PyWavelets Documentation

Release 1.0.0.dev0+bf7be1f

The PyWavelets Developers

Mar 16, 2018

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 :)

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

- 1D Continuous Wavelet Transform

- Computing Approximations of wavelet and scaling functions

- Over 100 built-in wavelet filters and support for custom wavelets

- 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 2.7 or >=3.4
- Numpy >= 1.9.1

Download

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

The latest release, including source and binary packages for Intel Linux, macOS and Windows, is available for download from the Python Package Index. You can find source releases at the Releases Page.

CHAPTER 4

Install

There are binary wheels for Intel Linux, Windows and macOS / OSX on PyPi. If you are on one of these platforms, you should get a binary (precompiled) installation with:

```
pip install PyWavelets
```

Users of the Anaconda Python distribution may wish to obtain pre-built Windows, Intel Linux or macOS / OSX binaries from the default channel. This can be done via:

```
conda install pywavelets
```

Several Linux distributions have their own packages for PyWavelets, but these tend to be moderately out of date. Query your Linux package manager tool for `python-pywavelets`, `python-wavelets`, `python-pywt` or a similar package name.

If you want or need to install from source, you will need a working C compiler (any common one will work) and a recent version of Cython. Navigate to the PyWavelets source code directory (containing `setup.py`) and type:

```
pip install .
```

To run all the tests for PyWavelets, you will also need to install the Matplotlib package.

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

The latest release, including source and binary packages, is available for download from the Python Package Index or on the Releases Page.

See also:

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

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!

Contact

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

CHAPTER 8

License

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

Citing

If you use PyWavelets in a scientific publication, we would appreciate citations of the project:

Lee G, Wasilewski F, Gommers R, Wohlfahrt K, O'Leary A, Nahrstaedt H, and Contributors, "PyWavelets - Wavelet Transforms in Python", 2006-, https://github.com/PyWavelets/pywt [Online; accessed 2018-MM-DD].

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`)
- Gaussian wavelets (`gaus`)
- Mexican hat wavelet (`mexh`)
- Morlet wavelet (`morl`)
- Complex Gaussian wavelets (`cgau`)
- Shannon wavelets (`shan`)
- Frequency B-Spline wavelets (`fbsp`)
- Complex Morlet wavelets (`cmor`)

> **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', 'gaus', 'mexh', 'morl',
↪'cgau', 'shan', 'fbsp', 'cmor']
>>> pywt.families(short=False)
['Haar', 'Daubechies', 'Symlets', 'Coiflets', 'Biorthogonal', 'Reverse␣
↪biorthogonal', 'Discrete Meyer (FIR Approximation)', 'Gaussian', 'Mexican hat␣
↪wavelet', 'Morlet wavelet', 'Complex Gaussian wavelets', 'Shannon wavelets',
↪'Frequency B-Spline wavelets', 'Complex Morlet wavelets']
```

Built-in wavelets - `wavelist()`

pywt.**wavelist**(*family=None*, *kind='all'*)

> Returns list of available wavelet names for the given family name.
>
> > **Parameters family** : str, optional
> >
> > > 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. Valid names are:
> > >
> > > ```
> > > 'haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey', 'gaus',
> > > 'mexh', 'morl', 'cgau', 'shan', 'fbsp', 'cmor'
> > > ```
> >
> > **kind** : {'all', 'continuous', 'discrete'}, optional
> >
> > > Whether to return only wavelet names of discrete or continuous wavelets, or all
> > > wavelets. Default is `'all'`. Ignored if `family` is specified.
> >
> > **Returns wavelist** : list of str
> >
> > > List of available wavelet names.

Examples

```
>>> import pywt
>>> pywt.wavelist('coif')
['coif1', 'coif2', 'coif3', 'coif4', 'coif5', 'coif6', 'coif7', ...
>>> pywt.wavelist(kind='continuous')
['cgau1', 'cgau2', 'cgau3', 'cgau4', 'cgau5', 'cgau6', 'cgau7', ...
```

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:

```python
>>> 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
  DWT:            True
  CWT:            False
>>> 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* object.

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)
```

> [   ]

ContinuousWavelet object

class pywt.**ContinuousWavelet**(*name*)

Describes properties of a continuous 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.

> **Parameters name** – Wavelet name

Example:

```
>>> import pywt
>>> wavelet = pywt.ContinuousWavelet('gaus1')
```

name
> Continuous Wavelet name.

short_family_name
> Wavelet short family name

family_name
> Wavelet family name

orthogonal
> Set if wavelet is orthogonal

biorthogonal
> Set if wavelet is biorthogonal

complex_cwt
> Returns if wavelet is complex

lower_bound
> Set the lower bound of the effective support

upper_bound
> Set the upper bound of the effective support

center_frequency
> Set the center frequency for the shan, fbsp and cmor wavelets

bandwidth_frequency
> Set the bandwidth frequency for the shan, fbsp and cmor wavelets

fbsp_order
> Set the order for the fbsp wavelet

symmetry
> asymmetric, near symmetric, symmetric, anti-symmetric

Example:

```
>>> import pywt
>>> wavelet = pywt.ContinuousWavelet('gaus1')
>>> print(wavelet)
ContinuousWavelet gaus1
  Family name:    Gaussian
  Short name:     db
  Symmetry:       anti-symmetric
  DWT:            False
```

```
  CWT:           True
  Complex CWT:   False
```

Approximating wavelet functions - `ContinuousWavelet.wavefun()`

ContinuousWavelet.**wavefun**(*level*, *length = None*)

The *wavefun()* method can be used to calculate approximations of scaling function (psi) with grid (x). The vector length is set by `length`. The vector length can also be defined by `2**level` if `length` is not set.

For *complex_cwt* wavelets returns a complex approximations of wavelet function with corresponding x-grid coordinates:

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

Example:

```python
>>> import pywt
>>> wavelet = pywt.ContinuousWavelet('gaus1')
>>> psi, x = wavelet.wavefun(level=5)
```

Approximating wavelet functions - `ContinuousWavelet.wavefun()`

pywt.**DiscreteContinuousWavelet**(*name*[, *filter_bank = None*])

The *DiscreteContinuousWavelet()* **returns a** Wavelet or a ContinuousWavelet object depending on the given name.

Example:

```python
>>> import pywt
>>> wavelet = pywt.DiscreteContinuousWavelet('db1')
>>> print(wavelet)
Wavelet db1
  Family name:    Daubechies
  Short name:     db
  Filters length: 2
  Orthogonal:     True
  Biorthogonal:   True
  Symmetry:       asymmetric
  DWT:            True
  CWT:            False
>>> wavelet = pywt.DiscreteContinuousWavelet('gaus1')
>>> print(wavelet)
ContinuousWavelet gaus1
  Family name:    Gaussian
  Short name:     db
  Symmetry:       anti-symmetric
  DWT:            False
  CWT:            True
  Complex CWT:    False
```

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 ...
  ```

- `reflect` - **reflect-padding** - signal is extended by *reflecting* samples:

  ```
  ... x3 x2 | x1 x2 ... xn | xn-1 xn-2 ...
  ```

- `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.

