data decomposition.

Similar to *pywt.dwt*, but computes only one set of coefficients. Useful when you need only approximation or only details at the given level.

Parameters **part** : str

Coefficients type:

- 'a' - approximations reconstruction is performed
- 'd' - details reconstruction is performed

data : array_like

Input signal.

wavelet : Wavelet object or name

Wavelet to use

mode : str, optional

Signal extension mode, see *Modes*. Default is 'symmetric'.

level : int, optional

Decomposition level. Default is 1.

Returns **coeffs** : ndarray

1-D array of coefficients.

See also:

`upcoef`

Maximum decomposition level - `dwt_max_level`

`pywt.`**`dwt_max_level`**(*data_len*, *filter_len*)

Compute the maximum useful level of decomposition.

Parameters **data_len** : int

Input data length.

filter_len : int

Wavelet filter length.

Returns **max_level** : int

Maximum level.

Examples

```
>>> import pywt
>>> w = pywt.Wavelet('sym5')
>>> pywt.dwt_max_level(data_len=1000, filter_len=w.dec_len)
6
>>> pywt.dwt_max_level(1000, w)
6
```

Result coefficients length - `dwt_coeff_len`

`pywt.`**`dwt_coeff_len`**(*data_len*, *filter_len*, *mode='symmetric'*)

Returns length of dwt output for given data length, filter length and mode

Parameters **data_len** : int

Data length.

filter_len : int

Filter length.

mode : str, optional (default: 'symmetric')

Signal extension mode, see Modes

Returns **len** : int

Length of dwt output.

Notes

For all modes except periodization:

```
len(cA) == len(cD) == floor((len(data) + wavelet.dec_len - 1) / 2)
```

for periodization mode ("per"):

```
len(cA) == len(cD) == ceil(len(data) / 2)
```

Based on the given *input data length*, Wavelet *decomposition filter length* and *signal extension mode*, the *`dwt_coeff_len()`* function calculates the length of the resulting coefficients arrays that would be created while performing *`dwt()`* transform.

filter_len can be either an *int* or *`Wavelet`* object for convenience.

Inverse Discrete Wavelet Transform (IDWT)

Single level `idwt`

`pywt.idwt(cA, cD, wavelet, mode='symmetric', axis=-1)`

Single level Inverse Discrete Wavelet Transform.

Parameters **cA** : array_like or None

Approximation coefficients. If None, will be set to array of zeros with same shape as *cD*.

cD : array_like or None

Detail coefficients. If None, will be set to array of zeros with same shape as *cA*.

wavelet : Wavelet object or name

Wavelet to use

mode : str, optional (default: 'symmetric')

Signal extension mode, see Modes

axis: int, optional

Axis over which to compute the inverse DWT. If not given, the last axis is used.

Returns rec: array_like

Single level reconstruction of signal from given coefficients.

Example:

```
>>> import pywt
>>> (cA, cD) = pywt.dwt([1,2,3,4,5,6], 'db2', 'smooth')
>>> print pywt.idwt(cA, cD, 'db2', 'smooth')
array([ 1.,   2.,   3.,   4.,   5.,   6.])
```

One of the neat features of `idwt()` is that one of the *cA* and *cD* arguments can be set to `None`. In that situation the reconstruction will be performed using only the other one. Mathematically speaking, this is equivalent to passing a zero-filled array as one of the arguments.

Example:

```
>>> import pywt
>>> (cA, cD) = pywt.dwt([1,2,3,4,5,6], 'db2', 'smooth')
>>> A = pywt.idwt(cA, None, 'db2', 'smooth')
>>> D = pywt.idwt(None, cD, 'db2', 'smooth')
>>> print A + D
array([ 1.,   2.,   3.,   4.,   5.,   6.])
```

Multilevel reconstruction using `waverec`

`pywt.waverec(coeffs, wavelet, mode='symmetric')`

Multilevel 1D Inverse Discrete Wavelet Transform.

Parameters **coeffs** : array_like

Coefficients list [cAn, cDn, cDn-1, ..., cD2, cD1]

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

Examples

```
>>> import pywt
>>> coeffs = pywt.wavedec([1,2,3,4,5,6,7,8], 'db1', level=2)
>>> pywt.waverec(coeffs, 'db1')
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
```

Direct reconstruction with `upcoef`

pywt.**upcoef**(*part*, *coeffs*, *wavelet*, *level=1*, *take=0*)

Direct reconstruction from coefficients.

Parameters **part** : str

Coefficients type: * 'a' - approximations reconstruction is performed * 'd' - details reconstruction is performed

coeffs : array_like

Coefficients array to recontruct

wavelet : Wavelet object or name

Wavelet to use

level : int, optional

Multilevel reconstruction level. Default is 1.

take : int, optional

Take central part of length equal to 'take' from the result. Default is 0.

Returns **rec** : ndarray

1-D array with reconstructed data from coefficients.

See also:

downcoef

Examples

```
>>> import pywt
>>> data = [1,2,3,4,5,6]
>>> (cA, cD) = pywt.dwt(data, 'db2', 'smooth')
>>> pywt.upcoef('a', cA, 'db2') + pywt.upcoef('d', cD, 'db2')
array([-0.25      , -0.4330127 ,  1.        ,  2.        ,  3.        ,
        4.        ,  5.        ,  6.        ,  1.78589838, -1.03108891])
>>> n = len(data)
>>> pywt.upcoef('a', cA, 'db2', take=n) + pywt.upcoef('d', cD, 'db2', take=n)
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

2D Forward and Inverse Discrete Wavelet Transform

Single level `dwt2`

`pywt.dwt2`(*data*, *wavelet*, *mode='symmetric'*, *axes=(-2, -1)*)

2D Discrete Wavelet Transform.

Parameters **data** : ndarray

2D array with input data

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

axes : 2-tuple of ints, optional

Axes over which to compute the DWT. Repeated elements mean the DWT will be performed multiple times along these axes.

Returns **(cA, (cH, cV, cD))** : tuple

Approximation, horizontal detail, vertical detail and diagonal detail coefficients respectively. Horizontal refers to array axis 0.

Examples

```
>>> import numpy as np
>>> import pywt
>>> data = np.ones((4,4), dtype=np.float64)
>>> coeffs = pywt.dwt2(data, 'haar')
>>> cA, (cH, cV, cD) = coeffs
>>> cA
array([[ 2.,  2.],
       [ 2.,  2.]])
>>> cV
array([[ 0.,  0.],
       [ 0.,  0.]])
```

The relation to the other common data layout where all the approximation and details coefficients are stored in one big 2D array is as follows:

```
                                -------------------
                                |        |        |
                                | cA(LL) | cH(LH) |
                                |        |        |
(cA, (cH, cV, cD))   <--->      -------------------
                                |        |        |
                                | cV(HL) | cD(HH) |
                                |        |        |
                                -------------------
```

PyWavelets does not follow this pattern because of pure practical reasons of simple access to particular type of the output coefficients.

Single level `idwt2`

`pywt.`**`idwt2`**(*coeffs*, *wavelet*, *mode='symmetric'*, *axes=(-2, -1)*)

2-D Inverse Discrete Wavelet Transform.

Reconstructs data from coefficient arrays.

Parameters **coeffs** : tuple

(cA, (cH, cV, cD)) A tuple with approximation coefficients and three details coefficients 2D arrays like from *dwt2()*

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

axes : 2-tuple of ints, optional

Axes over which to compute the IDWT. Repeated elements mean the IDWT will be performed multiple times along these axes.

Examples

```
>>> import numpy as np
>>> import pywt
>>> data = np.array([[1,2], [3,4]], dtype=np.float64)
>>> coeffs = pywt.dwt2(data, 'haar')
>>> pywt.idwt2(coeffs, 'haar')
array([[ 1.,  2.],
       [ 3.,  4.]])
```

2D multilevel decomposition using `wavedec2`

`pywt.`**`wavedec2`**(*data*, *wavelet*, *mode='symmetric'*, *level=None*)

Multilevel 2D Discrete Wavelet Transform.

Parameters **data** : ndarray

2D input data

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

level : int, optional

Decomposition level (must be >= 0). If level is None (default) then it will be calculated using the `dwt_max_level` function.

Returns **[cAn, (cHn, cVn, cDn), ... (cH1, cV1, cD1)]** : list

Coefficients list

Examples

```
>>> import pywt
>>> import numpy as np
>>> coeffs = pywt.wavedec2(np.ones((4,4)), 'db1')
>>> # Levels:
>>> len(coeffs)-1
2
>>> pywt.waverec2(coeffs, 'db1')
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

2D multilevel reconstruction using `waverec2`

`pywt.`**`waverec2`** (*coeffs*, *wavelet*, *mode='symmetric'*)
Multilevel 2D Inverse Discrete Wavelet Transform.

coeffs [list or tuple] Coefficients list [cAn, (cHn, cVn, cDn), ... (cH1, cV1, cD1)]

wavelet [Wavelet object or name string] Wavelet to use

mode [str, optional] Signal extension mode, see Modes (default: 'symmetric')

Returns 2D array of reconstructed data.

