PyWavelets Documentation

Release 0.3.0

The PyWavelets Developers

July 31, 2015

Contents

1 Main features 3

2 Requirements 5

3 Download 7

4 Install 9

5 Documentation 11

6 State of development & Contributing 13

7 Python 3 15

8 Contact 17

9 License 19

10 Contents 21
- 10.1 API Reference 21
- 10.2 Usage examples 41
- 10.3 Development notes 61
- 10.4 Resources 65
- 10.5 PyWavelets 66
- 10.6 Indices and tables 113

PyWavelets is a free Open Source wavelet transform software for Python programming language. It is written in Python, Cython and C for a mix of easy and powerful high-level interface and the best performance.

PyWavelets is very easy to start with and use. 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 never before has been so simple :)

CHAPTER 1

Main features

The main features of PyWavelets are:

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

CHAPTER 2

Requirements

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

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

CHAPTER 3

Download

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

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

CHAPTER 4

Install

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

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

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

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

See also:

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

CHAPTER 5

Documentation

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

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

CHAPTER 6

State of development & Contributing

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

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

CHAPTER 7

Python 3

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

CHAPTER 8

Contact

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

CHAPTER 9

License

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

CHAPTER 10

Contents

10.1 API Reference

10.1.1 Wavelets

Wavelet `families()`

`pywt.families()`

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`)

Example:

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

Built-in wavelets - `wavelist()`

`pywt.wavelist([family])`

The *`wavelist()`* function returns a list of names of the built-in wavelets.

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

Example:

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

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

`Wavelet` object

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

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

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

Parameters

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

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

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

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

Example:

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

name
Wavelet name.

short_name
Short wavelet name.

dec_lo
Decomposition filter values.

dec_hi
Decomposition filter values.

rec_lo
Reconstruction filter values.

rec_hi
Reconstruction filter values.

dec_len
Decomposition filter length.

rec_len
Reconstruction filter length.

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

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

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

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

short_family_name
Wavelet short family name

family_name
Wavelet family name

orthogonal
Set if wavelet is orthogonal

biorthogonal
Set if wavelet is biorthogonal

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

vanishing_moments_psi
Number of vanishing moments for the wavelet function

vanishing_moments_phi
Number of vanishing moments for the scaling function

Example:

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

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

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

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

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

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

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

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

Example:

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

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

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

Example:

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

See also:

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

Using custom wavelets

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

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

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

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

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

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

Example:

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

10.1.2 Signal extension modes

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

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

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

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

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

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

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

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

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

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

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

- `sp1` - **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:

- `per` - **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
['zpd', 'cpd', 'sym', 'ppd', 'sp1', 'per']
```

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', 'sp1')
>>> (a, d) = pywt.dwt([1,2,3,4,5,6], pywt.Wavelet('db2'), pywt.MODES.sp1)
```

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

10.1.3 Discrete Wavelet Transform (DWT)

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

Single level `dwt`

pywt.**dwt**(*data*, *wavelet*[, *mode='sym'*])

The *dwt()* function is used to perform single level, one dimensional Discrete Wavelet Transform.

```
(cA, cD) = dwt(data, wavelet, mode='sym')
```

Parameters

- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **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 to deal with the border distortion problem. See *MODES* for details.

The transform coefficients are returned as two arrays containing approximation (*cA*) and detail (*cD*) coefficients respectively. Length of returned arrays depends on the selected signal extension *mode* - 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:

- 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)
```

Example:

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

Multilevel decomposition using `wavedec`

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

The *wavedec()* function performs 1D multilevel Discrete Wavelet Transform decomposition of given signal and returns ordered list of coefficients arrays in the form:

```
[cA_n, cD_n, cD_n-1, ..., cD2, cD1],
```

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.

Parameters

- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.

- **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 to deal with the border distortion problem. See *MODES* for details.
- **level** – Number of decomposition steps to perform. If the level is `None`, then the full decomposition up to the level computed with *dwt_max_level()* function for the given data and wavelet lengths is performed.

Example:

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

Partial Discrete Wavelet Transform data decomposition `downcoef`

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

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

Parameters

- **part** – decomposition type. For `a` computes approximation coefficients, for `d` - details coefficients.
- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **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 to deal with the border distortion problem. See *MODES* for details.
- **level** – Number of decomposition steps to perform.

Maximum decomposition level - `dwt_max_level`

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

The *dwt_max_level()* function can be used to compute the maximum *useful* level of decomposition for the given *input data length* and *wavelet filter length*.

The returned value equals to:

```
floor( log(data_len/(filter_len-1)) / log(2) )
```

Although the maximum decomposition level can be quite high for long signals, usually smaller values are chosen depending on the application.

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

Example:

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

Result coefficients length - `dwt_coeff_len`

`pywt.`**`dwt_coeff_len`**`(`*data_len*, *filter_len*, *mode*`)`

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

For *periodization* mode this equals:

```
ceil(data_len / 2)
```

which is the lowest possible length guaranteeing perfect reconstruction.

For other *modes*:

```
floor((data_len + filter_len - 1) / 2)
```

The *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='sym'*[, *correct_size=0*]]`)`

The `idwt()` function reconstructs data from the given coefficients by performing single level Inverse Discrete Wavelet Transform.

Parameters

- **`cA`** – Approximation coefficients.
- **`cD`** – Detail coefficients.
- **`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 to deal with the border distortion problem. See *MODES* for details. This is only important when DWT was performed in *periodization* mode.
- **`correct_size`** – Typically, *cA* and *cD* coefficients lists must have equal lengths in order to perform IDWT. Setting *correct_size* to *True* allows *cA* to be greater in size by one element compared to the *cD* size. This option is very useful when doing multilevel decomposition and reconstruction (as for example with the `wavedec()` function) of non-dyadic length signals when such minor differences can occur at various levels of IDWT.

Example:

```
>>> import pywt
>>> (cA, cD) = pywt.dwt([1,2,3,4,5,6], 'db2', 'sp1')
>>> print pywt.idwt(cA, cD, 'db2', 'sp1')
[ 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', 'sp1')
>>> A = pywt.idwt(cA, None, 'db2', 'sp1')
>>> D = pywt.idwt(None, cD, 'db2', 'sp1')
>>> print A + D
[ 1.  2.  3.  4.  5.  6.]
```

Multilevel reconstruction using `waverec`

`pywt.`**`waverec`**(*coeffs*, *wavelet*[, *mode='sym'*])

Performs multilevel reconstruction of signal from the given list of coefficients.

Parameters

- **`coeffs`** – Coefficients list must be in the form like returned by `wavedec()` decomposition function, which is:

```
[cAn, cDn, cDn-1, ..., cD2, cD1]
```

- **`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 to deal with the border distortion problem. See *MODES* for details.

Example:

```
>>> import pywt
>>> coeffs = pywt.wavedec([1,2,3,4,5,6,7,8], 'db2', level=2)
>>> print pywt.waverec(coeffs, 'db2')
[ 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`** – Defines the input coefficients type:
  - **'a'** - approximations reconstruction is performed
  - **'d'** - details reconstruction is performed
- **`coeffs`** – Coefficients array to reconstruct.

- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the *wavelist()* list or a *Wavelet* object instance.
- **level** – If *level* value is specified then a multilevel reconstruction is performed (first reconstruction is of type specified by *part* and all the following ones with *part* type `a`)
- **take** – If *take* is specified then only the central part of length equal to the *take* parameter value is returned.

Example:

```
>>> import pywt
>>> data = [1,2,3,4,5,6]
>>> (cA, cD) = pywt.dwt(data, 'db2', 'sp1')
>>> print pywt.upcoef('a', cA, 'db2') + pywt.upcoef('d', cD, 'db2')
[-0.25        -0.4330127   1.          2.          3.          4.          5.
  6.          1.78589838 -1.03108891]
>>> n = len(data)
>>> print pywt.upcoef('a',cA,'db2',take=n) + pywt.upcoef('d',cD,'db2',take=n)
[ 1.  2.  3.  4.  5.  6.]
```

10.1.5 2D Forward and Inverse Discrete Wavelet Transform

Single level dwt2

`pywt.dwt2(data, wavelet[, mode='sym'])`

The *dwt2()* function performs single level 2D Discrete Wavelet Transform.

Parameters

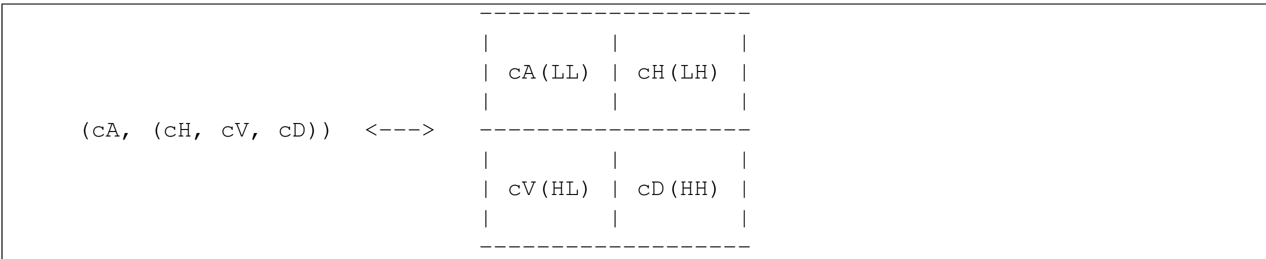
- **data** – 2D input data.
- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the *wavelist()* list or a *Wavelet* object instance.
- **mode** – Signal extension mode to deal with the border distortion problem. See *MODES* for details. This is only important when DWT was performed in *periodization* mode.

Returns one average and three details 2D coefficients arrays. The coefficients arrays are organized in tuples in the following form:

```
(cA, (cH, cV, cD))
```

where *cA*, *cH*, *cV*, *cD* denote approximation, horizontal detail, vertical detail and diagonal detail coefficients respectively.

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.

Example:

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

Single level `idwt2`

`pywt.idwt2(coeffs, wavelet[, mode='sym'])`

The `idwt2()` function reconstructs data from the given coefficients set by performing single level 2D Inverse Discrete Wavelet Transform.

Parameters

- **coeffs** – A tuple with approximation coefficients and three details coefficients 2D arrays like from `dwt2()`:

```
(cA, (cH, cV, cD))
```

- **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 to deal with the border distortion problem. See *MODES* for details. This is only important when the `dwt()` was performed in the *periodization* mode.

Example:

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

2D multilevel decomposition using `wavedec2`

`pywt.wavedec2(data, wavelet[, mode='sym'[, level=None]])`

Performs multilevel 2D Discrete Wavelet Transform decomposition and returns coefficients list:

```
[cAn, (cHn, cVn, cDn), ..., (cH1, cV1, cD1)]
```

where *n* denotes the level of decomposition and *cA*, *cH*, *cV* and *cD* are approximation, horizontal detail, vertical detail and diagonal detail coefficients arrays respectively.

Parameters

- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **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 to deal with the border distortion problem. See *MODES* for details.
- **level** – Decomposition level. This should not be greater than the reasonable maximum value computed with the `dwt_max_level()` function for the smaller dimension of the input data.

Example:

```
>>> import pywt, numpy
>>> coeffs = pywt.wavedec2(numpy.ones((8,8)), 'db1', level=2)
>>> cA2, (cH2, cV2, cD2), (cH1, cV1, cD1) = coeffs
>>> print cA2
[[ 4.  4.]
 [ 4.  4.]]
```

2D multilevel reconstruction using `waverec2`

`pywt.`**`waverec2`**(*coeffs*, *wavelet*[, *mode='sym'*])

Performs multilevel reconstruction from the given coefficients set.

Parameters

- **coeffs** – Coefficients set must be in the form like that from `wavedec2()` decomposition:

```
[cAn, (cHn, cVn, cDn), ..., (cH1, cV1, cD1)]
```

- **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 to deal with the border distortion problem. See *MODES* for details.

Example:

```
>>> import pywt, numpy
>>> coeffs = pywt.wavedec2(numpy.ones((4,4)), 'db1')
>>> print "levels:", len(coeffs)-1
levels: 2
>>> print pywt.waverec2(coeffs, 'db1')
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

10.1.6 Stationary Wavelet Transform

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

Multilevel `swt`

`pywt.swt(data, wavelet, level[, start_level=0])`

Performs multilevel Stationary Wavelet Transform.

Parameters

- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the `wavelist()` list or a `Wavelet` object instance.
- **level** (*int*) – Required transform level. See the `swt_max_level()` function.
- **start_level** (*int*) – The level at which the decomposition will begin (it allows to skip a given number of transform steps and compute coefficients starting directly from the *start_level*)

Returns list of coefficient pairs in the form:

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

where *n* is the *level* value.

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

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

Multilevel `swt2`

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

Performs multilevel 2D Stationary Wavelet Transform.

Parameters

- **data** – 2D array with input data.
- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the `wavelist()` list or a `Wavelet` object instance.
- **level** – Number of decomposition steps to perform.
- **start_level** – The level at which the decomposition will begin.

The result is a set of coefficients arrays over the range of decomposition levels:

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

where *cA* is approximation, *cH* is horizontal details, *cV* is vertical details, *cD* is diagonal details, *n* is *start_level* and *m* equals *n+level*.

Maximum decomposition level - `swt_max_level`

`pywt.swt_max_level(input_len)`
Calculates the maximum level of Stationary Wavelet Transform for data of given length.

Parameters **`input_len`** – Input data length.

10.1.7 Wavelet Packets

New in version 0.2.

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

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

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

The below diagram illustrates the inheritance tree:

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

BaseNode - a common interface of WaveletPacket and WaveletPacket2D

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

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

`__init__(parent, data, node_name)`

Parameters

- **`parent`** – parent node. If parent is `None` then the node is considered detached.
- **`data`** – data associated with the node. 1D or 2D numeric array, depending on the transform type.

- **node_name** – a name identifying the coefficients type. See `Node.node_name` and `Node2D.node_name` for information on the accepted subnodes names.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

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

Parameters

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

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

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

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

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

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

Parameters

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

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

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

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

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

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

Parameters

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

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

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

Args arguments to pass to the *func*

Kwargs keyword arguments to pass to the *func*

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

Parameters

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

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

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

Args arguments to pass to the *func*

Kwargs keyword arguments to pass to the *func*

WaveletPacket and WaveletPacket tree Node

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

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

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

- a - approximation coefficients
- d - details coefficients

decompose ()

See also:

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

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

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

Parameters

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

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

Collects nodes from the given level of decomposition.

Parameters

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

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

10.1.8 Thresholding functions

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

Hard thresholding

hard(*data*, *value*[, *substitute=0*])

Hard thresholding. Replace all *data* values with *substitute* where their absolute value is less than the *value* param.

Data values with absolute value greater or equal to the thresholding *value* stay untouched.

Parameters

- **data** – numeric data
- **value** – thresholding value
- **substitute** – substitute value

Returns array

Soft thresholding

soft (*data*, *value*[, *substitute=0*])

Soft thresholding.

Parameters

- **data** – numeric data
- **value** – thresholding value
- **substitute** – substitute value

Returns array

Greater

greater (*data*, *value*[, *substitute=0*])

Replace *data* with *substitute* where *data* is below the thresholding *value*.

Greater data values pass untouched.

Parameters

- **data** – numeric data
- **value** – thresholding value
- **substitute** – substitute value

Returns array

Less

less (*data*, *value*[, *substitute=0*])

Replace *data* with *substitute* where *data* is above the thresholding *value*.

Less data values pass untouched.

Parameters

- **data** – numeric data
- **value** – thresholding value
- **substitute** – substitute value

Returns array

10.1.9 Other functions

Single-level n-dimensional Discrete Wavelet Transform.

pywt.**dwtn**(*data*, *wavelet*[, *mode='sym'*])

Performs single-level n-dimensional Discrete Wavelet Transform.

Parameters

- **data** – n-dimensional 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 to deal with the border distortion problem. See *MODES* for details.

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

Integrating wavelet functions - intwave()

pywt.**intwave**(*wavelet*[, *precision=8*])

Integration of wavelet function approximations as well as any other signals can be performed using the *pywt.intwave()* function.

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

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