Naming Conventions

The correspondence between PyWavelets edge modes and the extension modes available in Matlab's dwtmode and numpy's pad are tabulated here for reference.

PyWavelets	Matlab	numpy.pad
symmetric	sym, symh	symmetric
reflect	symw	reflect
smooth	spd, sp1	N/A
constant	sp0	edge
zero	zpd	constant, cval=0
periodic	ppd	wrap
periodization	per	N/A
N/A	asym, asymh	N/A
N/A	asymw	N/A

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*, *axis=-1*)
   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.
>
> **axis: int, optional**
>
> > Axis over which to compute the DWT. If not given, the last axis is used.
>
> **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, str or Wavelet
>>
>> The wavelet filter length. Alternatively, the name of a discrete wavelet or a Wavelet object can be specified.
>
> **Returns max_level** : int
>
>> Maximum level.

Notes

The rational for the choice of levels is the maximum level where at least one coefficient in the output is uncorrupted by edge effects caused by signal extension. Put another way, decomposition stops when the signal

becomes shorter than the FIR filter length for a given wavelet. This corresponds to:

$$\texttt{max\_level} = \left\lfloor \log_2 \left(\frac{\texttt{data\_len}}{\texttt{filter\_len} - 1} \right) \right\rfloor$$

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
>>> pywt.dwt_max_level(1000, 'sym5')
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 (`data_len`), wavelet decomposition filter length (`filter_len`) 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'*, *axis=-1*)

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')

axis: int, optional

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

Notes

It may sometimes be desired to run `waverec` with some sets of coefficients omitted. This can best be done by setting the corresponding arrays to zero arrays of matching shape and dtype. Explicitly removing list entries or setting them to `None` is not supported.

Specifically, to ignore detail coefficients at level 2, one could do:

```
coeffs[-2] == np.zeros_like(coeffs[-2])
```

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** : array_like
>
> > 2D array with input data
>
> **wavelet** : Wavelet object or name string, or 2-tuple of wavelets
>
> > Wavelet to use. This can also be a tuple containing a wavelet to apply along each axis in `axes`.
>
> **mode** : str or 2-tuple of strings, optional
>
> > Signal extension mode, see Modes (default: 'symmetric'). This can also be a tuple of modes specifying the mode to use on each axis in `axes`.
>
> **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.
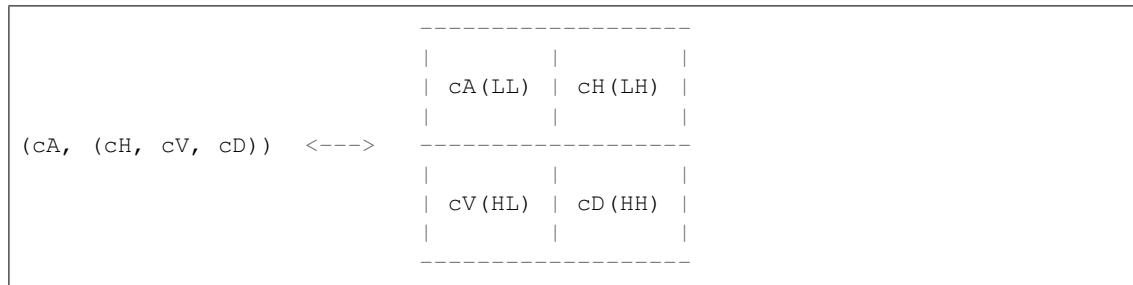>
> **Returns (cA, (cH, cV, cD))** : tuple
>
> > Approximation, horizontal detail, vertical detail and diagonal detail coefficients respectively. Horizontal refers to array axis 0 (or `axes[0]` for user-specified `axes`).

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*. If any of these components are set to None, it will be treated as zeros.

>> **wavelet** : Wavelet object or name string, or 2-tuple of wavelets

>>> Wavelet to use. This can also be a tuple containing a wavelet to apply along each axis in axes.

>> **mode** : str or 2-tuple of strings, optional

>>> Signal extension mode, see Modes (default: 'symmetric'). This can also be a tuple of modes specifying the mode to use on each axis in axes.

>> **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*, *axes=(-2, -1)*)
    Multilevel 2D Discrete Wavelet Transform.

>   **Parameters data** : ndarray
>
>>   2D input data
>
>>   **wavelet** : Wavelet object or name string, or 2-tuple of wavelets
>
>>   Wavelet to use. This can also be a tuple containing a wavelet to apply along each axis
>>   in `axes`.
>
>>   **mode** : str or 2-tuple of str, optional
>
>>   Signal extension mode, see Modes (default: 'symmetric'). This can also be a tuple
>>   containing a mode to apply along each axis in `axes`.
>
>>   **level** : int, optional
>
>>   Decomposition level (must be >= 0). If level is None (default) then it will be calculated
>>   using the `dwt_max_level` function.
>
>>   **axes** : 2-tuple of ints, optional
>
>>   Axes over which to compute the DWT. Repeated elements are not allowed.
>
>   **Returns [cAn, (cHn, cVn, cDn), … (cH1, cV1, cD1)]** : list
>
>>   Coefficients list. For user-specified axes, `cH*` corresponds to `axes[0]` while `cV*`
>>   corresponds to `axes[1]`. The first element returned is the approximation coefficients
>>   for the nth level of decomposition. Remaining elements are tuples of detail coefficients
>>   in descending order of decomposition level. (i.e. cH1 are the horizontal detail coeffi-
>>   cients at the first level)