Examples

```
>>> import pywt
>>> import numpy as np
>>> coeffs = pywt.wavedec2(np.ones((4,4)), 'db1')
>>> # Levels:
>>> len(coeffs)-1
2
>>> pywt.waverec2(coeffs, 'db1')
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

nD Forward and Inverse Discrete Wavelet Transform

Single level - `dwtn`

`pywt.`**`dwtn`** (*data*, *wavelet*, *mode='symmetric'*, *axes=None*)
Single-level n-dimensional Discrete Wavelet Transform.

Parameters **data** : ndarray

n-dimensional array with input data.

wavelet : Wavelet object or name string

Wavelet to use.

mode : str, optional

Signal extension mode, see *Modes*. Default is 'symmetric'.

axes : sequence of ints, optional

Axes over which to compute the DWT. Repeated elements mean the DWT will be performed multiple times along these axes. A value of *None* (the default) selects all axes.

Axes may be repeated, but information about the original size may be lost if it is not divisible by *2 ** nrepeats*. The reconstruction will be larger, with additional values derived according to the *mode* parameter. *pywt.wavedecn* should be used for multilevel decomposition.

Returns **coeffs** : dict

Results are arranged in a dictionary, where key specifies the transform type on each dimension and value is a n-dimensional coefficients array.

For example, for a 2D case the result will look something like this:

```
{'aa': <coeffs>  # A(LL) - approx. on 1st dim, approx. on 2nd dim
 'ad': <coeffs>  # V(LH) - approx. on 1st dim, det. on 2nd dim
 'da': <coeffs>  # H(HL) - det. on 1st dim, approx. on 2nd dim
 'dd': <coeffs>  # D(HH) - det. on 1st dim, det. on 2nd dim
}
```

Single level - `idwtn`

`pywt.`**`idwtn`**(*coeffs*, *wavelet*, *mode='symmetric'*, *axes=None*)

Single-level n-dimensional Inverse Discrete Wavelet Transform.

Parameters **coeffs: dict**

Dictionary as in output of *dwtn*. Missing or None items will be treated as zeroes.

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode used in the decomposition, see Modes (default: 'symmetric').

axes : sequence of ints, optional

Axes over which to compute the IDWT. Repeated elements mean the IDWT will be performed multiple times along these axes. A value of *None* (the default) selects all axes.

For the most accurate reconstruction, the axes should be provided in the same order as they were provided to *dwtn*.

Returns data: ndarray

Original signal reconstructed from input data.

Multilevel decomposition - `wavedecn`

`pywt.`**`wavedecn`**(*data*, *wavelet*, *mode='symmetric'*, *level=None*)

Multilevel nD Discrete Wavelet Transform.

Parameters **data** : ndarray

nD input data

wavelet : Wavelet object or name string

Wavelet to use

mode : str, optional

Signal extension mode, see Modes (default: 'symmetric')

level : int, optional

Dxecomposition level (must be >= 0). If level is None (default) then it will be calculated using the `dwt_max_level` function.

Returns **[cAn, {details_level_n}, ... {details_level_1}]** : list

Coefficients list

Examples

```
>>> import numpy as np
>>> from pywt import wavedecn, waverecn
>>> coeffs = wavedecn(np.ones((4, 4, 4)), 'db1')
>>> # Levels:
>>> len(coeffs)-1
2
>>> waverecn(coeffs, 'db1')
array([[[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]]])
```

Multilevel reconstruction - `waverecn`

`pywt.`**`waverecn`**(*coeffs*, *wavelet*, *mode='symmetric'*)

Multilevel nD Inverse Discrete Wavelet Transform.

coeffs [array_like] Coefficients list [cAn, {details_level_n}, ... {details_level_1}]

wavelet [Wavelet object or name string] Wavelet to use

mode [str, optional] Signal extension mode, see Modes (default: 'symmetric')

Returns nD array of reconstructed data.

Examples

```
>>> import numpy as np
>>> from pywt import wavedecn, waverecn
>>> coeffs = wavedecn(np.ones((4, 4, 4)), 'db1')
>>> # Levels:
>>> len(coeffs)-1
2
>>> waverecn(coeffs, 'db1')
array([[[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]]])
```

Stationary Wavelet Transform

Stationary Wavelet Transform (SWT), also known as *Undecimated wavelet transform* or *Algorithme à trous* is a translation-invariance modification of the *Discrete Wavelet Transform* that does not decimate coefficients at every transformation level.

Multilevel `swt`

`pywt.swt(data, wavelet, level=None, start_level=0)`

Performs multilevel Stationary Wavelet Transform.

Parameters **data :**

Input signal

wavelet :

Wavelet to use (Wavelet object or name)

level : int, optional

Transform level.

start_level : int, optional

The level at which the decomposition will begin (it allows one to skip a given number of transform steps and compute coefficients starting from start_level) (default: 0)

Returns **coeffs** : list

List of approximation and details coefficients pairs in order similar to wavedec function:

```
[(cAn, cDn), ..., (cA2, cD2), (cA1, cD1)]
```

where `n` equals input parameter *level*.

If *m* = *start_level* is given, then the beginning *m* steps are skipped:

```
[(cAm+n, cDm+n), ..., (cAm+1, cDm+1), (cAm, cDm)]
```

Multilevel `swt2`

`pywt.`**`swt2`**(*data*, *wavelet*, *level*, *start_level=0*)

2D Stationary Wavelet Transform.

Parameters **data** : ndarray

2D array with input data

wavelet : Wavelet object or name string

Wavelet to use

level : int

How many decomposition steps to perform

start_level : int, optional

The level at which the decomposition will start (default: 0)

Returns **coeffs** : list

Approximation and details coefficients:

```
[
    (cA_n,
        (cH_n, cV_n, cD_n)
    ),
    (cA_n+1,
        (cH_n+1, cV_n+1, cD_n+1)
    ),
    ...,
    (cA_n+level,
        (cH_n+level, cV_n+level, cD_n+level)
    )
]
```

where cA is approximation, cH is horizontal details, cV is vertical details, cD is diagonal details and n is start_level.

Maximum decomposition level - `swt_max_level`

`pywt.`**`swt_max_level`**(*input_len*)

Calculates the maximum level of Stationary Wavelet Transform for data of given length.

Parameters **input_len** : int

Input data length.

Returns **max_level** : int

Maximum level of Stationary Wavelet Transform for data of given length.

Wavelet Packets

New in version 0.2.

Version *0.2* of PyWavelets includes many new features and improvements. One of such new feature is a two-dimensional wavelet packet transform structure that is almost completely sharing programming interface with the one-dimensional tree structure.

In order to achieve this simplification, a new inheritance scheme was used in which a `BaseNode` base node class is a superclass for both `Node` and `Node2D` node classes.

The node classes are used as data wrappers and can be organized in trees (binary trees for 1D transform case and quad-trees for the 2D one). They are also superclasses to the `WaveletPacket` class and `WaveletPacket2D` class that are used as the decomposition tree roots and contain a couple additional methods.

The below diagram illustrates the inheritance tree:

- `BaseNode` - common interface for 1D and 2D nodes:
  - `Node` - data carrier node in a 1D decomposition tree
    - `WaveletPacket` - 1D decomposition tree root node
  - `Node2D` - data carrier node in a 2D decomposition tree
    - `WaveletPacket2D` - 2D decomposition tree root node

BaseNode - a common interface of WaveletPacket and WaveletPacket2D

class `pywt.`**`BaseNode`**
class `pywt.`**`Node`**(*BaseNode*)
class `pywt.`**`WaveletPacket`**(*Node*)
class `pywt.`**`Node2D`**(*BaseNode*)
class `pywt.`**`WaveletPacket2D`**(*Node2D*)

Note: The BaseNode is a base class for `Node` and `Node2D`. It should not be used directly unless creating a new transformation type. It is included here to document the common interface of 1D and 2D node an wavelet packet transform classes.

`__init__`(*parent*, *data*, *node_name*)

Parameters

- **`parent`** – parent node. If parent is `None` then the node is considered detached.
- **`data`** – data associated with the node. 1D or 2D numeric array, depending on the transform type.
- **`node_name`** – a name identifying the coefficients type. See `Node.node_name` and `Node2D.node_name` for information on the accepted subnodes names.

`data`
Data associated with the node. 1D or 2D numeric array (depends on the transform type).

`parent`
Parent node. Used in tree navigation. `None` for root node.

wavelet
Wavelet used for decomposition and reconstruction. Inherited from parent node.

mode
Signal extension *mode* for the `dwt()` (`dwt2()`) and `idwt()` (`idwt2()`) decomposition and reconstruction functions. Inherited from parent node.

level
Decomposition level of the current node. `0` for root (original data), `1` for the first decomposition level, etc.

path
Path string defining position of the node in the decomposition tree.

node_name
Node name describing `data` coefficients type of the current subnode.

See `Node.node_name` and `Node2D.node_name`.

maxlevel
Maximum allowed level of decomposition. Evaluated from parent or child nodes.

is_empty
Checks if `data` attribute is `None`.

has_any_subnode
Checks if node has any subnodes (is not a leaf node).

decompose()
Performs Discrete Wavelet Transform on the `data` and returns transform coefficients.

reconstruct([*update=False*])
Performs Inverse Discrete Wavelet Transform on subnodes coefficients and returns reconstructed data for the current level.

Parameters update – If set, the `data` attribute will be updated with the reconstructed value.

Note: Descends to subnodes and recursively calls `reconstruct()` on them.

get_subnode(*part*[, *decompose=True*])
Returns subnode or None (see *decomposition* flag description).

Parameters

- **part** – Subnode name
- **decompose** – If True and subnode does not exist, it will be created using coefficients from the DWT decomposition of the current node.

__getitem__(*path*)
Used to access nodes in the decomposition tree by string *path*.

Parameters path – Path string composed from valid node names. See `Node.node_name` and `Node2D.node_name` for node naming convention.

Similar to `get_subnode()` method with *decompose=True*, but can access nodes on any level in the decomposition tree.

If node does not exist yet, it will be created by decomposition of its parent node.

__setitem__(*path*, *data*)
Used to set node or node's data in the decomposition tree. Nodes are identified by string *path*.

Parameters

- **path** – Path string composed from valid node names. See `Node.node_name` and `Node2D.node_name` for node naming convention.
- **data** – numeric array or `BaseNode` subclass.

`__delitem__`(*path*)
Used to delete node from the decomposition tree.

Parameters path – Path string composed from valid node names. See `Node.node_name` and `Node2D.node_name` for node naming convention.

`get_leaf_nodes`([*decompose=False*])
Traverses through the decomposition tree and collects leaf nodes (nodes without any subnodes).

Parameters decompose – If *decompose* is `True`, the method will try to decompose the tree up to the `maximum level`.

`walk`(*self, func*[, *args=()*[, *kwargs={}*[, *decompose=True*]]])
Traverses the decomposition tree and calls `func(node, *args, **kwargs)` on every node. If *func* returns `True`, descending to subnodes will continue.

Parameters

- **func** – callable accepting `BaseNode` as the first param and optional positional and keyword arguments:

```
func(node, *args, **kwargs)
```

- **decompose** – If *decompose* is `True` (default), the method will also try to decompose the tree up to the `maximum level`.

Args arguments to pass to the *func*

Kwargs keyword arguments to pass to the *func*

`walk_depth`(*self, func*[, *args=()*[, *kwargs={}*[, *decompose=False*]]])
Similar to `walk()` but traverses the tree in depth-first order.

Parameters

- **func** – callable accepting `BaseNode` as the first param and optional positional and keyword arguments:

```
func(node, *args, **kwargs)
```

- **decompose** – If *decompose* is `True`, the method will also try to decompose the tree up to the `maximum level`.

Args arguments to pass to the *func*

Kwargs keyword arguments to pass to the *func*

WaveletPacket and WaveletPacket tree Node

class `pywt.`**`Node`**(*BaseNode*)
class `pywt.`**`WaveletPacket`**(*Node*)

`node_name`
Node name describing `data` coefficients type of the current subnode.

For `WaveletPacket` case it is just as in `dwt()`:

- `a` - approximation coefficients
- `d` - details coefficients

decompose()

See also:

- *dwt()* for 1D Discrete Wavelet Transform output coefficients.

class pywt.**WaveletPacket**(*Node*)

__init__(*data, wavelet*[, *mode='symmetric'*[, *maxlevel=None*]])

Parameters

- **data** – data associated with the node. 1D numeric array.
- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the *wavelist()* list or a *Wavelet* object instance.
- **mode** – Signal extension *mode* for the *dwt()* and *idwt()* decomposition and reconstruction functions.
- **maxlevel** – Maximum allowed level of decomposition. If not specified it will be calculated based on the *wavelet* and *data* length using *pywt.dwt_max_level()*.

get_level(*level*[, *order="natural"*[, *decompose=True*]])

Collects nodes from the given level of decomposition.

Parameters

- **level** – Specifies decomposition *level* from which the nodes will be collected.
- **order** – Specifies nodes order - natural (`natural`) or frequency (`freq`).
- **decompose** – If set then the method will try to decompose the data up to the specified *level*.

If nodes at the given level are missing (i.e. the tree is partially decomposed) and the *decompose* is set to `False`, only existing nodes will be returned.

WaveletPacket2D and WaveletPacket2D tree Node2D

class pywt.**Node2D**(*BaseNode*)

class pywt.**WaveletPacket2D**(*Node2D*)

node_name

For *WaveletPacket2D* case it is just as in *dwt2()*:

- `a` - approximation coefficients (*LL*)
- `h` - horizontal detail coefficients (*LH*)
- `v` - vertical detail coefficients (*HL*)
- `d` - diagonal detail coefficients (*HH*)

decompose()

See also:

`dwt2()` for 2D Discrete Wavelet Transform output coefficients.

expand_2d_path(self, path):

class pywt.**WaveletPacket2D**(*Node2D*)

__init__(*data*, *wavelet*[, *mode='symmetric'*[, *maxlevel=None*]])

Parameters

- **data** – data associated with the node. 2D numeric array.
- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the `wavelist()` list or a `Wavelet` object instance.
- **mode** – Signal extension *mode* for the `dwt()` and `idwt()` decomposition and reconstruction functions.
- **maxlevel** – Maximum allowed level of decomposition. If not specified it will be calculated based on the *wavelet* and *data* length using `pywt.dwt_max_level()`.

get_level(*level*[, *order="natural"*[, *decompose=True*]])

Collects nodes from the given level of decomposition.

Parameters

- **level** – Specifies decomposition *level* from which the nodes will be collected.
- **order** – Specifies nodes order - natural (`natural`) or frequency (`freq`).
- **decompose** – If set then the method will try to decompose the data up to the specified *level*.

If nodes at the given level are missing (i.e. the tree is partially decomposed) and the *decompose* is set to `False`, only existing nodes will be returned.

Thresholding functions

The `thresholding` helper module implements the most popular signal thresholding functions.

Thresholding

pywt.**threshold**(*data*, *value*, *mode='soft'*, *substitute=0*)

Thresholds the input data depending on the mode argument.

In `soft` thresholding, the data values where their absolute value is less than the value param are replaced with substitute. From the data values with absolute value greater or equal to the thresholding value, a quantity of `(signum * value)` is subtracted.

In `hard` thresholding, the data values where their absolute value is less than the value param are replaced with substitute. Data values with absolute value greater or equal to the thresholding value stay untouched.

In `greater` thresholding, the data is replaced with substitute where data is below the thresholding value. Greater data values pass untouched.

In `less` thresholding, the data is replaced with substitute where data is above the thresholding value. Less data values pass untouched.

Parameters **data** : array_like

Numeric data.

value : scalar

Thresholding value.

mode : {'soft', 'hard', 'greater', 'less'}

Decides the type of thresholding to be applied on input data. Default is 'soft'.

substitute : float, optional

Substitute value (default: 0).

Returns **output** : array

Thresholded array.

Examples

```
>>> import numpy as np
>>> import pywt
>>> data = np.linspace(1, 4, 7)
>>> data
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ])
>>> pywt.threshold(data, 2, 'soft')
array([ 0. ,  0. ,  0. ,  0.5,  1. ,  1.5,  2. ])
>>> pywt.threshold(data, 2, 'hard')
array([ 0. ,  0. ,  2. ,  2.5,  3. ,  3.5,  4. ])
>>> pywt.threshold(data, 2, 'greater')
array([ 0. ,  0. ,  2. ,  2.5,  3. ,  3.5,  4. ])
>>> pywt.threshold(data, 2, 'less')
array([ 1. ,  1.5,  2. ,  0. ,  0. ,  0. ,  0. ])
```

Other functions

Integrating wavelet functions

`pywt.`**`integrate_wavelet`**(*wavelet*, *precision=8*)

Integrate *psi* wavelet function from -Inf to x using the rectangle integration method.

Parameters **wavelet** : Wavelet instance or str

Wavelet to integrate. If a string, should be the name of a wavelet.

precision : int, optional

Precision that will be used for wavelet function approximation computed with the wavefun(level=precision) Wavelet's method (default: 8).

Returns [int_psi, x] :

for orthogonal wavelets

[int_psi_d, int_psi_r, x] :

for other wavelets

Examples

```
>>> from pywt import Wavelet, integrate_wavelet
>>> wavelet1 = Wavelet('db2')
>>> [int_psi, x] = integrate_wavelet(wavelet1, precision=5)
>>> wavelet2 = Wavelet('bior1.3')
>>> [int_psi_d, int_psi_r, x] = integrate_wavelet(wavelet2, precision=5)
```

The result of the call depends on the *wavelet* argument:

- for orthogonal and continuous wavelets - an integral of the wavelet function specified on an x-grid:

```
[int_psi, x_grid] = integrate_wavelet(wavelet, precision)
```

- for other wavelets - integrals of decomposition and reconstruction wavelet functions and a corresponding x-grid:

```
[int_psi_d, int_psi_r, x_grid] = integrate_wavelet(wavelet, precision)
```

Central frequency of *psi* wavelet function

`pywt.`**`central_frequency`**(*wavelet*, *precision=8*)

Computes the central frequency of the *psi* wavelet function.

Parameters **wavelet** : Wavelet instance, str or tuple

Wavelet to integrate. If a string, should be the name of a wavelet.

precision : int, optional

Precision that will be used for wavelet function approximation computed with the wavefun(level=precision) Wavelet's method (default: 8).

Returns scalar

`pywt.`**`scale2frequency`**(*wavelet*, *scale*, *precision=8*)

Parameters **wavelet** : Wavelet instance or str

Wavelet to integrate. If a string, should be the name of a wavelet.

scale : scalar

precision : int, optional

Precision that will be used for wavelet function approximation computed with `wavelet.wavefun(level=precision)`. Default is 8.

Returns **freq** : scalar

Quadrature Mirror Filter

`pywt.`**`qmf`**(*filter*)

Returns the Quadrature Mirror Filter(QMF).

The magnitude response of QMF is mirror image about *pi/2* of that of the input filter.

Parameters **filter** : array_like

Input filter for which QMF needs to be computed.

Returns **qm_filter** : ndarray

Quadrature mirror of the input filter.

Orthogonal Filter Banks

pywt.**orthogonal_filter_bank**(*scaling_filter*)

Returns the orthogonal filter bank.

The orthogonal filter bank consists of the HPFs and LPFs at decomposition and reconstruction stage for the input scaling filter.

Parameters **scaling_filter** : array_like

Input scaling filter (father wavelet).

Returns **orth_filt_bank** : tuple of 4 ndarrays

The orthogonal filter bank of the input scaling filter in the order : 1] Decomposition LPF 2] Decomposition HPF 3] Reconstruction LPF 4] Reconstruction HPF

Example Datasets

The following example datasets are available in the module *pywt.data*:

name	description
ecg	ECG waveform (1024 samples)
aero	grayscale image (512x512)
ascent	grayscale image (512x512)
camera	grayscale image (512x512)

Each can be loaded via a function of the same name.

Example: .. sourcecode:: python

```
>>> import pywt
>>> camera = pywt.data.camera()
```

10.5.2 Usage examples

The following examples are used as doctest regression tests written using reST markup. They are included in the documentation since they contain various useful examples illustrating how to use and how not to use PyWavelets.

The Wavelet object

Wavelet families and builtin Wavelets names

`Wavelet` objects are really a handy carriers of a bunch of DWT-specific data like *quadrature mirror filters* and some general properties associated with them.

At first let's go through the methods of creating a `Wavelet` object. The easiest and the most convenient way is to use builtin named Wavelets.

These wavelets are organized into groups called wavelet families. The most commonly used families are:

```
>>> import pywt
>>> pywt.families()
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey']
```

The `wavelist()` function with family name passed as an argument is used to obtain the list of wavelet names in each family.

```
>>> for family in pywt.families():
...     print("%s family: " % family + ', '.join(pywt.wavelist(family)))
haar family: haar
db family: db1, db2, db3, db4, db5, db6, db7, db8, db9, db10, db11, db12, db13, db14, db15, db16, db1
sym family: sym2, sym3, sym4, sym5, sym6, sym7, sym8, sym9, sym10, sym11, sym12, sym13, sym14, sym15,
coif family: coif1, coif2, coif3, coif4, coif5
bior family: bior1.1, bior1.3, bior1.5, bior2.2, bior2.4, bior2.6, bior2.8, bior3.1, bior3.3, bior3.5
rbio family: rbio1.1, rbio1.3, rbio1.5, rbio2.2, rbio2.4, rbio2.6, rbio2.8, rbio3.1, rbio3.3, rbio3.5
dmey family: dmey
```

To get the full list of builtin wavelets' names just use the `wavelist()` with no argument. As you can see currently there are 76 builtin wavelets.

```
>>> len(pywt.wavelist())
76
```

Creating Wavelet objects

Now when we know all the names let's finally create a `Wavelet` object:

```
>>> w = pywt.Wavelet('db3')
```

So.. that's it.

Wavelet properties

But what can we do with `Wavelet` objects? Well, they carry some interesting information.

First, let's try printing a `Wavelet` object. This shows a brief information about its name, its family name and some properties like orthogonality and symmetry.