```
[int_psi, x] = intwave(wavelet, precision)
```

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

```
[int_psi_d, int_psi_r, x] = intwave(wavelet, precision)
```

- for a tuple of coefficients data and a x-grid - an integral of function and the given x-grid is returned (the x-grid is used for computations).:

```
[int_function, x] = intwave((data, x), precision)
```

Example:

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

Central frequency of *psi* wavelet function

`pywt.centfrq(wavelet[, precision=8])`
`pywt.centfrq((function_approx, x))`

Parameters

- **wavelet** – `Wavelet`, wavelet name string or *(wavelet function approx., x grid)* pair
- **precision** – Precision that will be used for wavelet function approximation computed with the `Wavelet.wavefun()` method.

10.2 Usage examples

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

10.2.1 The Wavelet object

Wavelet families and builtin Wavelets names

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

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

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

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

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

```
>>> for family in pywt.families():
...     print "%s family:" % family, ', '.join(pywt.wavelist(family))
haar family: haar
db family: db1, db2, db3, db4, db5, db6, db7, db8, db9, db10, db11, db12, db13, db14, db15, db16, db1
sym family: sym2, sym3, sym4, sym5, sym6, sym7, sym8, sym9, sym10, sym11, sym12, sym13, sym14, sym15,
coif family: coif1, coif2, coif3, coif4, coif5
bior family: bior1.1, bior1.3, bior1.5, bior2.2, bior2.4, bior2.6, bior2.8, bior3.1, bior3.3, bior3.5
rbio family: rbio1.1, rbio1.3, rbio1.5, rbio2.2, rbio2.4, rbio2.6, rbio2.8, rbio3.1, rbio3.3, rbio3.5
dmey family: dmey
```

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

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

Creating Wavelet objects

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

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

So.. that's it.

Wavelet properties

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

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

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

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

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

```
>>> print_array(w.dec_lo)
[0.03522629188210, -0.08544127388224, -0.13501102001039, 0.45987750211933, 0.80689150931334, 0.33267
>>> print_array(w.dec_hi)
[-0.33267055295096, 0.80689150931334, -0.45987750211933, -0.13501102001039, 0.08544127388224, 0.0352
>>> print_array(w.rec_lo)
[0.33267055295096, 0.80689150931334, 0.45987750211933, -0.13501102001039, -0.08544127388224, 0.03522
>>> print_array(w.rec_hi)
[0.03522629188210, 0.08544127388224, -0.13501102001039, -0.45987750211933, 0.80689150931334, -0.3326
```

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

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

Other Wavelet's properties are:

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

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

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

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

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

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

- Symmetry (`symmetry`):

```
>>> print w.symmetry
asymmetric
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

And now... the *wavefun*!

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

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

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

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

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

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

See also:

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

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
['zpd', 'cpd', 'sym', 'ppd', 'sp1', 'per']
```

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

```
>>> x = [1,2,1,5,-1,8,4,6]
>>> for mode in pywt.MODES.modes:
...     cA, cD = pywt.dwt(x, 'db2', mode)
...     print "Mode:", mode
...     print "cA:", format_array(cA)
...     print "cD:", format_array(cD)
...     print "Reconstruction:", pywt.idwt(cA, cD, 'db2', mode)
Mode: zpd
cA: [-0.03468  1.73309  3.40612  6.32929  6.95095]
cD: [-0.12941 -2.156   -5.95035 -1.21545 -1.8625 ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: cpd
cA: [ 1.2848   1.73309  3.40612  6.32929  7.51936]
cD: [-0.48296 -2.156   -5.95035 -1.21545  0.25882]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: sym
cA: [ 1.76777  1.73309  3.40612  6.32929  7.77817]
cD: [-0.61237 -2.156   -5.95035 -1.21545  1.22474]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: ppd
cA: [ 6.91627  1.73309  3.40612  6.32929  6.91627]
cD: [-1.99191 -2.156   -5.95035 -1.21545 -1.99191]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: sp1
cA: [-0.51764  1.73309  3.40612  6.32929  7.45001]
cD: [ 0.      -2.156   -5.95035 -1.21545  0.      ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: per
cA: [ 4.05317  3.05257  2.85381  8.42522]
cD: [ 0.18947  4.18258  4.33738  2.60428]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
```

Invalid mode name should rise a `ValueError`:

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

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

```
>>> for mode_name in ['zpd', 'cpd', 'sym', 'ppd', 'sp1', 'per']:
...     mode = getattr(pywt.MODES, mode_name)
...     cA, cD = pywt.dwt([1,2,1,5,-1,8,4,6], 'db2', mode)
...     print "Mode:", mode, "(%s)" % mode_name
...     print "cA:", format_array(cA)
...     print "cD:", format_array(cD)
...     print "Reconstruction:", pywt.idwt(cA, cD, 'db2', mode)
Mode: 0 (zpd)
cA: [-0.03468  1.73309  3.40612  6.32929  6.95095]
cD: [-0.12941 -2.156   -5.95035 -1.21545 -1.8625 ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 2 (cpd)
cA: [ 1.2848   1.73309  3.40612  6.32929  7.51936]
cD: [-0.48296 -2.156   -5.95035 -1.21545  0.25882]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 1 (sym)
cA: [ 1.76777  1.73309  3.40612  6.32929  7.77817]
cD: [-0.61237 -2.156   -5.95035 -1.21545  1.22474]
```