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.]])
```
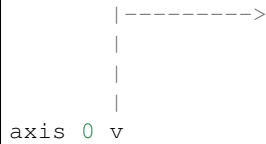
2D multilevel reconstruction using `waverec2`

pywt.**waverec2**(*coeffs*, *wavelet*, *mode='symmetric'*, *axes=(-2, -1)*)
    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, or 2-tuple of wavelets] Wavelet to use. This can also be a tuple con-
    taining a wavelet to apply along each axis in `axes`.

mode [str or 2-tuple of str, optional] Signal extension mode, see Modes (default: 'symmetric'). This can also be a tuple containing a mode to apply along each axis in `axes`.

axes [2-tuple of ints, optional] Axes over which to compute the IDWT. Repeated elements are not allowed.

> **Returns** 2D array of reconstructed data.

Notes

It may sometimes be desired to run `waverec2` with some sets of coefficients omitted. This can best be done by setting the corresponding arrays to zero arrays of matching shape and dtype. Explicitly removing list or tuple entries or setting them to `None` is not supported.

Specifically, to ignore all detail coefficients at level 2, one could do:

```
coeffs[-2] == tuple([np.zeros_like(v) for v in coeffs[-2]])
```

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 coordinate conventions

The labels for "horizontal" and "vertical" used by `dwt2` and `idwt2` follow the common mathematical convention that coordinate axis 0 is horizontal while axis 1 is vertical:

```
dwt2, idwt2 convention
----------------------

axis 1 ^
       |
       |
       |
       |--------->
             axis 0
```

Note that this is different from another common convention used in computer graphics and image processing (e.g. by matplotlib's `imshow` and functions in `scikit-image`). In those packages axis 0 is a vertical axis and axis 1 is horizontal as follows:

```
 imshow convention
-------------------
          axis 1
```

```
          |--------->
          |
          |
          |
axis 0  v
```

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** : array_like
  >
  >> n-dimensional array with input data.
  >
  >> **wavelet** : Wavelet object or name string, or tuple of wavelets
  >>
  >>> Wavelet to use. This can also be a tuple containing a wavelet to apply along each axis in `axes`.
  >
  >> **mode** : str or tuple of string, optional
  >>
  >>> Signal extension mode used in the decomposition, see Modes (default: 'symmetric'). This can also be a tuple of modes specifying the mode to use on each axis in `axes`.
  >
  >> **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
}
```

  >> For user-specified `axes`, the order of the characters in the dictionary keys map to the specified `axes`.

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 zeros.
>
> > **wavelet** : Wavelet object or name string, or tuple of wavelets
> >
> > Wavelet to use. This can also be a tuple containing a wavelet to apply along each axis in `axes`.
> >
> > **mode** : str or list of string, optional
> >
> > Signal extension mode used in the decomposition, see Modes (default: 'symmetric'). This can also be a tuple of modes specifying the mode to use on each axis in `axes`.
> >
> > **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*, *axes=None*)
> Multilevel nD Discrete Wavelet Transform.
>
> > **Parameters data** : ndarray
> >
> > > nD input data
> >
> > > **wavelet** : Wavelet object or name string, or tuple of wavelets
> > >
> > > Wavelet to use. This can also be a tuple containing a wavelet to apply along each axis in `axes`.
> > >
> > > **mode** : str or tuple of str, optional
> > >
> > > Signal extension mode, see Modes (default: 'symmetric'). This can also be a tuple containing a mode to apply along each axis in `axes`.
> > >
> > > **level** : int, optional
> > >
> > > Decomposition level (must be >= 0). If level is None (default) then it will be calculated using the `dwt_max_level` function.
> > >
> > > **axes** : sequence of ints, optional
> > >
> > > Axes over which to compute the DWT. Axes may not be repeated. The default is `None`, which means transform all axes (`axes = range(data.ndim)`).
> >
> > **Returns [cAn, {details_level_n}, ... {details_level_1}]** : list
> >
> > > Coefficients list. Coefficients are listed in descending order of decomposition level. `cAn` are the approximation coefficients at level `n`. Each `details_level_i` element is a dictionary containing detail coefficients at level *i* of the decomposition. As a concrete example, a 3D decomposition would have the following set of keys in each `details_level_i` dictionary:

```
{'aad', 'ada', 'daa', 'add', 'dad', 'dda', 'ddd'}
```

where the order of the characters in each key map to the specified `axes`.

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'*, *axes=None*)
   Multilevel nD Inverse Discrete Wavelet Transform.

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

   wavelet [Wavelet object or name string, or tuple of wavelets] Wavelet to use. This can also be a tuple containing a wavelet to apply along each axis in `axes`.

   mode [str or tuple of str, optional] Signal extension mode, see Modes (default: 'symmetric'). This can also be a tuple containing a mode to apply along each axis in `axes`.

   axes [sequence of ints, optional] Axes over which to compute the IDWT. Axes may not be repeated.

   Returns nD array of reconstructed data.

Notes

It may sometimes be desired to run `waverecn` with some sets of coefficients omitted. This can best be done by setting the corresponding arrays to zero arrays of matching shape and dtype. Explicitly removing list or dictionary entries or setting them to `None` is not supported.

Specifically, to ignore all detail coefficients at level 2, one could do:

```
coeffs[-2] = {k: np.zeros_like(v) for k, v in coeffs[-2].items()}
```

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 Handling DWT Coefficients

Convenience routines are available for converting the outputs of the multilevel dwt functions (`wavedec`, `wavedec2` and `wavedecn`) to and from a single, concatenated coefficient array.

Concatenating all coefficients into a single array

pywt.**coeffs_to_array**(*coeffs*, *padding=0*, *axes=None*)
    Arrange a wavelet coefficient list from *wavedecn* into a single array.

> **Parameters coeffs: array-like**
>
>> dictionary of wavelet coefficients as returned by pywt.wavedecn
>
> **padding** : float or None, optional
>
>> If None, raise an error if the coefficients cannot be tightly packed.
>
> **axes** : sequence of ints, optional
>
>> Axes over which the DWT that created `coeffs` was performed. The default value of `None` corresponds to all axes.
>
> **Returns coeff_arr** : array-like
>
>> Wavelet transform coefficient array.
>
> **coeff_slices** : list

List of slices corresponding to each coefficient. As a 2D example, *coeff_arr[coeff_slices[1]['dd']]* would extract the first level detail coefficients from *coeff_arr*.