```
>>> print(w)
Wavelet db3
  Family name:    Daubechies
  Short name:     db
  Filters length: 6
  Orthogonal:     True
  Biorthogonal:   True
  Symmetry:       asymmetric
```

But the most important information are the wavelet filters coefficients, which are used in *Discrete Wavelet Transform*. These coefficients can be obtained via the `dec_lo`, `Wavelet.dec_hi`, `rec_lo` and `rec_hi` attributes, which corresponds to lowpass and highpass decomposition filters and lowpass and highpass reconstruction filters respectively:

```
>>> def print_array(arr):
...     print("[%s]" % ", ".join(["%.14f" % x for x in arr]))
```

```
>>> print_array(w.dec_lo)
[0.03522629188210, -0.08544127388224, -0.13501102001039, 0.45987750211933, 0.80689150931334, 0.33267
>>> print_array(w.dec_hi)
[-0.33267055295096, 0.80689150931334, -0.45987750211933, -0.13501102001039, 0.08544127388224, 0.0352
>>> print_array(w.rec_lo)
[0.33267055295096, 0.80689150931334, 0.45987750211933, -0.13501102001039, -0.08544127388224, 0.03522
```

```
>>> print_array(w.rec_hi)
[0.03522629188210, 0.08544127388224, -0.13501102001039, -0.45987750211933, 0.80689150931334, -0.3326
```

Another way to get the filters data is to use the `filter_bank` attribute, which returns all four filters in a tuple:

```
>>> w.filter_bank == (w.dec_lo, w.dec_hi, w.rec_lo, w.rec_hi)
True
```

Other Wavelet's properties are:

Wavelet `name`, `short_family_name` and `family_name`:

```
>>> print(w.name)
db3
>>> print(w.short_family_name)
db
>>> print(w.family_name)
Daubechies
```

- Decomposition (`dec_len`) and reconstruction (`rec_len`) filter lengths:

```
>>> int(w.dec_len) # int() is for normalizing longs and ints for doctest
6
>>> int(w.rec_len)
6
```

- Orthogonality (`orthogonal`) and biorthogonality (`biorthogonal`):

```
>>> w.orthogonal
True
>>> w.biorthogonal
True
```

- Symmetry (`symmetry`):

```
>>> print(w.symmetry)
asymmetric
```

- Number of vanishing moments for the scaling function *phi* (`vanishing_moments_phi`) and the wavelet function *psi* (`vanishing_moments_psi`) associated with the filters:

```
>>> w.vanishing_moments_phi
0
>>> w.vanishing_moments_psi
3
```

Now when we know a bit about the builtin Wavelets, let's see how to create *custom Wavelets* objects. These can be done in two ways:

1. Passing the filter bank object that implements the *filter_bank* attribute. The attribute must return four filters coefficients.

```
>>> class MyHaarFilterBank(object):
...     @property
...     def filter_bank(self):
...         from math import sqrt
...         return ([sqrt(2)/2, sqrt(2)/2], [-sqrt(2)/2, sqrt(2)/2],
...                 [sqrt(2)/2, sqrt(2)/2], [sqrt(2)/2, -sqrt(2)/2])
```

```
>>> my_wavelet = pywt.Wavelet('My Haar Wavelet', filter_bank=MyHaarFilterBank())
```

2. Passing the filters coefficients directly as the *filter_bank* parameter.

```
>>> from math import sqrt
>>> my_filter_bank = ([sqrt(2)/2, sqrt(2)/2], [-sqrt(2)/2, sqrt(2)/2],
...                   [sqrt(2)/2, sqrt(2)/2], [sqrt(2)/2, -sqrt(2)/2])
>>> my_wavelet = pywt.Wavelet('My Haar Wavelet', filter_bank=my_filter_bank)
```

Note that such custom wavelets **will not** have all the properties set to correct values:

```
>>> print(my_wavelet)
Wavelet My Haar Wavelet
  Family name:
  Short name:
  Filters length: 2
  Orthogonal:     False
  Biorthogonal:   False
  Symmetry:       unknown
```

You can however set a few of them on your own:

```
>>> my_wavelet.orthogonal = True
>>> my_wavelet.biorthogonal = True
```

```
>>> print(my_wavelet)
Wavelet My Haar Wavelet
  Family name:
  Short name:
  Filters length: 2
  Orthogonal:     True
  Biorthogonal:   True
  Symmetry:       unknown
```

And now... the *wavefun*!

We all know that the fun with wavelets is in wavelet functions. Now what would be this package without a tool to compute wavelet and scaling functions approximations?

This is the purpose of the `wavefun()` method, which is used to approximate scaling function (*phi*) and wavelet function (*psi*) at the given level of refinement, based on the filters coefficients.

The number of returned values varies depending on the wavelet's orthogonality property. For orthogonal wavelets the result is tuple with scaling function, wavelet function and xgrid coordinates.

```
>>> w = pywt.Wavelet('sym3')
>>> w.orthogonal
True
>>> (phi, psi, x) = w.wavefun(level=5)
```

For biorthogonal (non-orthogonal) wavelets different scaling and wavelet functions are used for decomposition and reconstruction, and thus five elements are returned: decomposition scaling and wavelet functions approximations, reconstruction scaling and wavelet functions approximations, and the xgrid.

```
>>> w = pywt.Wavelet('bior1.3')
>>> w.orthogonal
False
>>> (phi_d, psi_d, phi_r, psi_r, x) = w.wavefun(level=5)
```

See also:

You can find live examples of *wavefun()* usage and images of all the built-in wavelets on the Wavelet Properties Browser page.

Signal Extension Modes

Import pywt first

```
>>> import pywt
```

```
>>> def format_array(a):
...     """Consistent array representation across different systems"""
...     import numpy
...     a = numpy.where(numpy.abs(a) < 1e-5, 0, a)
...     return numpy.array2string(a, precision=5, separator=' ', suppress_small=True)
```

List of available signal extension *modes*:

```
>>> print(pywt.Modes.modes)
['zero', 'constant', 'symmetric', 'periodic', 'smooth', 'periodization']
```

Test that *dwt()* and *idwt()* can be performed using every mode:

```
>>> x = [1,2,1,5,-1,8,4,6]
>>> for mode in pywt.Modes.modes:
...     cA, cD = pywt.dwt(x, 'db2', mode)
...     print("Mode: %s" % mode)
...     print("cA: " + format_array(cA))
...     print("cD: " + format_array(cD))
...     print("Reconstruction: " + format_array(
...         pywt.idwt(cA, cD, 'db2', mode)))
Mode: zero
cA: [-0.03468  1.73309  3.40612  6.32929  6.95095]
cD: [-0.12941 -2.156   -5.95035 -1.21545 -1.8625 ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: constant
cA: [ 1.2848   1.73309  3.40612  6.32929  7.51936]
cD: [-0.48296 -2.156   -5.95035 -1.21545  0.25882]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: symmetric
cA: [ 1.76777  1.73309  3.40612  6.32929  7.77817]
cD: [-0.61237 -2.156   -5.95035 -1.21545  1.22474]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: periodic
cA: [ 6.91627  1.73309  3.40612  6.32929  6.91627]
cD: [-1.99191 -2.156   -5.95035 -1.21545 -1.99191]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: smooth
cA: [-0.51764  1.73309  3.40612  6.32929  7.45001]
cD: [ 0.      -2.156   -5.95035 -1.21545  0.     ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: periodization
cA: [ 4.05317  3.05257  2.85381  8.42522]
cD: [ 0.18947  4.18258  4.33738  2.60428]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
```

Invalid mode name should rise a `ValueError`:

```
>>> pywt.dwt([1,2,3,4], 'db2', 'invalid')
Traceback (most recent call last):
...
ValueError: Unknown mode name 'invalid'.
```

You can also refer to modes via *Modes* class attributes:

```
>>> for mode_name in ['zero', 'constant', 'symmetric', 'periodic', 'smooth', 'periodization']:
...     mode = getattr(pywt.Modes, mode_name)
...     cA, cD = pywt.dwt([1,2,1,5,-1,8,4,6], 'db2', mode)
...     print("Mode: %d (%s)" % (mode, mode_name))
...     print("cA: " + format_array(cA))
...     print("cD: " + format_array(cD))
...     print("Reconstruction: " + format_array(
...         pywt.idwt(cA, cD, 'db2', mode)))
Mode: 0 (zero)
cA: [-0.03468  1.73309  3.40612  6.32929  6.95095]
cD: [-0.12941 -2.156   -5.95035 -1.21545 -1.8625 ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 2 (constant)
cA: [ 1.2848   1.73309  3.40612  6.32929  7.51936]
cD: [-0.48296 -2.156   -5.95035 -1.21545  0.25882]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 1 (symmetric)
cA: [ 1.76777  1.73309  3.40612  6.32929  7.77817]
cD: [-0.61237 -2.156   -5.95035 -1.21545  1.22474]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 4 (periodic)
cA: [ 6.91627  1.73309  3.40612  6.32929  6.91627]
cD: [-1.99191 -2.156   -5.95035 -1.21545 -1.99191]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 3 (smooth)
cA: [-0.51764  1.73309  3.40612  6.32929  7.45001]
cD: [ 0.      -2.156   -5.95035 -1.21545  0.     ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 5 (periodization)
cA: [ 4.05317  3.05257  2.85381  8.42522]
cD: [ 0.18947  4.18258  4.33738  2.60428]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
```

The default mode is *symmetric*:

```
>>> cA, cD = pywt.dwt(x, 'db2')
>>> print(cA)
[ 1.76776695  1.73309178  3.40612438  6.32928585  7.77817459]
>>> print(cD)
[-0.61237244 -2.15599552 -5.95034847 -1.21545369  1.22474487]
>>> print(pywt.idwt(cA, cD, 'db2'))
[ 1.  2.  1.  5. -1.  8.  4.  6.]
```

And using a keyword argument:

```
>>> cA, cD = pywt.dwt(x, 'db2', mode='symmetric')
>>> print(cA)
[ 1.76776695  1.73309178  3.40612438  6.32928585  7.77817459]
>>> print(cD)
[-0.61237244 -2.15599552 -5.95034847 -1.21545369  1.22474487]
>>> print(pywt.idwt(cA, cD, 'db2'))
[ 1.  2.  1.  5. -1.  8.  4.  6.]
```

DWT and IDWT

Discrete Wavelet Transform

Let's do a *Discrete Wavelet Transform* of a sample data *x* using the `db2` wavelet. It's simple..