```
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 4 (ppd)
cA: [ 6.91627  1.73309  3.40612  6.32929  6.91627]
cD: [-1.99191 -2.156   -5.95035 -1.21545 -1.99191]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 3 (sp1)
cA: [-0.51764  1.73309  3.40612  6.32929  7.45001]
cD: [ 0.      -2.156   -5.95035 -1.21545  0.     ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 5 (per)
cA: [ 4.05317  3.05257  2.85381  8.42522]
cD: [ 0.18947  4.18258  4.33738  2.60428]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
```

The default mode is *sym*:

```
>>> 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='sym')
>>> 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 *sym*) for the border effect handling:

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

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

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

```
>>> # int() is for normalizing Python integers and long integers for documentation tests
>>> int(pywt.dwt_coeff_len(data_len=len(x), filter_len=w.dec_len, mode='sym'))
6
>>> int(pywt.dwt_coeff_len(len(x), w, 'sym'))
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
['zpd', 'cpd', 'sym', 'ppd', 'sp1', 'per']
```

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

As you see in the above example, the *per* (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='per')
>>> print pywt.idwt(cA, cD, 'sym3', 'sym') # invalid mode
[ 1.  1. -2.  5.]
>>> print pywt.idwt(cA, cD, 'sym3', 'per')
[ 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', 'sym')
[ 1.19006969  1.54362308  0.44828774 -0.25881905  0.48296291  0.8365163 ]
```

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

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

```
>>> print pywt.idwt([0, 0, 0, 0], [1, 2, 0, 1], 'db2', 'sym')
[ 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', 'sym')
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', 'sym')
Traceback (most recent call last):
...
ValueError: Coefficients arrays must have the same size.
```

But for some applications like multilevel DWT and IDWT it is sometimes convenient to allow for a small departure from this behaviour. When the *correct_size* flag is set, the approximation coefficients array can be larger from the details coefficient array by one element:

```
>>> print pywt.idwt([1, 2, 3, 4, 5], [1, 2, 3, 4], 'db2', 'sym', correct_size=True)
[ 1.76776695  0.61237244  3.18198052  0.61237244  4.59619408  0.61237244]
```

```
>>> print pywt.idwt([1, 2, 3, 4], [1, 2, 3, 4, 5], 'db2', 'sym', correct_size=True)
Traceback (most recent call last):
...
ValueError: Coefficients arrays must satisfy (0 <= len(cA) - len(cD) <= 1).
```

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 *sym* mode is `4`, not `3`:

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

```
>>> int(pywt.dwt_coeff_len(1, pywt.Wavelet('db4').dec_len, 'sym'))
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='cpd', 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
```

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='sym')
```

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='sym')
```

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='sym')
```

```
>>> 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
sym
```

Collecting nodes

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

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='sym')
```

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

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

```
>>> 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='sym')
```

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='sym')
```

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

```
>>> 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='sym')
```

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
sym
```

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):
...     print path,
...     if (i+1) % 4 == 0: print
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):
...     print path,
...     if (i+1) % 8 == 0: print
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='sym')
```

```
>>> 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):
...     print path,
...     if (i+1) % 8 == 0: print
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='sym')
```

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 = 'sym'
>>> 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

Microsoft Visual C++ 2008 (Microsoft Visual Studio 9.0) is the compiler that is suitable for building extensions for Python 2.6, 2.7, 3.0, 3.1 and 3.2 (both 32 and 64 bit).

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 2011 (11.0)

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

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

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

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

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

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

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

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

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

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

MinGW C/C++ compiler

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

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

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

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

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

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

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

Next steps

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

10.3.2 Preparing Linux build environment

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

Installing basic build tools

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

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

Next steps

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

10.3.3 Installing build dependencies

Setting up Python virtual environment

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

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

Installing Cython

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

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

Installing numpy

Use `pip` to install numpy:

```
pip install numpy
```

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

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

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

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

Installing Sphinx

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

```
pip install Sphinx
```

10.3.4 Building and installing PyWavelets

Installing from source code

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

```
git clone https://github.com/nigma/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/nigma/pywt.git#egg=PyWavelets
```

or:

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

Installing a regular release from PyPi

A regular release can be installed with pip or easy_install:

```
pip install PyWavelets
```

10.3.5 Testing

Continous integration with Travis-CI

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

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

Running tests locally

Tests are implemented with nose, so use one of:

$ nosetests pywt

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

Running tests with Tox

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

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

For more information see the Tox documentation.

10.3.6 Something not working?

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

10.4 Resources

10.4.1 Code

The GitHub repository is now the main code repository.

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

10.4.2 Questions and bug reports

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

10.4.3 Wavelet Properties Browser

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

10.4.4 Articles

- Denoising: wavelet thresholding
- Wavelet Regression in Python

10.5 PyWavelets

10.5.1 API Reference

Wavelets

Wavelet `families()`

`pywt.families()`

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`)

Example:

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

Built-in wavelets - `wavelist()`

`pywt.wavelist([family])`

The `wavelist()` function returns a list of names of the built-in wavelets.

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

Example:

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

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

`Wavelet` object

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

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

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

Parameters

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

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

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

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

Example:

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

name
Wavelet name.

short_name
Short wavelet name.

dec_lo
Decomposition filter values.

dec_hi
Decomposition filter values.

rec_lo
Reconstruction filter values.

rec_hi
Reconstruction filter values.

dec_len
Decomposition filter length.

rec_len
Reconstruction filter length.

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

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

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

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

short_family_name
Wavelet short family name

family_name
Wavelet family name

orthogonal
Set if wavelet is orthogonal

biorthogonal
Set if wavelet is biorthogonal

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

vanishing_moments_psi
Number of vanishing moments for the wavelet function

vanishing_moments_phi
Number of vanishing moments for the scaling function

Example:

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

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

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

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

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

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

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

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

Example:

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

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

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

Example:

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

See also:

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

Using custom wavelets

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

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

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

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

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

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

Example:

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

Signal extension modes

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

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

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

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

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

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

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

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

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

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

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

- `sp1` - **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:

- `per` - **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
['zpd', 'cpd', 'sym', 'ppd', 'sp1', 'per']
```

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', 'sp1')
>>> (a, d) = pywt.dwt([1,2,3,4,5,6], pywt.Wavelet('db2'), pywt.MODES.sp1)
```

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

Discrete Wavelet Transform (DWT)

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

Single level `dwt`

`pywt.`**`dwt`**(*data*, *wavelet*[, *mode='sym'*])

The `dwt()` function is used to perform single level, one dimensional Discrete Wavelet Transform.

```
(cA, cD) = dwt(data, wavelet, mode='sym')
```

Parameters

- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **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 to deal with the border distortion problem. See *MODES* for details.