See also:

array_to_coeffs the inverse of coeffs_to_array

Notes

Assume a 2D coefficient dictionary, c, from a two-level transform.

Then all 2D coefficients will be stacked into a single larger 2D array as follows:

```
+---------------+---------------+---------------------------+
|               |               |                           |
|     c[0]      |  c[1]['da']   |                           |
|               |               |                           |
+---------------+---------------+        c[2]['da']         |
|               |               |                           |
|  c[1]['ad']   |  c[1]['dd']   |                           |
|               |               |                           |
+---------------+---------------+---------------------------+
|                               |                           |
|                               |                           |
|                               |                           |
|          c[2]['ad']           |        c[2]['dd']         |
|                               |                           |
|                               |                           |
|                               |                           |
+-------------------------------+---------------------------+
```

Examples

```
>>> import pywt
>>> cam = pywt.data.camera()
>>> coeffs = pywt.wavedecn(cam, wavelet='db2', level=3)
>>> arr, coeff_slices = pywt.coeffs_to_array(coeffs)
```

Splitting concatenated coefficient array back into its components

pywt.**array_to_coeffs**(*arr*, *coeff_slices*, *output_format='wavedecn'*)
Convert a combined array of coefficients back to a list compatible with *waverecn*.

Parameters arr: array-like

An array containing all wavelet coefficients. This should have been generated via *coeffs_to_array*.

coeff_slices : list of tuples

List of slices corresponding to each coefficient as obtained from *array_to_coeffs*.

output_format : {'wavedec', 'wavedec2', 'wavedecn'}

Make the form of the coefficients compatible with this type of multilevel transform.

Returns coeffs: array-like

Wavelet transform coefficient array.

See also:

coeffs_to_array the inverse of array_to_coeffs

Notes

A single large array containing all coefficients will have subsets stored, into a *waverecn* list, c, as indicated below:

```
+--------------+--------------+---------------------------+
|              |              |                           |
|     c[0]     |  c[1]['da']  |                           |
|              |              |                           |
+--------------+--------------+            c[2]['da']     |
|              |              |                           |
| c[1]['ad']   |  c[1]['dd']  |                           |
|              |              |                           |
+--------------+--------------+ --------------------------+
|                             |                           |
|                             |                           |
|                             |                           |
|          c[2]['ad']         |         c[2]['dd']        |
|                             |                           |
|                             |                           |
|                             |                           |
+-----------------------------+---------------------------+
```
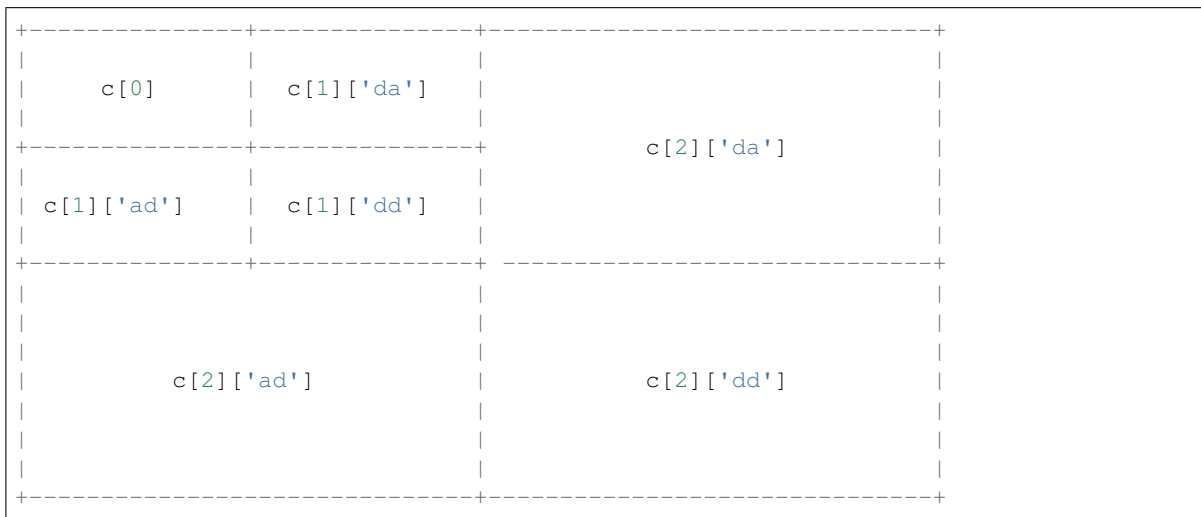
Examples

```
>>> import pywt
>>> from numpy.testing import assert_array_almost_equal
>>> cam = pywt.data.camera()
>>> coeffs = pywt.wavedecn(cam, wavelet='db2', level=3)
>>> arr, coeff_slices = pywt.coeffs_to_array(coeffs)
>>> coeffs_from_arr = pywt.array_to_coeffs(arr, coeff_slices)
>>> cam_recon = pywt.waverecn(coeffs_from_arr, wavelet='db2')
>>> assert_array_almost_equal(cam, cam_recon)
```

10.1.8 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 1D `swt`

pywt.**swt** (*data*, *wavelet*, *level=None*, *start_level=0*, *axis=-1*)
    Multilevel 1D stationary wavelet transform.

Parameters data :

Input signal

wavelet :

Wavelet to use (Wavelet object or name)

level : int, optional

The number of decomposition steps to perform.

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)

axis: int, optional

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

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

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

Notes

The implementation here follows the "algorithm a-trous" and requires that the signal length along the transformed axis be a multiple of `2**level`. If this is not the case, the user should pad up to an appropriate size using a function such as `numpy.pad`.

Multilevel 2D `swt2`

pywt.**swt2**(*data*, *wavelet*, *level*, *start_level=0*, *axes=(-2, -1)*)

Multilevel 2D stationary wavelet transform.