```
>>> import pywt
>>> x = [3, 7, 1, 1, -2, 5, 4, 6]
>>> cA, cD = pywt.dwt(x, 'db2')
```

And the approximation and details coefficients are in `cA` and `cD` respectively:

```
>>> print(cA)
[ 5.65685425  7.39923721  0.22414387  3.33677403  7.77817459]
>>> print(cD)
[-2.44948974 -1.60368225 -4.44140056 -0.41361256  1.22474487]
```

Inverse Discrete Wavelet Transform

Now let's do an opposite operation - *Inverse Discrete Wavelet Transform*:

```
>>> print(pywt.idwt(cA, cD, 'db2'))
[ 3.  7.  1.  1. -2.  5.  4.  6.]
```

Voilà! That's it!

More Examples

Now let's experiment with the `dwt()` some more. For example let's pass a `Wavelet` object instead of the wavelet name and specify signal extension mode (the default is *symmetric*) for the border effect handling:

```
>>> w = pywt.Wavelet('sym3')
>>> cA, cD = pywt.dwt(x, wavelet=w, mode='constant')
>>> print(cA)
[ 4.38354585  3.80302657  7.31813271 -0.58565539  4.09727044  7.81994027]
>>> print(cD)
[-1.33068221 -2.78795192 -3.16825651 -0.67715519 -0.09722957 -0.07045258]
```

Note that the output coefficients arrays length depends not only on the input data length but also on the :class:Wavelet type (particularly on its `filters lenght` that are used in the transformation).

To find out what will be the output data size use the `dwt_coeff_len()` function:

```
>>> # int() is for normalizing Python integers and long integers for documentation tests
>>> int(pywt.dwt_coeff_len(data_len=len(x), filter_len=w.dec_len, mode='symmetric'))
6
>>> int(pywt.dwt_coeff_len(len(x), w, 'symmetric'))
6
>>> len(cA)
6
```

Looks fine. (And if you expected that the output length would be a half of the input data length, well, that's the trade-off that allows for the perfect reconstruction...).

The third argument of the `dwt_coeff_len()` is the already mentioned signal extension mode (please refer to the PyWavelets' documentation for the *modes* description). Currently there are six *extension modes* available:

```
>>> pywt.Modes.modes
['zero', 'constant', 'symmetric', 'periodic', 'smooth', 'periodization']
```

```
>>> [int(pywt.dwt_coeff_len(len(x), w.dec_len, mode)) for mode in pywt.Modes.modes]
[6, 6, 6, 6, 6, 4]
```

As you see in the above example, the *periodization* (periodization) mode is slightly different from the others. It's aim when doing the `DWT` transform is to output coefficients arrays that are half of the length of the input data.

Knowing that, you should never mix the periodization mode with other modes when doing `DWT` and `IDWT`. Otherwise, it will produce **invalid results**:

```
>>> x
[3, 7, 1, 1, -2, 5, 4, 6]
>>> cA, cD = pywt.dwt(x, wavelet=w, mode='periodization')
>>> print(pywt.idwt(cA, cD, 'sym3', 'symmetric')) # invalid mode
[ 1.  1. -2.  5.]
>>> print(pywt.idwt(cA, cD, 'sym3', 'periodization'))
[ 3.  7.  1.  1. -2.  5.  4.  6.]
```

Tips & tricks

Passing `None` instead of coefficients data to `idwt()` Now some tips & tricks. Passing `None` as one of the coefficient arrays parameters is similar to passing a *zero-filled* array. The results are simply the same:

```
>>> print(pywt.idwt([1,2,0,1], None, 'db2', 'symmetric'))
[ 1.19006969  1.54362308  0.44828774 -0.25881905  0.48296291  0.8365163 ]
```

```
>>> print(pywt.idwt([1, 2, 0, 1], [0, 0, 0, 0], 'db2', 'symmetric'))
[ 1.19006969  1.54362308  0.44828774 -0.25881905  0.48296291  0.8365163 ]
```

```
>>> print(pywt.idwt(None, [1, 2, 0, 1], 'db2', 'symmetric'))
[ 0.57769726 -0.93125065  1.67303261 -0.96592583 -0.12940952 -0.22414387]
```

```
>>> print(pywt.idwt([0, 0, 0, 0], [1, 2, 0, 1], 'db2', 'symmetric'))
[ 0.57769726 -0.93125065  1.67303261 -0.96592583 -0.12940952 -0.22414387]
```

Remember that only one argument at a time can be `None`:

```
>>> print(pywt.idwt(None, None, 'db2', 'symmetric'))
Traceback (most recent call last):
...
ValueError: At least one coefficient parameter must be specified.
```

Coefficients data size in `idwt` When doing the `IDWT` transform, usually the coefficient arrays must have the same size.

```
>>> print(pywt.idwt([1, 2, 3, 4, 5], [1, 2, 3, 4], 'db2', 'symmetric'))
Traceback (most recent call last):
...
ValueError: Coefficients arrays must have the same size.
```

Not every coefficient array can be used in `IDWT`. In the following example the `idwt()` will fail because the input arrays are invalid - they couldn't be created as a result of `DWT`, because the minimal output length for dwt using `db4` wavelet and the *symmetric* mode is `4`, not `3`:

```
>>> pywt.idwt([1,2,4], [4,1,3], 'db4', 'symmetric')
Traceback (most recent call last):
...
ValueError: Invalid coefficient arrays length for specified wavelet. Wavelet and mode must be the sa
```

```
>>> int(pywt.dwt_coeff_len(1, pywt.Wavelet('db4').dec_len, 'symmetric'))
4
```

Multilevel DWT, IDWT and SWT

Multilevel DWT decomposition

```
>>> import pywt
>>> x = [3, 7, 1, 1, -2, 5, 4, 6]
>>> db1 = pywt.Wavelet('db1')
>>> cA3, cD3, cD2, cD1 = pywt.wavedec(x, db1)
>>> print(cA3)
[ 8.83883476]
>>> print(cD3)
[-0.35355339]
>>> print(cD2)
[ 4.  -3.5]
>>> print(cD1)
[-2.82842712  0.          -4.94974747 -1.41421356]
```

```
>>> pywt.dwt_max_level(len(x), db1)
3
```

```
>>> cA2, cD2, cD1 = pywt.wavedec(x, db1, mode='constant', level=2)
```

Multilevel IDWT reconstruction

```
>>> coeffs = pywt.wavedec(x, db1)
>>> print(pywt.waverec(coeffs, db1))
[ 3.  7.  1.  1. -2.  5.  4.  6.]
```

Multilevel SWT decomposition

```
>>> x = [3, 7, 1, 3, -2, 6, 4, 6]
>>> (cA2, cD2), (cA1, cD1) = pywt.swt(x, db1, level=2)
>>> print(cA1)
[ 7.07106781  5.65685425  2.82842712  0.70710678  2.82842712  7.07106781
  7.07106781  6.36396103]
>>> print(cD1)
[-2.82842712  4.24264069 -1.41421356  3.53553391 -5.65685425  1.41421356
 -1.41421356  2.12132034]
>>> print(cA2)
[  7.    4.5   4.    5.5   7.    9.5  10.    8.5]
>>> print(cD2)
[ 3.   3.5  0.  -4.5 -3.   0.5  0.   0.5]
```

```
>>> [(cA2, cD2)] = pywt.swt(cA1, db1, level=1, start_level=1)
>>> print(cA2)
[  7.    4.5   4.    5.5   7.    9.5  10.    8.5]
>>> print(cD2)
[ 3.   3.5  0.  -4.5 -3.   0.5  0.   0.5]
```

```
>>> coeffs = pywt.swt(x, db1)
>>> len(coeffs)
3
>>> pywt.swt_max_level(len(x))
3
```

```
>>> from __future__ import print_function
```

Wavelet Packets

Import pywt

```
>>> import pywt
```

```
>>> def format_array(a):
...     """Consistent array representation across different systems"""
...     import numpy
...     a = numpy.where(numpy.abs(a) < 1e-5, 0, a)
...     return numpy.array2string(a, precision=5, separator=' ', suppress_small=True)
```

Create Wavelet Packet structure

Ok, let's create a sample *WaveletPacket*:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

The input *data* and decomposition coefficients are stored in the `WaveletPacket.data` attribute:

```
>>> print(wp.data)
[1, 2, 3, 4, 5, 6, 7, 8]
```

Nodes are identified by `paths`. For the root node the path is `''` and the decomposition level is `0`.

```
>>> print(repr(wp.path))
''
>>> print(wp.level)
0
```

The *maxlevel*, if not given as param in the constructor, is automatically computed:

```
>>> print(wp['ad'].maxlevel)
3
```

Traversing WP tree:

Accessing subnodes:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

First check what is the maximum level of decomposition:

```
>>> print(wp.maxlevel)
3
```

and try accessing subnodes of the WP tree:

- 1st level:

  ```
  >>> print(wp['a'].data)
  [  2.12132034   4.94974747   7.77817459  10.60660172]
  >>> print(wp['a'].path)
  a
  ```

- 2nd level:

  ```
  >>> print(wp['aa'].data)
  [  5.  13.]
  >>> print(wp['aa'].path)
  aa
  ```

- 3rd level:

  ```
  >>> print(wp['aaa'].data)
  [ 12.72792206]
  >>> print(wp['aaa'].path)
  aaa
  ```

  Ups, we have reached the maximum level of decomposition and got an `IndexError`:

  ```
  >>> print(wp['aaaa'].data)
  Traceback (most recent call last):
  ...
  IndexError: Path length is out of range.
  ```

Now try some invalid path:

```
>>> print(wp['ac'])
Traceback (most recent call last):
...
ValueError: Subnode name must be in ['a', 'd'], not 'c'.
```

which just yielded a `ValueError`.

Accessing Node's attributes: *`WaveletPacket`* object is a tree data structure, which evaluates to a set of *`Node`* objects. *`WaveletPacket`* is just a special subclass of the *`Node`* class (which in turn inherits from the *`BaseNode`*).

Tree nodes can be accessed using the *obj[x]* (`Node.__getitem__()`) operator. Each tree node has a set of attributes: `data`, `path`, *`node_name`*, `parent`, `level`, `maxlevel` and `mode`.

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

```
>>> print(wp['ad'].data)
[-2. -2.]
```

```
>>> print(wp['ad'].path)
ad
```

```
>>> print(wp['ad'].node_name)
d
```

```
>>> print(wp['ad'].parent.path)
a
```

```
>>> print(wp['ad'].level)
2
```

```
>>> print(wp['ad'].maxlevel)
3
```

```
>>> print(wp['ad'].mode)
symmetric
```

Collecting nodes

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

We can get all nodes on the particular level either in `natural` order:

```
>>> print([node.path for node in wp.get_level(3, 'natural')])
['aaa', 'aad', 'ada', 'add', 'daa', 'dad', 'dda', 'ddd']
```

or sorted based on the band frequency (`freq`):

```
>>> print([node.path for node in wp.get_level(3, 'freq')])
['aaa', 'aad', 'add', 'ada', 'dda', 'ddd', 'dad', 'daa']
```

Note that *`WaveletPacket.get_level()`* also performs automatic decomposition until it reaches the specified *level*.

Reconstructing data from Wavelet Packets:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

Now create a new *`Wavelet Packet`* and set its nodes with some data.