The transform coefficients are returned as two arrays containing approximation (*cA*) and detail (*cD*) coefficients respectively. Length of returned arrays depends on the selected signal extension *mode* - 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:

•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)
```

Example:

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

Multilevel decomposition using `wavedec`

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

The `wavedec()` function performs 1D multilevel Discrete Wavelet Transform decomposition of given signal and returns ordered list of coefficients arrays in the form:

```
[cA_n, cD_n, cD_n-1, ..., cD2, cD1],
```

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.

Parameters

- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **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 to deal with the border distortion problem. See *MODES* for details.
- **level** – Number of decomposition steps to perform. If the level is `None`, then the full decomposition up to the level computed with `dwt_max_level()` function for the given data and wavelet lengths is performed.

Example:

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

Partial Discrete Wavelet Transform data decomposition `downcoef`

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

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

Parameters

- **part** – decomposition type. For `a` computes approximation coefficients, for `d` - details coefficients.
- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **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 to deal with the border distortion problem. See *MODES* for details.
- **level** – Number of decomposition steps to perform.

Maximum decomposition level - `dwt_max_level`

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

The `dwt_max_level()` function can be used to compute the maximum *useful* level of decomposition for the given *input data length* and *wavelet filter length*.

The returned value equals to:

```
floor( log(data_len/(filter_len-1)) / log(2) )
```

Although the maximum decomposition level can be quite high for long signals, usually smaller values are chosen depending on the application.

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

Example:

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

Result coefficients length - `dwt_coeff_len`

pywt.**dwt_coeff_len**(*data_len*, *filter_len*, *mode*)

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

For *periodization* mode this equals:

```
ceil(data_len / 2)
```

which is the lowest possible length guaranteeing perfect reconstruction.

For other *modes*:

```
floor((data_len + filter_len - 1) / 2)
```

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

Inverse Discrete Wavelet Transform (IDWT)

Single level `idwt`

pywt.**idwt**(*cA*, *cD*, *wavelet*[, *mode='sym'*[, *correct_size=0*]])

The `idwt()` function reconstructs data from the given coefficients by performing single level Inverse Discrete Wavelet Transform.

Parameters

- **cA** – Approximation coefficients.
- **cD** – Detail coefficients.
- **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 to deal with the border distortion problem. See *MODES* for details. This is only important when DWT was performed in *periodization* mode.
- **correct_size** – Typically, *cA* and *cD* coefficients lists must have equal lengths in order to perform IDWT. Setting *correct_size* to *True* allows *cA* to be greater in size by one element compared to the *cD* size. This option is very useful when doing multilevel decomposition and reconstruction (as for example with the `wavedec()` function) of non-dyadic length signals when such minor differences can occur at various levels of IDWT.

Example:

```
>>> import pywt
>>> (cA, cD) = pywt.dwt([1,2,3,4,5,6], 'db2', 'sp1')
```

```
>>> print pywt.idwt(cA, cD, 'db2', 'sp1')
[ 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', 'sp1')
>>> A = pywt.idwt(cA, None, 'db2', 'sp1')
>>> D = pywt.idwt(None, cD, 'db2', 'sp1')
>>> print A + D
[ 1.  2.  3.  4.  5.  6.]
```

Multilevel reconstruction using `waverec`

`pywt.`**`waverec`**(*coeffs*, *wavelet*[, *mode='sym'*])

Performs multilevel reconstruction of signal from the given list of coefficients.

Parameters

- **`coeffs`** – Coefficients list must be in the form like returned by `wavedec()` decomposition function, which is:

```
[cAn, cDn, cDn-1, ..., cD2, cD1]
```

- **`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 to deal with the border distortion problem. See *MODES* for details.

Example:

```
>>> import pywt
>>> coeffs = pywt.wavedec([1,2,3,4,5,6,7,8], 'db2', level=2)
>>> print pywt.waverec(coeffs, 'db2')
[ 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`** – Defines the input coefficients type:
  - **'a'** - approximations reconstruction is performed
  - **'d'** - details reconstruction is performed
- **`coeffs`** – Coefficients array to reconstruct.
- **`wavelet`** – Wavelet to use in the transform. This can be a name of the wavelet from the `wavelist()` list or a `Wavelet` object instance.
- **`level`** – If *level* value is specified then a multilevel reconstruction is performed (first reconstruction is of type specified by *part* and all the following ones with *part* type `a`)

- **take** – If *take* is specified then only the central part of length equal to the *take* parameter value is returned.

Example:

```
>>> import pywt
>>> data = [1,2,3,4,5,6]
>>> (cA, cD) = pywt.dwt(data, 'db2', 'sp1')
>>> print pywt.upcoef('a', cA, 'db2') + pywt.upcoef('d', cD, 'db2')
[-0.25        -0.4330127   1.          2.          3.          4.          5.
  6.          1.78589838 -1.03108891]
>>> n = len(data)
>>> print pywt.upcoef('a',cA,'db2',take=n) + pywt.upcoef('d',cD,'db2',take=n)
[ 1.  2.  3.  4.  5.  6.]
```

2D Forward and Inverse Discrete Wavelet Transform

Single level dwt2

pywt.**dwt2**(*data*, *wavelet*[, *mode='sym'*])

The *dwt2()* function performs single level 2D Discrete Wavelet Transform.

Parameters

- **data** – 2D input data.
- **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 to deal with the border distortion problem. See *MODES* for details. This is only important when DWT was performed in *periodization* mode.

Returns one average and three details 2D coefficients arrays. The coefficients arrays are organized in tuples in the following form:

```
(cA, (cH, cV, cD))
```

where *cA*, *cH*, *cV*, *cD* denote approximation, horizontal detail, vertical detail and diagonal detail coefficients respectively.

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.

Example:

```
>>> import pywt, numpy
>>> data = numpy.ones((4,4), dtype=numpy.float64)
>>> coeffs = pywt.dwt2(data, 'haar')
```

```
>>> cA, (cH, cV, cD) = coeffs
>>> print cA
[[ 2.  2.]
 [ 2.  2.]]
>>> print cV
[[ 0.  0.]
 [ 0.  0.]]
```

Single level idwt2

pywt.**idwt2**(*coeffs*, *wavelet*[, *mode='sym'*])

The *idwt2()* function reconstructs data from the given coefficients set by performing single level 2D Inverse Discrete Wavelet Transform.

Parameters

- **coeffs** – A tuple with approximation coefficients and three details coefficients 2D arrays like from *dwt2()*:

```
(cA, (cH, cV, cD))
```

- **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 to deal with the border distortion problem. See *MODES* for details. This is only important when the *dwt()* was performed in the *periodization* mode.

Example:

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

2D multilevel decomposition using wavedec2

pywt.**wavedec2**(*data*, *wavelet*[, *mode='sym'*[, *level=None*]])

Performs multilevel 2D Discrete Wavelet Transform decomposition and returns coefficients list:

```
[cAn, (cHn, cVn, cDn), ..., (cH1, cV1, cD1)]
```

where *n* denotes the level of decomposition and *cA*, *cH*, *cV* and *cD* are approximation, horizontal detail, vertical detail and diagonal detail coefficients arrays respectively.

Parameters

- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **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 to deal with the border distortion problem. See *MODES* for details.
- **level** – Decomposition level. This should not be greater than the reasonable maximum value computed with the `dwt_max_level()` function for the smaller dimension of the input data.

Example:

```
>>> import pywt, numpy
>>> coeffs = pywt.wavedec2(numpy.ones((8,8)), 'db1', level=2)
>>> cA2, (cH2, cV2, cD2), (cH1, cV1, cD1) = coeffs
>>> print cA2
[[ 4.  4.]
 [ 4.  4.]]
```

2D multilevel reconstruction using `waverec2`

`pywt.`**`waverec2`**(*coeffs*, *wavelet*[, *mode='sym'*])

Performs multilevel reconstruction from the given coefficients set.