Parameters data : array_like

2D array with input data

wavelet : Wavelet object or name string, or 2-tuple of wavelets

Wavelet to use. This can also be a tuple of wavelets to apply per axis in `axes`.

level : int

The number of decomposition steps to perform.

start_level : int, optional

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

axes : 2-tuple of ints, optional

Axes over which to compute the SWT. Repeated elements are not allowed.

Returns coeffs : list

Approximation and details coefficients:

```
[
    (cA_m,
        (cH_m,  cV_m,  cD_m)
    ),
    (cA_m+1,
        (cH_m+1,  cV_m+1,  cD_m+1)
    ),
    ...,
    (cA_m+level,
        (cH_m+level,  cV_m+level,  cD_m+level)
    )
]
```

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

Notes

The implementation here follows the "algorithm a-trous" and requires that the signal length along the transformed axes be a multiple of `2**level`. If this is not the case, the user should pad up to an appropriate size using a function such as `numpy.pad`.

Multilevel n-dimensional `swtn`

pywt.**swtn**(*data*, *wavelet*, *level*, *start_level=0*, *axes=None*)
    n-dimensional stationary wavelet transform.

Parameters data : array_like

n-dimensional array with input data.

wavelet : Wavelet object or name string, or tuple of wavelets

Wavelet to use. This can also be a tuple of wavelets to apply per axis in `axes`.

level : int

The number of decomposition steps to perform.

start_level : int, optional

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

axes : sequence of ints, optional

Axes over which to compute the SWT. A value of `None` (the default) selects all axes. Axes may not be repeated.

Returns [{coeffs_level_n}, ..., {coeffs_level_1}]: list of dict

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

For example, for a 2D case the result at a given level 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
}
```

For user-specified `axes`, the order of the characters in the dictionary keys map to the specified `axes`.

Notes

The implementation here follows the "algorithm a-trous" and requires that the signal length along the transformed axes be a multiple of `2**level`. If this is not the case, the user should pad up to an appropriate size using a function such as `numpy.pad`.

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.

Notes

For the current implementation of the stationary wavelet transform, this corresponds to the number of times `input_len` is evenly divisible by two. In other words, for an n-level transform, the signal length must be a multiple of `2**n`. `numpy.pad` can be used to pad a signal up to an appropriate length as needed.

10.1.9 Inverse Stationary Wavelet Transform

Inverse *stationary wavelet transforms* are provided.

Note: These inverse transforms are not yet optimized for speed. Only, the n-dimensional inverse transform currently has `axes` support.

Multilevel 1D `iswt`

pywt.**iswt**(*coeffs*, *wavelet*)

Multilevel 1D inverse discrete stationary wavelet transform.

> **Parameters coeffs** : array_like
>
> > Coefficients list of tuples:
> >
> > ```
> > [(cAn, cDn), ..., (cA2, cD2), (cA1, cD1)]
> > ```

where cA is approximation, cD is details. Index 1 corresponds to `start_level` from `pywt.swt`.

wavelet : Wavelet object or name string

Wavelet to use

Returns 1D array of reconstructed data.

Examples

```
>>> import pywt
>>> coeffs = pywt.swt([1,2,3,4,5,6,7,8], 'db2', level=2)
>>> pywt.iswt(coeffs, 'db2')
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
```

Multilevel 2D `iswt2`

pywt.**iswt2**(*coeffs*, *wavelet*)

Multilevel 2D inverse discrete stationary wavelet transform.

Parameters coeffs : list

Approximation and details coefficients:

```
[
    (cA_1,
        (cH_1, cV_1, cD_1)
    ),
    (cA_2,
        (cH_2, cV_2, cD_2)
    ),
    ...,
    (cA_n
        (cH_n, cV_n, cD_n)
    )
]
```

where cA is approximation, cH is horizontal details, cV is vertical details, cD is diagonal details and n is the number of levels. Index 1 corresponds to `start_level` from `pywt.swt2`.

wavelet : Wavelet object or name string, or 2-tuple of wavelets

Wavelet to use. This can also be a 2-tuple of wavelets to apply per axis.

Returns 2D array of reconstructed data.

Examples

```
>>> import pywt
>>> coeffs = pywt.swt2([[1,2,3,4],[5,6,7,8],
...                     [9,10,11,12],[13,14,15,16]],
...                     'db1', level=2)
>>> pywt.iswt2(coeffs, 'db1')
array([[  1.,   2.,   3.,   4.],
```

```
       [  5.,    6.,    7.,    8.],
       [  9.,   10.,   11.,   12.],
       [ 13.,   14.,   15.,   16.]])
```

Multilevel n-dimensional `iswtn`

pywt.**iswtn**(*coeffs*, *wavelet*, *axes=None*)

 Multilevel nD inverse discrete stationary wavelet transform.

  Parameters coeffs : list

    [{coeffs_level_n}, ..., {coeffs_level_1}]: list of dict

   wavelet : Wavelet object or name string, or tuple of wavelets

    Wavelet to use. This can also be a tuple of wavelets to apply per axis in `axes`.

   axes : sequence of ints, optional

    Axes over which to compute the inverse SWT. Axes may not be repeated. The default is `None`, which means transform all axes (`axes = range(data.ndim)`).

  Returns nD array of reconstructed data.

Examples

```
>>> import pywt
>>> coeffs = pywt.swtn([[1,2,3,4],[5,6,7,8],
...                     [9,10,11,12],[13,14,15,16]],
...                     'db1', level=2)
>>> pywt.iswtn(coeffs, 'db1')
array([[  1.,    2.,    3.,    4.],
       [  5.,    6.,    7.,    8.],
       [  9.,   10.,   11.,   12.],
       [ 13.,   14.,   15.,   16.]])
```

10.1.10 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 key-word 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 key-word 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*$\big[$, *order="natural"*$\big[$, *decompose=True*$\big]\big]$)

    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.

10.1.11 Continuous Wavelet Transform (CWT)

This section describes functions used to perform single continuous wavelet transforms.

Single level - `cwt`

pywt.**cwt** (*data*, *scales*, *wavelet*)

    One dimensional Continuous Wavelet Transform.

> **Parameters** **data** : array_like
>
> > Input signal
>
> **scales** : array_like
>
> > scales to use
>
> **wavelet** : Wavelet object or name
>
> > Wavelet to use
>
> **sampling_period** : float
>
> > Sampling period for frequencies output (optional)
>
> **Returns** **coefs** : array_like
>
> > Continuous wavelet transform of the input signal for the given scales and wavelet
>
> **frequencies** : array_like
>
> > if the unit of sampling period are seconds and given, than frequencies are in hertz. Otherwise Sampling period of 1 is assumed.

Notes

Size of coefficients arrays depends on the length of the input array and the length of given scales.

Examples