```
>>> new_wp = pywt.WaveletPacket(data=None, wavelet='db1', mode='symmetric')
```

```
>>> new_wp['aa'] = wp['aa'].data
>>> new_wp['ad'] = [-2., -2.]
```

For convenience, `Node.data` gets automatically extracted from the *`Node`* object:

```
>>> new_wp['d'] = wp['d']
```

And reconstruct the data from the `aa`, `ad` and `d` packets.

```
>>> print(new_wp.reconstruct(update=False))
[ 1.  2.  3.  4.  5.  6.  7.  8.]
```

If the *update* param in the reconstruct method is set to `False`, the node's `data` will not be updated.

```
>>> print(new_wp.data)
None
```

Otherwise, the `data` attribute will be set to the reconstructed value.

```
>>> print(new_wp.reconstruct(update=True))
[ 1.  2.  3.  4.  5.  6.  7.  8.]
>>> print(new_wp.data)
[ 1.  2.  3.  4.  5.  6.  7.  8.]
```

```
>>> print([n.path for n in new_wp.get_leaf_nodes(False)])
['aa', 'ad', 'd']
```

```
>>> print([n.path for n in new_wp.get_leaf_nodes(True)])
['aaa', 'aad', 'ada', 'add', 'daa', 'dad', 'dda', 'ddd']
```

Removing nodes from Wavelet Packet tree:

Let's create a sample data:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

First, start with a tree decomposition at level 2. Leaf nodes in the tree are:

```
>>> dummy = wp.get_level(2)
>>> for n in wp.get_leaf_nodes(False):
...     print(n.path, format_array(n.data))
aa [  5.  13.]
ad [-2. -2.]
da [-1. -1.]
dd [ 0.  0.]
```

```
>>> node = wp['ad']
>>> print(node)
ad: [-2. -2.]
```

To remove a node from the WP tree, use Python's *del obj[x]* (`Node.__delitem__`):

```
>>> del wp['ad']
```

The leaf nodes that left in the tree are:

```
>>> for n in wp.get_leaf_nodes():
...     print(n.path, format_array(n.data))
aa [  5.  13.]
da [-1. -1.]
dd [ 0.  0.]
```

And the reconstruction is:

```
>>> print(wp.reconstruct())
[ 2.  3.  2.  3.  6.  7.  6.  7.]
```

Now restore the deleted node value.

```
>>> wp['ad'].data = node.data
```

Printing leaf nodes and tree reconstruction confirms the original state of the tree:

```
>>> for n in wp.get_leaf_nodes(False):
...     print(n.path, format_array(n.data))
aa [  5.   13.]
ad [-2. -2.]
da [-1. -1.]
dd [ 0.   0.]
```

```
>>> print(wp.reconstruct())
[ 1.   2.   3.   4.   5.   6.   7.   8.]
```

Lazy evaluation:

Note: This section is for demonstration of pywt internals purposes only. Do not rely on the attribute access to nodes as presented in this example.

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> wp = pywt.WaveletPacket(data=x, wavelet='db1', mode='symmetric')
```

1. At first the wp's attribute *a* is None

    ```
    >>> print(wp.a)
    None
    ```

    Remember that you should not rely on the attribute access.

2. At first attempt to access the node it is computed via decomposition of its parent node (the wp object itself).

    ```
    >>> print(wp['a'])
    a: [  2.12132034   4.94974747   7.77817459  10.60660172]
    ```

3. Now the *wp.a* is set to the newly created node:

    ```
    >>> print(wp.a)
    a: [  2.12132034   4.94974747   7.77817459  10.60660172]
    ```

    And so is *wp.d*:

    ```
    >>> print(wp.d)
    d: [-0.70710678 -0.70710678 -0.70710678 -0.70710678]
    ```

2D Wavelet Packets

Import pywt

```
>>> from __future__ import print_function
>>> import pywt
>>> import numpy
```

Create 2D Wavelet Packet structure

Start with preparing test data:

```
>>> x = numpy.array([[1, 2, 3, 4, 5, 6, 7, 8]] * 8, 'd')
>>> print(x)
[[ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]]
```

Now create a `2D Wavelet Packet` object:

```
>>> wp = pywt.WaveletPacket2D(data=x, wavelet='db1', mode='symmetric')
```

The input *data* and decomposition coefficients are stored in the `WaveletPacket2D.data` attribute:

```
>>> print(wp.data)
[[ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]]
```

`Nodes` are identified by paths. For the root node the path is `''` and the decomposition level is `0`.

```
>>> print(repr(wp.path))
''
>>> print(wp.level)
0
```

The `WaveletPacket2D.maxlevel`, if not given in the constructor, is automatically computed based on the data size:

```
>>> print(wp.maxlevel)
3
```

Traversing WP tree:

Wavelet Packet `nodes` are arranged in a tree. Each node in a WP tree is uniquely identified and addressed by a `path` string.

In the 1D `WaveletPacket` case nodes were accessed using `'a'` (approximation) and `'d'` (details) path names (each node has two 1D children).

Because now we deal with a bit more complex structure (each node has four children), we have four basic path names based on the dwt 2D output convention to address the WP2D structure:

- `a` - LL, low-low coefficients
- `h` - LH, low-high coefficients
- `v` - HL, high-low coefficients
- `d` - HH, high-high coefficients

In other words, subnode naming corresponds to the `dwt2()` function output naming convention (as wavelet packet transform is based on the dwt2 transform):

```
                            -------------------
                            |        |        |
                            | cA(LL) | cH(LH) |
                            |        |        |
(cA, (cH, cV, cD))  <--->   -------------------
                            |        |        |
                            | cV(HL) | cD(HH) |
                            |        |        |
                            -------------------

   (fig.1: DWT 2D output and interpretation)
```

Knowing what the nodes names are, we can now access them using the indexing operator *obj[x]* (`WaveletPacket2D.__getitem__()`):

```
>>> print(wp['a'].data)
[[  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]]
>>> print(wp['h'].data)
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
>>> print(wp['v'].data)
[[-1. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1. -1.]]
>>> print(wp['d'].data)
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

Similarly, a subnode of a subnode can be accessed by:

```
>>> print(wp['aa'].data)
[[ 10.  26.]
 [ 10.  26.]]
```

Indexing base `WaveletPacket2D` (as well as 1D `WaveletPacket`) using compound path is just the same as indexing WP subnode:

```
>>> node = wp['a']
>>> print(node['a'].data)
[[ 10.  26.]
 [ 10.  26.]]
>>> print(wp['a']['a'].data is wp['aa'].data)
True
```

Following down the decomposition path:

```
>>> print(wp['aaa'].data)
[[ 36.]]
>>> print(wp['aaaa'].data)
```

```
Traceback (most recent call last):
...
IndexError: Path length is out of range.
```

Ups, we have reached the maximum level of decomposition for the `'aaaa'` path, which btw. was:

```
>>> print(wp.maxlevel)
3
```

Now try some invalid path:

```
>>> print(wp['f'])
Traceback (most recent call last):
...
ValueError: Subnode name must be in ['a', 'h', 'v', 'd'], not 'f'.
```

Accessing Node2D's attributes: *WaveletPacket2D* is a tree data structure, which evaluates to a set of *Node2D* objects. *WaveletPacket2D* is just a special subclass of the *Node2D* class (which in turn inherits from a *BaseNode*, just like with *Node* and *WaveletPacket* for the 1D case.).

```
>>> print(wp['av'].data)
[[-4. -4.]
 [-4. -4.]]
```

```
>>> print(wp['av'].path)
av
```

```
>>> print(wp['av'].node_name)
v
```

```
>>> print(wp['av'].parent.path)
a
```

```
>>> print(wp['av'].parent.data)
[[  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]]
```

```
>>> print(wp['av'].level)
2
```

```
>>> print(wp['av'].maxlevel)
3
```

```
>>> print(wp['av'].mode)
symmetric
```

Collecting nodes We can get all nodes on the particular level using the *WaveletPacket2D.get_level()* method:

- 0 level - the root *wp* node:

```
    >>> len(wp.get_level(0))
    1
    >>> print([node.path for node in wp.get_level(0)])
    ['']
```

- 1st level of decomposition:

```
>>> len(wp.get_level(1))
4
>>> print([node.path for node in wp.get_level(1)])
['a', 'h', 'v', 'd']
```

- 2nd level of decomposition:

```
>>> len(wp.get_level(2))
16
>>> paths = [node.path for node in wp.get_level(2)]
>>> for i, path in enumerate(paths):
...     if (i+1) % 4 == 0:
...         print(path)
...     else:
...         print(path, end=' ')
aa ah av ad
ha hh hv hd
va vh vv vd
da dh dv dd
```

- 3rd level of decomposition:

```
>>> print(len(wp.get_level(3)))
64
>>> paths = [node.path for node in wp.get_level(3)]
>>> for i, path in enumerate(paths):
...     if (i+1) % 8 == 0:
...         print(path)
...     else:
...         print(path, end=' ')
aaa aah aav aad aha ahh ahv ahd
ava avh avv avd ada adh adv add
haa hah hav had hha hhh hhv hhd
hva hvh hvv hvd hda hdh hdv hdd
vaa vah vav vad vha vhh vhv vhd
vva vvh vvv vvd vda vdh vdv vdd
daa dah dav dad dha dhh dhv dhd
dva dvh dvv dvd dda ddh ddv ddd
```

Note that `WaveletPacket2D.get_level()` performs automatic decomposition until it reaches the given level.

Reconstructing data from Wavelet Packets:

Let's create a new empty 2D Wavelet Packet structure and set its nodes values with known data from the previous examples:

```
>>> new_wp = pywt.WaveletPacket2D(data=None, wavelet='db1', mode='symmetric')
```

```
>>> new_wp['vh'] = wp['vh'].data # [[0.0, 0.0], [0.0, 0.0]]
>>> new_wp['vv'] = wp['vh'].data # [[0.0, 0.0], [0.0, 0.0]]
>>> new_wp['vd'] = [[0.0, 0.0], [0.0, 0.0]]
```

```
>>> new_wp['a'] = [[3.0, 7.0, 11.0, 15.0], [3.0, 7.0, 11.0, 15.0],
...                [3.0, 7.0, 11.0, 15.0], [3.0, 7.0, 11.0, 15.0]]
>>> new_wp['d'] = [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0],
...                [0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]]
```

For convenience, `Node2D.data` gets automatically extracted from the base *Node2D* object:

```
>>> new_wp['h'] = wp['h'] # all zeros
```

Note: just remember to not assign to the node.data parameter directly (todo).

And reconstruct the data from the `a`, `d`, `vh`, `vv`, `vd` and `h` packets (Note that `va` node was not set and the WP tree is "not complete" - the `va` branch will be treated as *zero-array*):

```
>>> print(new_wp.reconstruct(update=False))
[[ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]]
```

Now set the `va` node with the known values and do the reconstruction again:

```
>>> new_wp['va'] = wp['va'].data # [[-2.0, -2.0], [-2.0, -2.0]]
>>> print(new_wp.reconstruct(update=False))
[[ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.]]
```

which is just the same as the base sample data *x*.

Of course we can go the other way and remove nodes from the tree. If we delete the `va` node, again, we get the "not complete" tree from one of the previous examples:

```
>>> del new_wp['va']
>>> print(new_wp.reconstruct(update=False))
[[ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]
 [ 1.5  1.5  3.5  3.5  5.5  5.5  7.5  7.5]]
```

Just restore the node before next examples.

```
>>> new_wp['va'] = wp['va'].data
```

If the *update* param in the `WaveletPacket2D.reconstruct()` method is set to `False`, the node's `Node2D.data` attribute will not be updated.

```
>>> print(new_wp.data)
None
```

Otherwise, the `WaveletPacket2D.data` attribute will be set to the reconstructed value.

```
>>> print(new_wp.reconstruct(update=True))
[[ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]]
>>> print(new_wp.data)
[[ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]
 [ 1.   2.   3.   4.   5.   6.   7.   8.]]
```

Since we have an interesting WP structure built, it is a good occasion to present the `WaveletPacket2D.get_leaf_nodes()` method, which collects non-zero leaf nodes from the WP tree:

```
>>> print([n.path for n in new_wp.get_leaf_nodes()])
['a', 'h', 'va', 'vh', 'vv', 'vd', 'd']
```

Passing the *decompose=True* parameter to the method will force the WP object to do a full decomposition up to the *maximum level* of decomposition:

```
>>> paths = [n.path for n in new_wp.get_leaf_nodes(decompose=True)]
>>> len(paths)
64
>>> for i, path in enumerate(paths):
...     if (i+1) % 8 == 0:
...         print(path)
...     else:
...         try:
...             print(path, end=' ')
...         except:
...             print(path, end=' ')
aaa aah aav aad aha ahh ahv ahd
ava avh avv avd ada adh adv add
haa hah hav had hha hhh hhv hhd
hva hvh hvv hvd hda hdh hdv hdd
vaa vah vav vad vha vhh vhv vhd
vva vvh vvv vvd vda vdh vdv vdd
daa dah dav dad dha dhh dhv dhd
dva dvh dvv dvd dda ddh ddv ddd
```

Lazy evaluation:

Note: This section is for demonstration of pywt internals purposes only. Do not rely on the attribute access to nodes as presented in this example.

```
>>> x = numpy.array([[1, 2, 3, 4, 5, 6, 7, 8]] * 8)
>>> wp = pywt.WaveletPacket2D(data=x, wavelet='db1', mode='symmetric')
```

1. At first the wp's attribute *a* is None

```
>>> print(wp.a)
None
```

Remember that you should not rely on the attribute access.

2. During the first attempt to access the node it is computed via decomposition of its parent node (the wp object itself).

```
>>> print(wp['a'])
a: [[  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]]
```

3. Now the *a* is set to the newly created node:

```
>>> print(wp.a)
a: [[  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]
 [  3.   7.  11.  15.]]
```

And so is *wp.d*:

```
>>> print(wp.d)
d: [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

Gotchas

PyWavelets utilizes NumPy under the hood. That's why handling the data containing None values can be surprising. None values are converted to 'not a number' (numpy.NaN) values:

```
>>> import numpy, pywt
>>> x = [None, None]
>>> mode = 'symmetric'
>>> wavelet = 'db1'
>>> cA, cD = pywt.dwt(x, wavelet, mode)
>>> numpy.all(numpy.isnan(cA))
True
>>> numpy.all(numpy.isnan(cD))
True
>>> rec = pywt.idwt(cA, cD, wavelet, mode)
>>> numpy.all(numpy.isnan(rec))
True
```

10.5.3 Development notes

This section contains information on building and installing PyWavelets from source code as well as instructions for preparing the build environment on Windows and Linux.

Preparing Windows build environment

To start developing PyWavelets code on Windows you will have to install a C compiler and prepare the build environment.

Installing Windows SDK C/C++ compiler

Depending on your Python version, a different version of the Microsoft Visual C++ compiler will be required to build extensions. The same compiler that was used to build Python itself should be used.

For official binary builds of Python 2.6 to 3.2, this will be VS 2008. Python 3.3 and 3.4 were compiled with VS 2010, and for Python 3.5 it will be MSVC 2015.

The MSVC version should be printed when starting a Python REPL, and can be checked against the note below:

Note: For reference:

- the *MSC v.1500* in the Python version string is Microsoft Visual C++ 2008 (Microsoft Visual Studio 9.0 with msvcr90.dll runtime)
- *MSC v.1600* is MSVC 2010 (10.0 with msvcr100.dll runtime)
- *MSC v.1700* is MSVC 2012 (11.0)
- *MSC v.1800* is MSVC 2013 (12.0)
- *MSC v.1900* is MSVC 2015 (14.0)

```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win32
Python 3.2 (r32:88445, Feb 20 2011, 21:30:00) [MSC v.1500 64 bit (AMD64)] on win32
```

To get started first download, extract and install *Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1* from http://www.microsoft.com/downloads/en/details.aspx?familyid=71DEB800-C591-4F97-A900-BEA146E4FAE1&displaylang=en.

There are several ISO images on the site, so just grab the one that is suitable for your platform:

- `GRMSDK_EN_DVD.iso` for 32-bit x86 platform
- `GRMSDKX_EN_DVD.iso` for 64-bit AMD64 platform (AMD64 is the codename for 64-bit CPU architecture, not the processor manufacturer)

After installing the SDK and before compiling the extension you have to configure some environment variables.

For 32-bit build execute the `util/setenv_build32.bat` script in the cmd window:

```
    rem Configure the environment for 32-bit builds.
    rem Use "vcvars32.bat" for a 32-bit build.
    "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat"
    rem Convince setup.py to use the SDK tools.
    set MSSdk=1
    setenv /x86 /release
    set DISTUTILS_USE_SDK=1
```

For 64-bit use `util/setenv_build64.bat`:

```
    rem Configure the environment for 64-bit builds.
    rem Use "vcvars32.bat" for a 32-bit build.
    "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\vcvars64.bat"
    rem Convince setup.py to use the SDK tools.
    set MSSdk=1
    setenv /x64 /release
    set DISTUTILS_USE_SDK=1
```

See also http://wiki.cython.org/64BitCythonExtensionsOnWindows.

MinGW C/C++ compiler

MinGW distribution can be downloaded from http://sourceforge.net/projects/mingwbuilds/.

In order to change the settings and use MinGW as the default compiler, edit or create a Distutils configuration file `c:\Python2*\Lib\distutils\distutils.cfg` and place the following entry in it:

```
[build]
compiler = mingw32
```

You can also take a look at Cython's "Installing MinGW on Windows" page at http://wiki.cython.org/InstallingOnWindows for more info.

Note: Python 2.7/3.2 distutils package is incompatible with the current version (4.7+) of MinGW (MinGW dropped the `-mno-cygwin` flag, which is still passed by distutils).

To use MinGW to compile Python extensions you have to patch the `distutils/cygwinccompiler.py` library module and remove every occurrence of `-mno-cygwin`.

See http://bugs.python.org/issue12641 bug report for more information on the issue.

Next steps

After completing these steps continue with *Installing build dependencies*.

Preparing Linux build environment

There is a good chance that you already have a working build environment. Just skip steps that you don't need to execute.

Installing basic build tools

Note that the example below uses `aptitude` package manager, which is specific to Debian and Ubuntu Linux distributions. Use your favourite package manager to install these packages on your OS.

```
aptitude install build-essential gcc python-dev git-core
```

Next steps

After completing these steps continue with *Installing build dependencies*.

Installing build dependencies

Setting up Python virtual environment

A good practice is to create a separate Python virtual environment for each project. If you don't have virtualenv yet, install and activate it using:

```
curl -O https://raw.github.com/pypa/virtualenv/master/virtualenv.py
python virtualenv.py <name_of_the_venv>
. <name_of_the_venv>/bin/activate
```

Installing Cython

Use `pip` (http://pypi.python.org/pypi/pip) to install Cython:

```
pip install Cython>=0.16
```

Installing numpy

Use `pip` to install numpy:

```
pip install numpy
```

It takes some time to compile numpy, so it might be more convenient to install it from a binary release.

Note: Installing numpy in a virtual environment on Windows is not straightforward.

It is recommended to download a suitable binary `.exe` release from http://www.scipy.org/Download/ and install it using `easy_install` (i.e. `easy_install numpy-1.6.2-win32-superpack-python2.7.exe`).

Note: You can find binaries for 64-bit Windows on http://www.lfd.uci.edu/~gohlke/pythonlibs/.