Parameters

- **coeffs** – Coefficients set must be in the form like that from `wavedec2()` decomposition:

```
[cAn, (cHn, cVn, cDn), ..., (cH1, cV1, cD1)]
```

- **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 to deal with the border distortion problem. See *MODES* for details.

Example:

```
>>> import pywt, numpy
>>> coeffs = pywt.wavedec2(numpy.ones((4,4)), 'db1')
>>> print "levels:", len(coeffs)-1
levels: 2
>>> print pywt.waverec2(coeffs, 'db1')
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

Stationary Wavelet Transform

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

Multilevel `swt`

`pywt.`**`swt`**(*data*, *wavelet*, *level*[, *start_level=0*])

Performs multilevel Stationary Wavelet Transform.

Parameters

- **data** – Input signal can be NumPy array, Python list or other iterable object. Both *single* and *double* precision floating-point data types are supported and the output type depends on the input type. If the input data is not in one of these types it will be converted to the default *double* precision data format before performing computations.
- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the `wavelist()` list or a `Wavelet` object instance.
- **level** (*int*) – Required transform level. See the `swt_max_level()` function.
- **start_level** (*int*) – The level at which the decomposition will begin (it allows to skip a given number of transform steps and compute coefficients starting directly from the *start_level*)

Returns list of coefficient pairs in the form:

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

where *n* is the *level* value.

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

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

Multilevel swt2

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

Performs multilevel 2D Stationary Wavelet Transform.

Parameters

- **data** – 2D array with input data.
- **wavelet** – Wavelet to use in the transform. This can be a name of the wavelet from the `wavelist()` list or a `Wavelet` object instance.
- **level** – Number of decomposition steps to perform.
- **start_level** – The level at which the decomposition will begin.

The result is a set of coefficients arrays over the range of decomposition levels:

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

where *cA* is approximation, *cH* is horizontal details, *cV* is vertical details, *cD* is diagonal details, *n* is *start_level* and *m* equals *n+level*.

Maximum decomposition level - `swt_max_level`

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

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

Parameters **input_len** – Input data length.

Wavelet Packets

New in version 0.2.

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

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

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

The below diagram illustrates the inheritance tree:

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

BaseNode - a common interface of WaveletPacket and WaveletPacket2D

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

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

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

Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Parameters

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

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

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

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

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

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

Parameters

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

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

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

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

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

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

Parameters

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

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

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

Args arguments to pass to the *func*

Kwargs keyword arguments to pass to the *func*

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

Parameters

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

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

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

Args arguments to pass to the *func*

Kwargs keyword arguments to pass to the *func*

WaveletPacket and WaveletPacket tree Node

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

node_name

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

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

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

decompose()

See also:

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

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

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

Parameters

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

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

Collects nodes from the given level of decomposition.

Parameters

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

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

Thresholding functions

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

Hard thresholding

hard(*data*, *value*[, *substitute=0*])

Hard thresholding. Replace all *data* values with *substitute* where their absolute value is less than the *value* param.

Data values with absolute value greater or equal to the thresholding *value* stay untouched.

Parameters

- **data** – numeric data
- **value** – thresholding value
- **substitute** – substitute value

Returns array

Soft thresholding

soft (*data*, *value*[, *substitute=0*])
Soft thresholding.

Parameters

- **data** – numeric data
- **value** – thresholding value
- **substitute** – substitute value

Returns array

Greater

greater (*data*, *value*[, *substitute=0*])
Replace *data* with *substitute* where *data* is below the thresholding *value*.

Greater data values pass untouched.

Parameters

- **data** – numeric data
- **value** – thresholding value
- **substitute** – substitute value

Returns array

Less

less (*data*, *value*[, *substitute=0*])
Replace *data* with *substitute* where *data* is above the thresholding *value*.

Less data values pass untouched.

Parameters

- **data** – numeric data
- **value** – thresholding value
- **substitute** – substitute value

Returns array

Other functions

Single-level n-dimensional Discrete Wavelet Transform.

pywt.**dwtn** (*data*, *wavelet*[, *mode='sym'*])
Performs single-level n-dimensional Discrete Wavelet Transform.

Parameters

- **data** – n-dimensional 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 to deal with the border distortion problem. See *MODES* for details.

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

Integrating wavelet functions - `intwave()`

`pywt.`**`intwave`**(*wavelet*[, *precision=8*])

Integration of wavelet function approximations as well as any other signals can be performed using the `pywt.intwave()` function.

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

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

```
[int_psi, x] = intwave(wavelet, precision)
```

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

```
[int_psi_d, int_psi_r, x] = intwave(wavelet, precision)
```

- for a tuple of coefficients data and a x-grid - an integral of function and the given x-grid is returned (the x-grid is used for computations).:

```
[int_function, x] = intwave((data, x), precision)
```

Example:

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

Central frequency of *psi* wavelet function

`pywt.`**`centfrq`**(*wavelet*[, *precision=8*])
`pywt.`**`centfrq`**((*function_approx*, *x*))

Parameters

- **wavelet** – `Wavelet`, wavelet name string or *(wavelet function approx., x grid)* pair

- **precision** – Precision that will be used for wavelet function approximation computed with the *Wavelet.wavefun()* method.

10.5.2 Usage examples

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

The Wavelet object

Wavelet families and builtin Wavelets names

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

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

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

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

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

```
>>> for family in pywt.families():
...     print "%s family:" % family, ', '.join(pywt.wavelist(family))
haar family: haar
db family: db1, db2, db3, db4, db5, db6, db7, db8, db9, db10, db11, db12, db13, db14, db15, db16, db1
sym family: sym2, sym3, sym4, sym5, sym6, sym7, sym8, sym9, sym10, sym11, sym12, sym13, sym14, sym15,
coif family: coif1, coif2, coif3, coif4, coif5
bior family: bior1.1, bior1.3, bior1.5, bior2.2, bior2.4, bior2.6, bior2.8, bior3.1, bior3.3, bior3.5
rbio family: rbio1.1, rbio1.3, rbio1.5, rbio2.2, rbio2.4, rbio2.6, rbio2.8, rbio3.1, rbio3.3, rbio3.5
dmey family: dmey
```

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

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

Creating Wavelet objects

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

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

So.. that's it.

Wavelet properties

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

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

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

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

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

```
>>> print_array(w.dec_lo)
[0.03522629188210, -0.08544127388224, -0.13501102001039, 0.45987750211933, 0.80689150931334, 0.332670
>>> print_array(w.dec_hi)
[-0.33267055295096, 0.80689150931334, -0.45987750211933, -0.13501102001039, 0.08544127388224, 0.03522
>>> print_array(w.rec_lo)
[0.33267055295096, 0.80689150931334, 0.45987750211933, -0.13501102001039, -0.08544127388224, 0.035226
>>> print_array(w.rec_hi)
[0.03522629188210, 0.08544127388224, -0.13501102001039, -0.45987750211933, 0.80689150931334, -0.33267
```

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

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

Other Wavelet's properties are:

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

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

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

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

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

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

- Symmetry (*symmetry*):