```
>>> import pywt
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.arange(512)
>>> y = np.sin(2*np.pi*x/32)
>>> coef, freqs=pywt.cwt(y,np.arange(1,129),'gaus1')
>>> plt.matshow(coef)
>>> plt.show()
----------
>>> import pywt
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(-1, 1, 200, endpoint=False)
>>> sig  = np.cos(2 * np.pi * 7 * t) + np.real(np.exp(-7*(t-0.4)**2)*np.
→exp(1j*2*np.pi*2*(t-0.4)))
>>> widths = np.arange(1, 31)
>>> cwtmatr, freqs = pywt.cwt(sig, widths, 'mexh')
>>> plt.imshow(cwtmatr, extent=[-1, 1, 1, 31], cmap='PRGn', aspect='auto',
...            vmax=abs(cwtmatr).max(), vmin=-abs(cwtmatr).max())
>>> plt.show()
```

10.1.12 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, data values with absolute value less than *param* are replaced with *substitute*. Data values with absolute value greater or equal to the thresholding value are shrunk toward zero by *value*. In other words, the new value is `data/np.abs(data) * np.maximum(np.abs(data) - value, 0)`.

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. Lesser data values pass untouched.

Both `hard` and `soft` thresholding also support complex-valued data.

> **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. ])
```

10.1.13 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**(*filt*)
    Returns the Quadrature Mirror Filter(QMF).

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

> **Parameters filt** : 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)
nino	sea surface temperature (264 samples)

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', 'gaus', 'mexh', 'morl', 'cgau',
↪'shan', 'fbsp', 'cmor']
```

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, db17, db18, db19, db20, db21, db22, db23, db24, db25, db26, db27, db28,
↪db29, db30, db31, db32, db33, db34, db35, db36, db37, db38
sym family: sym2, sym3, sym4, sym5, sym6, sym7, sym8, sym9, sym10, sym11, sym12,
↪sym13, sym14, sym15, sym16, sym17, sym18, sym19, sym20
coif family: coif1, coif2, coif3, coif4, coif5, coif6, coif7, coif8, coif9, coif10,
↪coif11, coif12, coif13, coif14, coif15, coif16, coif17
bior family: bior1.1, bior1.3, bior1.5, bior2.2, bior2.4, bior2.6, bior2.8, bior3.1,
↪bior3.3, bior3.5, bior3.7, bior3.9, bior4.4, bior5.5, bior6.8
rbio family: rbio1.1, rbio1.3, rbio1.5, rbio2.2, rbio2.4, rbio2.6, rbio2.8, rbio3.1,
↪rbio3.3, rbio3.5, rbio3.7, rbio3.9, rbio4.4, rbio5.5, rbio6.8
dmey family: dmey
gaus family: gaus1, gaus2, gaus3, gaus4, gaus5, gaus6, gaus7, gaus8
mexh family: mexh
morl family: morl
cgau family: cgau1, cgau2, cgau3, cgau4, cgau5, cgau6, cgau7, cgau8
shan family: shan
fbsp family: fbsp
cmor family: cmor
```

To get the full list of builtin wavelets' names just use the `wavelist()` with no argument.

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
  DWT:            True
  CWT:            False
```

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]))
```

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
  DWT:            True
  CWT:            False
```

You can however set a couple 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
  DWT:            True
  CWT:            False
```

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', 'reflect']
```

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:

```
>>> x = [1, 2, 1, 5, -1, 8, 4, 6]
>>> for mode_name in ['zero', 'constant', 'symmetric', 'reflect', 'periodic', 'smooth
→', 'periodization']:
...     mode = getattr(pywt.Modes, mode_name)
...     cA, cD = pywt.dwt(x, 'db2', mode)
...     print("Mode: %d (%s)" % (mode, mode_name))
Mode: 0 (zero)
Mode: 2 (constant)
Mode: 1 (symmetric)
Mode: 6 (reflect)
Mode: 4 (periodic)
Mode: 3 (smooth)
Mode: 5 (periodization)
```

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 length` 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', 'reflect']
```

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 same as used for decomposition.
```

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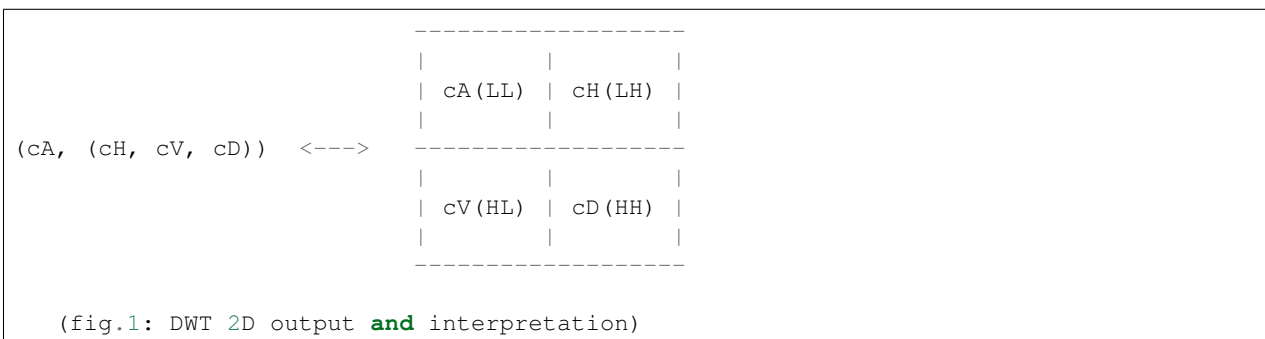
Knowing what the nodes names are, we can now access them using the indexing operator `obj[x]` (*WaveletPacket2D.__getitem__()*):

```
>>> 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
```

Numpy can also be obtained via scientific python distributions such as:

- Anaconda

- Enthought Canopy

- Python(x,y)

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 continuous 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()
```

Note doctests require Matplotlib in addition to the usual dependencies.