Installing Sphinx

Sphinx is a documentation tool that converts reStructuredText files into nicely looking html documentation. Install it with:

```
pip install Sphinx
```

numpydoc is used to format the API docmentation appropriately. Install it via:

```
pip install numpydoc
```

Building and installing PyWavelets

Installing from source code

Go to https://github.com/PyWavelets/pywt GitHub project page, fork and clone the repository or use the upstream repository to get the source code:

```
git clone https://github.com/PyWavelets/pywt.git PyWavelets
```

Activate your Python virtual environment, go to the cloned source directory and type the following commands to build and install the package:

```
python setup.py build
python setup.py install
```

To verify the installation run the following command:

```
python setup.py test
```

To build docs:

```
cd doc
make html
```

Installing a development version

You can also install directly from the source repository:

```
pip install -e git+https://github.com/PyWavelets/pywt.git#egg=PyWavelets
```

or:

```
pip install PyWavelets==dev
```

Installing a regular release from PyPi

A regular release can be installed with pip or easy_install:

```
pip install PyWavelets
```

Testing

Continous integration with Travis-CI

The project is using Travis-CI service for continous integration and testing.

Current build status is: If you are submitting a patch or pull request please make sure it does not break the build.

Running tests locally

Tests are implemented with nose, so use one of:

> $ nosetests pywt

```
    >>> pywt.test()
```

Running tests with Tox

There's also a config file for running tests with Tox (`pip install tox`). To for example run tests for Python 2.7 and Python 3.4 use:

```
tox -e py27,py34
```

For more information see the Tox documentation.

Something not working?

If these instructions are not clear or you need help setting up your development environment, go ahead and ask on the PyWavelets discussion group at http://groups.google.com/group/pywavelets or open a ticket on GitHub.

10.5.4 Resources

Code

The GitHub repository is now the main code repository.

If you are using the Mercurial repository at Bitbucket, please switch to Git/GitHub and follow for development updates.

Questions and bug reports

Use GitHub Issues or PyWavelets discussions group to post questions and open tickets.

Wavelet Properties Browser

Browse properties and graphs of wavelets included in PyWavelets on wavelets.pybytes.com.

Articles

- Denoising: wavelet thresholding
- Wavelet Regression in Python

10.5.5 Release Notes

PyWavelets 0.3.0 Release Notes

Contents

PyWavelets 0.3.0 is the first release of the package in 3 years. It is the result of a significant effort of a growing development team to modernize the package, to provide Python 3.x support and to make a start with providing new features as well as improved performance. A 0.4.0 release will follow shortly, and will contain more significant new features as well as changes/deprecations to streamline the API.

This release requires Python 2.6, 2.7 or 3.3-3.5 and NumPy 1.6.2 or greater.

Highlights of this release include:

- Support for Python 3.x (>=3.3)
- Added a test suite (based on nose, coverage up to 61% so far)
- Maintenance work: C style complying to the Numpy style guide, improved templating system, more complete docstrings, pep8/pyflakes compliance, and more.

New features

Test suite The test suite can be run with `nosetests pywt` or with:

```
>>> import pywt
>>> pywt.test()
```

n-D Inverse Discrete Wavelet Transform The function `pywt.idwtn`, which provides n-dimensional inverse DWT, has been added. It complements `idwt`, `idwt2` and `dwtn`.

Thresholding The function *pywt.threshold* has been added. It unifies the four thresholding functions that are still provided in the `pywt.thresholding` namespace.

Backwards incompatible changes

None in this release.

Other changes

Development has moved to a new repo. Everyone with an interest in wavelets is welcome to contribute!

Building wheels, building with `python setup.py develop` and many other standard ways to build and install PyWavelets are supported now.

Authors

- Ankit Agrawal +
- François Boulogne +
- Ralf Gommers +
- David Menéndez Hurtado +
- Gregory R. Lee +
- David McInnis +
- Helder Oliveira +
- Filip Wasilewski
- Kai Wohlfahrt +

A total of 9 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed for v0.3.0

- #3: Remove numerix compat layer
- #4: Add single code base Python 3 support
- #5: PEP8 issues
- #6: Migrate tests to nose
- #7: Expand test coverage without Matlab to a reasonable level
- #8: Replace custom C templates by Numpy's templating system
- #9: Replace Cython templates by fused types
- #10: Replace use of __array_interface__ with Cython's memoryviews
- #11: Format existing docstrings in numpydoc format.
- #12: Complete docstrings, they're quite sparse right now
- #13: Reorganize source tree
- #24: doc/source/regression should be moved
- #27: Broken test: test_swt_decomposition
- #28: Install issue, no module tools.six
- #29: wp.update fails after removal of nodes
- #32: wp.update fails on 2D
- #34: Wavelet string attributes shouldn't be bytes in Python 3
- #35: Re-enable float32 support
- #36: wavelet instance vs string
- #40: Test with Numpy 1.8rc1
- #45: demos should be updated and integrated in docs
- #60: Moving pywt forward faster

- #61: issues to address in moving towards 0.3.0
- #71: BUG: _pywt.downcoef always returns level=1 result

Pull requests for v0.3.0

- #1: travis: check all branches + fix URL
- #17: [DOC] doctrings for multilevel functions
- #18: DOC: format -> functions.py
- #20: MAINT: remove unnecessary zero() copy()
- #21: Doc wavelet_packets
- #22: Minor doc fixes
- #25: TEST: remove useless functions and use numpy instead
- #26: Merge most recent work
- #30: Adding test for wp.rst
- #41: Change to Numpy templating system
- #43: MAINT: update six.py to not use lazy loading.
- #49: Taking on API Issues
- #50: Add idwtn
- #53: readme updated with info related to Py3 version
- #63: Remove six
- #65: Thresholding
- #70: MAINT: PEP8 fixes
- #72: BUG: fix _downcoef for level > 1
- #73: MAINT: documentation and metadata update for repo fork
- #74: STY: fix pep8/pyflakes issues
- #77: MAINT: raise ValueError if data given to dwt or idwt is not 1D...

PyWavelets 0.4.0 Release Notes

Contents

PyWavelets 0.4.0 is the culmination of 6 months of work. In addition to several new features, some changes and deprecations have been made to streamline the API.

This release requires Python 2.6, 2.7 or 3.3-3.5 and NumPy 1.6.2 or greater.

Highlights of this release include:

- 1D and 2D inverse stationary wavelet transforms
- Substantially faster 2D and nD discrete wavelet transforms
- Complex number support
- nD versions of the multilevel DWT and IDWT

New features

1D and 2D inverse stationary wavelet transforms 1D (`iswt`) and 2D (`iswt2`) inverse stationary wavelet transforms were added. These currently only support even length inputs.

Faster 2D and nD wavelet transforms The multidimensional DWT and IDWT code was refactored and is now an order of magnitude faster than in previous releases. The following functions benefit: `dwt2`, `idwt2`, `dwtn`, `idwtn`.

Complex floating point support 64 and 128-bit complex data types are now supported by all wavelet transforms.

nD implementation of the multilevel DWT and IDWT The existing 1D and 2D multilevel transforms were supplemented with an nD implementation.

Wavelet transforms can be applied along a specific axis/axes All wavelet transform functions now support explicit specification of the axis or axes upon which to perform the transform.

Example Datasets Two additional 2D grayscale images were added (*camera*, *ascent*). The previously existing 1D ECG data (*ecg*) and the 2D aerial image (*aero*) used in the demos can also now be imported via functions defined in *pywt.data* (e.g. `camera = pywt.data.camera()`)

Deprecated features

A number of functions have been renamed, the old names are deprecated and will be removed in a future release:

- `intwave`, renamed to `integrate_wavelet`
- `centrfrq`, renamed to `central_frequency`
- `scal2frq`, renamed to `scale2frequency`
- `orthfilt`, renamed to `orthogonal_filter_bank`

Integration of general signals (i.e. not wavelets) with `integrate_wavelet` is deprecated.

The `MODES` object and its attributes are deprecated. The new name is `Modes`, and the attribute names are expanded:

- `zpd`, renamed to `zero`
- `cpd`, renamed to `constant`
- `sp1`, renamed to `smooth`
- `sym`, renamed to `symmetric`
- `ppd`, renamed to `periodic`
- `per`, renamed to `periodization`

Backwards incompatible changes

`idwt` no longer takes a `correct_size` parameter. As a consequence, `idwt2` inputs must match exactly in length. For multilevel transforms, where arrays differing in size by one element may be produced, use the `waverec` functions from the `multilevel` module instead.

Bugs Fixed

float32 inputs were not always respected. All transforms now return float32 outputs when called using float32 inputs.

Incorrect detail coefficients were returned by *downcoef* when *level > 1*.

Other changes

Much of the API documentation is now autogenerated from the corresponding function docstrings. The numpydoc sphinx extension is now needed to build the documentation.

Authors

- Thomas Arildsen +
- François Boulogne
- Ralf Gommers
- Gregory R. Lee
- Michael Marino +
- Aaron O'Leary +

- Daniele Tricoli +
- Kai Wohlfahrt

A total of 8 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed for v0.4.0

- #46: Independent test comparison
- #95: Simplify Matlab tests
- #97: BUG: erroneous detail coefficients returned by downcoef with...
- #140: demo/dwt_signal_decomposition.py : TypeError: object of type...
- #141: Documentation needs update: ImportError: cannot import name 'multilevel'

Pull requests for v0.4.0

- #55: [RFC] Api changes
- #59: Refactor convolution.c.src
- #64: MAINT: make LH, HL variable names in idwt2 consistent with dwt2
- #67: ENH: add wavedecn and waverecn functions
- #68: ENH: Faster dwtn and idwtn
- #88: DOC minor edit about possible naming
- #93: Added implementation of iswt and iswt2
- #98: fix downcoef detail coefficients for level > 1
- #99: complex support in all dwt and idwt related functions
- #100: replace mlabwrap with python-matlab-bridge in Matlab tests
- #102: Replace some .src expansion with macros
- #104: Faster idwtn/dwtn
- #106: make sure transforms respect float32 dtype
- #109: DOC: fix broken link in sidebar for html docs.
- #112: Complex fix
- #113: TST: don't build .exe installers on Appveyor anymore, only wheels.
- #116: [RFC] ENH: Add axis argument to dwt
- #117: MAINT: remove deprecated for loop syntax from Cython code
- #121: Fix typo
- #123: MAINT: remove some unused imports
- #124: switch travis from python 3.5-dev to 3.5
- #130: Add axis argument to multidim
- #138: WIP: Documentation updates for v0.4.0
- #139: Autogenerate function API docs

- #142: fix broken docstring examples in _multilevel.py
- #143: handle None properly in waverec
- #144: Add importable images
- #145: DOC: Document MSVC versions

10.6 Indices and tables

- genindex
- search

Index