```
>>> print w.symmetry
asymmetric
```

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

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

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

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

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

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

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

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

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

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

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

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

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

And now... the *wavefun*!

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

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

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

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

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

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

See also:

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

Signal Extension Modes

Import `pywt` first

```
>>> import pywt
```

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

List of available signal extension *modes*:

```
>>> print pywt.MODES.modes
['zpd', 'cpd', 'sym', 'ppd', 'sp1', 'per']
```

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

```
>>> x = [1,2,1,5,-1,8,4,6]
>>> for mode in pywt.MODES.modes:
...     cA, cD = pywt.dwt(x, 'db2', mode)
...     print "Mode:", mode
...     print "cA:", format_array(cA)
...     print "cD:", format_array(cD)
...     print "Reconstruction:", pywt.idwt(cA, cD, 'db2', mode)
Mode: zpd
cA: [-0.03468  1.73309  3.40612  6.32929  6.95095]
cD: [-0.12941 -2.156   -5.95035 -1.21545 -1.8625 ]
```

```
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: cpd
cA: [ 1.2848   1.73309  3.40612  6.32929  7.51936]
cD: [-0.48296 -2.156   -5.95035 -1.21545  0.25882]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: sym
cA: [ 1.76777  1.73309  3.40612  6.32929  7.77817]
cD: [-0.61237 -2.156   -5.95035 -1.21545  1.22474]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: ppd
cA: [ 6.91627  1.73309  3.40612  6.32929  6.91627]
cD: [-1.99191 -2.156   -5.95035 -1.21545 -1.99191]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: sp1
cA: [-0.51764  1.73309  3.40612  6.32929  7.45001]
cD: [ 0.      -2.156   -5.95035 -1.21545  0.     ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: per
cA: [ 4.05317  3.05257  2.85381  8.42522]
cD: [ 0.18947  4.18258  4.33738  2.60428]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
```

Invalid mode name should rise a `ValueError`:

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

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

```
>>> for mode_name in ['zpd', 'cpd', 'sym', 'ppd', 'sp1', 'per']:
...     mode = getattr(pywt.MODES, mode_name)
...     cA, cD = pywt.dwt([1,2,1,5,-1,8,4,6], 'db2', mode)
...     print "Mode:", mode, "(%s)" % mode_name
...     print "cA:", format_array(cA)
...     print "cD:", format_array(cD)
...     print "Reconstruction:", pywt.idwt(cA, cD, 'db2', mode)
Mode: 0 (zpd)
cA: [-0.03468  1.73309  3.40612  6.32929  6.95095]
cD: [-0.12941 -2.156   -5.95035 -1.21545 -1.8625 ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 2 (cpd)
cA: [ 1.2848   1.73309  3.40612  6.32929  7.51936]
cD: [-0.48296 -2.156   -5.95035 -1.21545  0.25882]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 1 (sym)
cA: [ 1.76777  1.73309  3.40612  6.32929  7.77817]
cD: [-0.61237 -2.156   -5.95035 -1.21545  1.22474]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 4 (ppd)
cA: [ 6.91627  1.73309  3.40612  6.32929  6.91627]
cD: [-1.99191 -2.156   -5.95035 -1.21545 -1.99191]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 3 (sp1)
cA: [-0.51764  1.73309  3.40612  6.32929  7.45001]
cD: [ 0.      -2.156   -5.95035 -1.21545  0.     ]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
Mode: 5 (per)
```

```
cA: [ 4.05317  3.05257  2.85381  8.42522]
cD: [ 0.18947  4.18258  4.33738  2.60428]
Reconstruction: [ 1.  2.  1.  5. -1.  8.  4.  6.]
```

The default mode is *sym*:

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

DWT and IDWT

Discrete Wavelet Transform

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

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

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

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

Inverse Discrete Wavelet Transform

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

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

Voilà! That's it!

More Examples

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

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

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

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

```
>>> # int() is for normalizing Python integers and long integers for documentation tests
>>> int(pywt.dwt_coeff_len(data_len=len(x), filter_len=w.dec_len, mode='sym'))
6
>>> int(pywt.dwt_coeff_len(len(x), w, 'sym'))
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
['zpd', 'cpd', 'sym', 'ppd', 'sp1', 'per']
```

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

As you see in the above example, the *per* (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='per')
>>> print pywt.idwt(cA, cD, 'sym3', 'sym') # invalid mode
[ 1.  1. -2.  5.]
>>> print pywt.idwt(cA, cD, 'sym3', 'per')
[ 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', 'sym')
[ 1.19006969  1.54362308  0.44828774 -0.25881905  0.48296291  0.8365163 ]
```

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

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

```
>>> print pywt.idwt([0, 0, 0, 0], [1, 2, 0, 1], 'db2', 'sym')
[ 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', 'sym')
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', 'sym')
Traceback (most recent call last):
...
ValueError: Coefficients arrays must have the same size.
```

But for some applications like multilevel DWT and IDWT it is sometimes convenient to allow for a small departure from this behaviour. When the *correct_size* flag is set, the approximation coefficients array can be larger from the details coefficient array by one element:

```
>>> print pywt.idwt([1, 2, 3, 4, 5], [1, 2, 3, 4], 'db2', 'sym', correct_size=True)
[ 1.76776695  0.61237244  3.18198052  0.61237244  4.59619408  0.61237244]
```

```
>>> print pywt.idwt([1, 2, 3, 4], [1, 2, 3, 4, 5], 'db2', 'sym', correct_size=True)
Traceback (most recent call last):
...
ValueError: Coefficients arrays must satisfy (0 <= len(cA) - len(cD) <= 1).
```

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 *sym* mode is `4`, not `3`:

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

```
>>> int(pywt.dwt_coeff_len(1, pywt.Wavelet('db4').dec_len, 'sym'))
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='cpd', 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
```

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='sym')
```

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='sym')
```

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='sym')
```

```
>>> 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
sym
```

Collecting nodes

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

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='sym')
```

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

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

```
>>> 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='sym')
```

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='sym')
```

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

```
>>> print wp.a
None
```

Remember that you should not rely on the attribute access.

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

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

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

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

And so is *wp.d*:

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

2D Wavelet Packets

Import pywt

```
>>> 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='sym')
```

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:

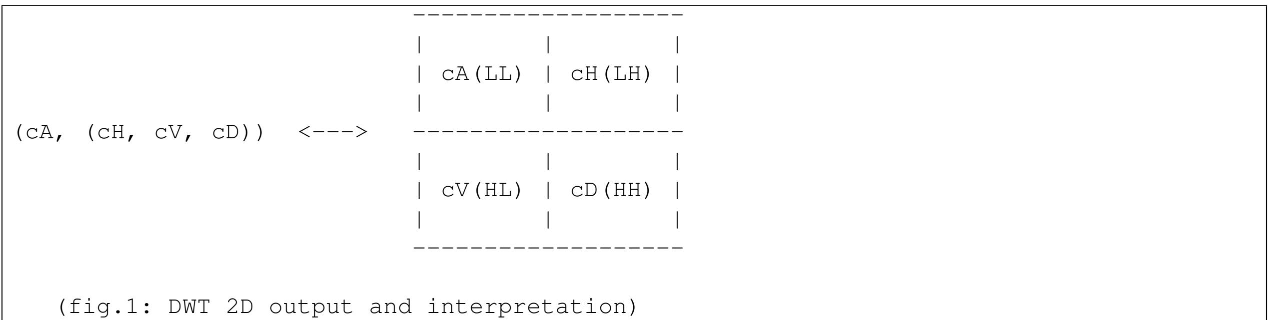
Wavelet Packet *nodes* are arranged in a tree. Each node in a WP tree is uniquely identified and addressed by a `path` string.

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
sym
```