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 Guidelines for Releasing PyWavelets

The following are guidelines for preparing a release of PyWavelets. The notation `vX.X.X` in the commands below would be replaced by the actual release number.

Updating the release notes

Prior to the release, make sure the release notes are up to date. The author lists can be generated via:

```
python  ./util/authors.py vP.P.P..
```

where `vP.P.P` is the previous release number.

The lists of issues closed and PRs merged can be generated via (script requires Python 2.X to run):

```
python ./util/gh_lists.py vX.X.X
```

Tag the release

Change `ISRELEASED` to `True` in `setup.py` and commit.

Tag the release via:

```
git tag -s vX.X.X
```

Then push the `vX.X.X` tag to the PyWavelets GitHub repo.

Note that while Appveyor will build wheels for Windows, it is preferred to get those wheels from the step below. Instructions for grabbing Appveyor wheels manually here for reference only: if the commit with `ISRELEASED=True` is submitted as a PR, the wheels can be downloaded from Appveyor once it has run on the PR. They can be found under the "Artifacts" tab in the Appveyor interface.

Build Windows, OS X and Linux wheels and upload to PyPI

Push a commit with the new tag and updates of dependency versions where needed to https://github.com/MacPython/pywavelets-wheels. The wheels will be produced automatically and uploaded to http://wheels.scipy.org/. From there they can be uploaded to PyPI automatically with `wheel-uploader`.

See the README on https://github.com/MacPython/pywavelets-wheels for more details.

Create the source distribution

Remove untracked files and directories with `git clean`. *Warning: this will delete files & directories that are not under version control so you may want to do a dry run first by adding -n, so you can see what will be removed*:

```
git clean -xfdn
```

Then run without `-n`:

```
git clean -xfd
```

Create the source distribution files via:

```
python setup.py sdist --formats=gztar,zip
```

Upload the release to PyPI

The binary Windows wheels downloaded from Appveyor (see above) should also be placed into the `/dist` subfolder along with the sdist archives.

The wheels and source distributions created above can all be securely uploaded to pypi.python.org using twine:

```
twine upload -s dist/*
```

Note that the documentation on ReadTheDocs (http://pywavelets.readthedocs.org) will have been automatically generated, so no actions need to be taken for documentation.

Update conda-forge

Send a PR with the new version number and `sha256` hash of the source release to https://github.com/conda-forge/pywavelets-feedstock.

Create the release on GitHub

On the project's GitHub page, click the releases tab and then press the "*Draft a new release*" button to create a release from the appropriate tag.

Announcing the release

Send release announcements to:

- pywavelets@googlegroups.com
- python-announce-list@python.org
- scipy-user@python.org

Prepare for continued development

Increment the version number in setup.py and change ISRELEASED to False.

Prepare new release note files for the upcoming release:

```
git add doc/release/X.X.X-notes.rst
git add doc/source/release.X.X.X.rst
```

And add `release.X.X.X` to the list in `doc/source/releasenotes.rst`

10.3.7 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 Release Notes

PyWavelets 0.3.0 Release Notes

Contents

- *PyWavelets 0.3.0 Release Notes*
  - *New features*
    * *Test suite*
    * *n-D Inverse Discrete Wavelet Transform*

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 Release Notes*

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

PyWavelets 0.5.0 Release Notes

Contents

> ∗ *Issues closed for v0.5.0*
>
> ∗ *Pull requests for v0.5.0*

PyWavelets is a Python toolbox implementing both discrete and continuous wavelet transforms (mathematical time-frequency transforms) with a wide range of built-in wavelets. C/Cython are used for the low-level routines, enabling high performance. Key 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 Forward and Inverse Stationary Wavelet Transform

- 1D and 2D Wavelet Packet decomposition and reconstruction

- 1D Continuous Wavelet Transform

- When multiple valid implementations are available, we have chosen to maintain consistency with MATLAB™'s Wavelet Toolbox.

PyWavelets 0.5.0 is the culmination of 1 year of work. In addition to several new features, substantial refactoring of the underlying C and Cython code have been made.

This release requires Python 2.6, 2.7 or 3.3-3.5 and NumPy 1.9.1 or greater. This will be the final release supporting Python 2.6 and 3.3.

Highlights of this release include:

- 1D continuous wavelet transforms

- new discrete wavelets added (additional Debauchies and Coiflet wavelets)

- new 'reflect' extension mode for discrete wavelet transforms

- faster performance for multilevel forward stationary wavelet transforms (SWT)

- n-dimensional support added to forward SWT

- routines to convert multilevel DWT coefficients to and from a single array

- axis support for multilevel DWT

- substantial refactoring/reorganization of the underlying C and Cython code

New features

1D Continous Wavelet Transforms

A wide range of continous wavelets are now available. These include the following:

- Gaussian wavelets (`gaus1`... ``gaus8``)

- Mexican hat wavelet (`mexh`)

- Morlet wavelet (`morl`)

- Complex Gaussian wavelets (`cgau1`... ``cgau8``)

- Shannon wavelet (`shan`)

- Frequency B-Spline wavelet (`fbsp`)

- Complex Morlet wavelet (`cmor`)

Also, see the new CWT-related demo: demo/cwt_analysis.py

New discrete wavelets

Additional Debauchies wavelets (`db20`... ``db38``) and Coiflets (`coif6`... ``coif17``) have been added.

New extension mode: reflect

Discrete wavelet transforms support a new extension mode, `reflect`. This mode pads an array symmetrically, but without repeating the edge value. As an example:

```
 pad       array       pad
4 3 2 | 1 2 3 4 5 | 4 3 2
```

This differs from `symmetric`, which repeats the values at the boundaries:

```
 pad       array       pad
3 2 1 | 1 2 3 4 5 | 5 4 3
```

Multilevel DWT Coefficient Handling

New routines to convert the coefficients returned by multilevel DWT routines to and from a single n-dimensional array have been added. `pywt.coeffs_to_array` concatenates the output of `wavedec`, `wavedec2` or `wavedecn` into a single numpy array. `pywt.array_to_coeffs` can be used to transform back from a single coefficient array to a format appropriate for `waverec`, `waverec2` or `waverecn`.

More C function calls release the GIL

Cython code calling the wavelet filtering routines (DWT and SWT) now releases the global interpreter lock (GIL) where possible. A potential use case is in speeding up the batch computation of several large DWTs using multi-threading (e.g. via concurrent.futures).

Multilevel wavelet transforms along specific axes

The axis specific transform support introduced in the prior release was extended to the multilevel DWT transforms. All `wavedec*` and `waverec*` routines have a new *axis* (1D) or *axes* (2D, nD) keyword argument. If unspecified the default behaviour is to transform all axes of the input.

Faster multilevel stationary wavelet transforms

Stationary wavelet transforms are now faster when the number of levels is greater than one. The improvement can be very large (multiple orders of magnitude) for transforms with a large number of levels.

Deprecated features

Backwards incompatible changes

A FutureWarning was added to `swt2` and `iswt2` to warn about a pending backwards incompatible change to the order of the coefficients in the list returned by these routines. The actual change will not occur until the next release. Transform coefficients will be returned in descending rather than ascending order. This change is being made for consistency with all other existing multi-level transforms in PyWavelets.

Bugs Fixed

`demo/image_blender.py` was updated to support the new api of Pillow 3.x

A bug related to size of assumed `size_t` on some platforms/compilers (e.g. Windows with mingw64) was fixed.

Fix to memory leak in `(i)dwt_axis`

Fix to a performance regression in `idwt` and `iswt` that was introduced in v0.4.0.

Fixed a bug in `dwtn` and `idwtn` for data with complex dtype when `axes != None`.

Other changes

The minimum supported numpy version has been increased to 1.9.1.

Test coverage (including for the Cython and C code) via Codecov was added and the overall test coverage has been improved.

A substantial overhaul of the C extension code has been performed. Custom templating is no longer used. The intention is to make this code easier to maintain and expand in the future.

The Cython code has been split out into a multiple files to hopefully make relevant portions of the wrappers easier to find for future developers.

`setup.py` now relies on setuptools in all cases (rather than distutils).

Authors

- Jonathan Dan +
- Ralf Gommers
- David Menéndez Hurtado
- Gregory R. Lee
- Holger Nahrstaedt +
- Daniel M. Pelt +
- Alexandre Saint +
- Scott Sievert +
- Kai Wohlfahrt
- Frank Yu +

A total of 10 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 v0.5.0

- #48: Continous wavelet transform?
- #127: Reorganize _pywt
- #160: Appveyor failing on recent PRs
- #163: Set up coveralls
- #166: Wavelet coefficients to single array (and vice versa?)
- #177: Fail to install pywt due to the use of index_t which conflict with the defination in /usr/include/sys/types.h on smartos sysmte(open solaris like system)
- #180: Memory leak
- #187: 'reflect' signal extension mode
- #189: bump minimum numpy version?
- #191: Upgrade removed Pillow methods
- #196: building in-place for development.
- #200: swt implementation is considerably slower than MATLAB
- #209: broken doctests
- #210: Run doctests in CI setup
- #211: Typo in iswt documentation
- #217: *blank_discrete_wavelet* does not properly intiailize some properties
- #231: I can't compile pywt

Pull requests for v0.5.0

- #148: Reorganize C v2
- #161: Remove numpy distutils
- #162: fix: iswt/idwt performance regression
- #164: Improved coefficients for db and coif
- #167: Add coverage (codecov.io)
- #168: convert transform coefficients to and from a single n-dimensional array
- #169: Remove templating
- #170: :Always install new pip on Appveyor
- #172: Adding of missing wavelets from the matlab list
- #178: use Index_t instead of index_t
- #179: add axis/axes support to multilevel discrete wavelet transforms
- #181: Fix memory leak

- #182: improve test coverage for _multidim.py and _multilevel.py
- #183: improve coverage for _dwt.py
- #184: fix corner case in coeffs_to_array
- #188: Drop GIL in c_wt calls
- #190: bump minimum numpy to 1.9
- #192: Upgrade to Pillow>=3 api
- #193: ENH: add 'reflect' extension mode
- #197: BLD: fix "python setup.py develop". Closes gh-196
- #198: Choose clz* based on SIZE_MAX
- #201: speedup multi-level swt
- #205: fix dwtn/idwtn with axes != None and complex data
- #206: DOC: correct typo in iswt docstring
- #207: minor documentation updates
- #208: document coeff_to_array and array_to_coeff
- #214: FIX: update several doctests to reflect the new wavelets added
- #218: FIX: initialize all properties of a blank discrete wavelet
- #219: document coordinate conventions for 2D DWT routines.
- #220: Run doctests on TravisCI
- #221: Documentation for cwt and ContinuousWavelet
- #222: consistent use of double backticks in docs
- #223: add FutureWarning about swt2 coefficient order
- #224: n-dimensional stationary wavelet transform (swtn) and axis support in swt, swt2
- #225: BUG: fix breakage on 32-bit Python.
- #226: DOC: update Copyright statements.
- #227: ENH: add kind keyword to wavelist()
- #228: MAINT: avoid using a builtin as variable name in qmf().
- #229: DOC: add swtn, iswt, iswt2 to the API documentation
- #230: add demo of batch processing via concurrent.futures
- #234: ENH: coeffs_to_array supports axes argument as recently added to wavedec*
- #236: BLD: raise an ImportError if Cython should be installed but isn't.

PyWavelets 1.0.0 Release Notes

Contents

PyWavelets is a Python toolbox implementing both discrete and continuous wavelet transforms (mathematical time-frequency transforms) with a wide range of built-in wavelets. C/Cython are used for the low-level routines, enabling high performance. Key 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 Forward and Inverse Stationary Wavelet Transform

- 1D and 2D Wavelet Packet decomposition and reconstruction

- 1D Continuous Wavelet Transform

- When multiple valid implementations are available, we have chosen to maintain consistency with MATLAB™'s Wavelet Toolbox.

New features

Deprecated features

Backwards incompatible changes

Python 2.6 and 3.3 are no longer supported.

Bugs Fixed

Other changes

Authors

Issues closed for v1.0.0

Pull requests for v1.0.0

10.6 Indices and tables

- genindex

- search

Index

Symbols

A

B

C

D

F

G

H

I

L

M

N