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):
...     print path,
...     if (i+1) % 4 == 0: print
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):
...     print path,
...     if (i+1) % 8 == 0: print
```

```
    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='sym')
```

```
>>> 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):
```

```
...     print path,
...     if (i+1) % 8 == 0: print
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='sym')
```

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

   ```
   >>> print wp.a
   None
   ```

   Remember that you should not rely on the attribute access.

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

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

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

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

   And so is *wp.d*:

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

Gotchas

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

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

10.5.3 Development notes

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

Preparing Windows build environment

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

Installing Windows SDK C/C++ compiler

Microsoft Visual C++ 2008 (Microsoft Visual Studio 9.0) is the compiler that is suitable for building extensions for Python 2.6, 2.7, 3.0, 3.1 and 3.2 (both 32 and 64 bit).

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 2011 (11.0)

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

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

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

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

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

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

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

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

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

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

MinGW C/C++ compiler

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

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

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

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

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

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

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

Next steps

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

Preparing Linux build environment

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

Installing basic build tools

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

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

Next steps

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

Installing build dependencies

Setting up Python virtual environment

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

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

Installing Cython

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

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

Installing numpy

Use `pip` to install numpy:

```
pip install numpy
```

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

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

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

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

Installing Sphinx

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

```
pip install Sphinx
```

Building and installing PyWavelets

Installing from source code

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

```
git clone https://github.com/nigma/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/nigma/pywt.git#egg=PyWavelets
```

or:

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

Installing a regular release from PyPi

A regular release can be installed with pip or easy_install:

```
pip install PyWavelets
```

Testing

Continous integration with Travis-CI

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

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

Running tests locally

Tests are implemented with nose, so use one of:

> $ nosetests pywt

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

Running tests with Tox

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

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

For more information see the Tox documentation.

Something not working?

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

10.5.4 Resources

Code

The GitHub repository is now the main code repository.

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

Questions and bug reports

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

Wavelet Properties Browser

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

Articles

- Denoising: wavelet thresholding
- Wavelet Regression in Python

10.5.5 Release Notes

PyWavelets 0.3.0 Release Notes

Contents

- *PyWavelets 0.3.0 Release Notes*
  - *New features*
    - *Test suite*
    - *n-D Inverse Discrete Wavelet Transform*
    - *Thresholding*
  - *Backwards incompatible changes*
  - *Other changes*
  - *Authors*
    - *Issues closed for v0.3.0*
    - *Pull requests for v0.3.0*

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...

10.6 Indices and tables

- genindex
- search

Symbols

__delitem__() (pywt.BaseNode method), 36, 81
__getitem__() (pywt.BaseNode method), 35, 80
__init__() (pywt.BaseNode method), 34, 79
__init__() (pywt.WaveletPacket method), 37, 82
__init__() (pywt.WaveletPacket2D method), 38, 83
__setitem__() (pywt.BaseNode method), 36, 80

B

BaseNode (class in pywt), 34, 79
biorthogonal (pywt.Wavelet attribute), 23, 68

C

centfrq() (in module pywt), 41, 85

D

data (pywt.BaseNode attribute), 35, 79
dec_hi (pywt.Wavelet attribute), 22, 67
dec_len (pywt.Wavelet attribute), 22, 67
dec_lo (pywt.Wavelet attribute), 22, 67
decompose() (pywt.BaseNode method), 35, 80
decompose() (pywt.Node method), 37, 82
decompose() (pywt.Node2D method), 38, 83
downcoef() (in module pywt), 27, 72
dwt() (in module pywt), 26, 70
dwt2() (in module pywt), 30, 75
dwt_coeff_len() (in module pywt), 28, 73
dwt_max_level() (in module pywt), 27, 72
dwtn() (in module pywt), 40, 84

F

families() (in module pywt), 21, 66
family_name (pywt.Wavelet attribute), 23, 67
filter_bank (pywt.Wavelet attribute), 22, 67

G

get_leaf_nodes() (pywt.BaseNode method), 36, 81
get_level() (pywt.WaveletPacket method), 37, 82
get_level() (pywt.WaveletPacket2D method), 38, 83
get_subnode() (pywt.BaseNode method), 35, 80
greater() (built-in function), 39, 84

H

hard() (built-in function), 38, 83
has_any_subnode (pywt.BaseNode attribute), 35, 80

I

idwt() (in module pywt), 28, 73
idwt2() (in module pywt), 31, 76
intwave() (in module pywt), 40, 85
inverse_filter_bank (pywt.Wavelet attribute), 22, 67
is_empty (pywt.BaseNode attribute), 35, 80

L

less() (built-in function), 39, 84
level (pywt.BaseNode attribute), 35, 80

M

maxlevel (pywt.BaseNode attribute), 35, 80
mode (pywt.BaseNode attribute), 35, 80

N

name (pywt.Wavelet attribute), 22, 67
Node (class in pywt), 34, 37, 79, 81
Node2D (class in pywt), 34, 37, 79, 82
node_name (pywt.BaseNode attribute), 35, 80
node_name (pywt.Node attribute), 37, 81
node_name (pywt.Node2D attribute), 37, 82

O

orthogonal (pywt.Wavelet attribute), 23, 68

P

parent (pywt.BaseNode attribute), 35, 80
path (pywt.BaseNode attribute), 35, 80

R

rec_hi (pywt.Wavelet attribute), 22, 67
rec_len (pywt.Wavelet attribute), 22, 67

rec_lo (pywt.Wavelet attribute), 22, 67
reconstruct() (pywt.BaseNode method), 35, 80

S

short_family_name (pywt.Wavelet attribute), 23, 67
short_name (pywt.Wavelet attribute), 22, 67
soft() (built-in function), 39, 84
swt() (in module pywt), 33, 77
swt2() (in module pywt), 33, 78
swt_max_level() (in module pywt), 34, 79
symmetry (pywt.Wavelet attribute), 23, 68

U

upcoef() (in module pywt), 29, 74

V

vanishing_moments_phi (pywt.Wavelet attribute), 23, 68
vanishing_moments_psi (pywt.Wavelet attribute), 23, 68

W

walk() (pywt.BaseNode method), 36, 81
walk_depth() (pywt.BaseNode method), 36, 81
wavedec() (in module pywt), 26, 71
wavedec2() (in module pywt), 31, 76
wavefun() (pywt.Wavelet method), 23, 68
Wavelet (class in pywt), 22, 67
wavelet (pywt.BaseNode attribute), 35, 80
WaveletPacket (class in pywt), 34, 37, 79, 81, 82
WaveletPacket2D (class in pywt), 34, 37, 38, 79, 82, 83
wavelist() (in module pywt), 21, 66
waverec() (in module pywt), 29, 74
waverec2() (in module pywt), 32, 77