dtcwt Documentation

Release 0.6

Rich Wareham, Nick Kingsbury, Cian Shaffrey

November 27, 2013

Contents

This library provides support for computing 1D, 2D and 3D dual-tree complex wavelet transforms and their inverse in Python. The interface is simple and easy to use. As a quick example, a 1D DT-CWT can be performed from the Python console in a single line:

```
>>> import dtcwt
>>> Yl, Yh = dtcwt.dtwavexfm([1,2,3,4], nlevels=3) # 3 levels, default wavelets
```

The interface is intentionally similar to the existing MATLAB dual-tree complex wavelet transform toolbox provided by Prof. Nick Kingsbury. This library is intended to ease the porting of algorithms written using the original MATLAB toolbox to Python.

CHAPTER 1

Features of note

The features of the `dtcwt` library are:

- 1D, 2D and 3D forward and inverse Dual-tree Complex Wavelet Transform implementations.
- API similarity with the DTCWT MATLAB toolbox.
- Automatic selection of single versus double precision calculation.
- Built-in support for the most common complex wavelet families.

CHAPTER 2

Contents

2.1 Getting Started

This section will guide you through using the `dtcwt` library. Once installed, you are most likely to use one of these functions:

- `dtcwt.dtwavexfm()` – 1D DT-CWT transform.
- `dtcwt.dtwaveifm()` – Inverse 1D DT-CWT transform.
- `dtcwt.dtwavexfm2()` – 2D DT-CWT transform.
- `dtcwt.dtwaveifm2()` – Inverse 2D DT-CWT transform.
- `dtcwt.dtwavexfm3()` – 3D DT-CWT transform.
- `dtcwt.dtwaveifm3()` – Inverse 3D DT-CWT transform.

See *API Reference* for full details on how to call these functions. We shall present some simple usage below.

2.1.1 Installation

The easiest way to install `dtcwt` is via `easy_install` or `pip`:

```
$ pip install dtcwt
```

If you want to check out the latest in-development version, look at the project's GitHub page. Once checked out, installation is based on setuptools and follows the usual conventions for a Python project:

```
$ python setup.py install
```

(Although the *develop* command may be more useful if you intend to perform any significant modification to the library.) A test suite is provided so that you may verify the code works on your system:

```
$ python setup.py nosetests
```

This will also write test-coverage information to the `cover/` directory.

Building the documentation

There is a pre-built version of this documentation available online and you can build your own copy via the Sphinx documentation system:

```
$ python setup.py build_sphinx
```

Compiled documentation may be found in `build/docs/html/`.

2.1.2 1D transform

This example generates two 1D random walks and demonstrates reconstructing them using the forward and inverse 1D transforms. Note that `dtcwt.dtwavexfm()` and `dtcwt.dtwaveifm()` will transform columns of an input array independently:

```
import numpy as np
from matplotlib.pyplot import *

# Generate a 300x2 array of a random walk
vecs = np.cumsum(np.random.rand(300,2) - 0.5, 0)

# Show input
figure(1)
plot(vecs)
title('Input')

import dtcwt

# 1D transform
Yl, Yh = dtcwt.dtwavexfm(vecs)

# Inverse
vecs_recon = dtcwt.dtwaveifm(Yl, Yh)

# Show output
figure(2)
plot(vecs_recon)
title('Output')

# Show error
figure(3)
plot(vecs_recon - vecs)
title('Reconstruction error')

print('Maximum reconstruction error: {0}'.format(np.max(np.abs(vecs - vecs_recon))))

show()
```

2.1.3 2D transform

Using the pylab environment (part of matplotlib) we can perform a simple example where we transform the standard 'Lena' image and show the level 2 wavelet coefficients:

```
# Load the Lena image from the Internet into a StringIO object
from StringIO import StringIO
from urllib2 import urlopen
```

```
LENA_URL = 'http://www.ece.rice.edu/~wakin/images/lena512.pgm'
lena_file = StringIO(urlopen(LENA_URL).read())

# Parse the lena file and rescale to be in the range (0,1]
from scipy.misc import imread
lena = imread(lena_file) / 255.0

from matplotlib.pyplot import *
import numpy as np

# Show lena on the left
figure(1)
imshow(lena, cmap=cm.gray, clim=(0,1))

import dtcwt

# Compute two levels of dtcwt with the defaul wavelet family
Yh, Yl = dtcwt.dtwavexfm2(lena, 2)

# Show the absolute images for each direction in level 2.
# Note that the 2nd level has index 1 since the 1st has index 0.
figure(2)
for slice_idx in xrange(Yl[1].shape[2]):
    subplot(2, 3, slice_idx)
    imshow(np.abs(Yl[1][:,:,slice_idx]), cmap=cm.spectral, clim=(0, 1))

# Show the phase images for each direction in level 2.
figure(3)
for slice_idx in xrange(Yl[1].shape[2]):
    subplot(2, 3, slice_idx)
    imshow(np.angle(Yl[1][:,:,slice_idx]), cmap=cm.hsv, clim=(-np.pi, np.pi))

show()
```

If the library is correctly installed and you also have matplotlib installed, you should see these three figures:

2.1.4 3D transform

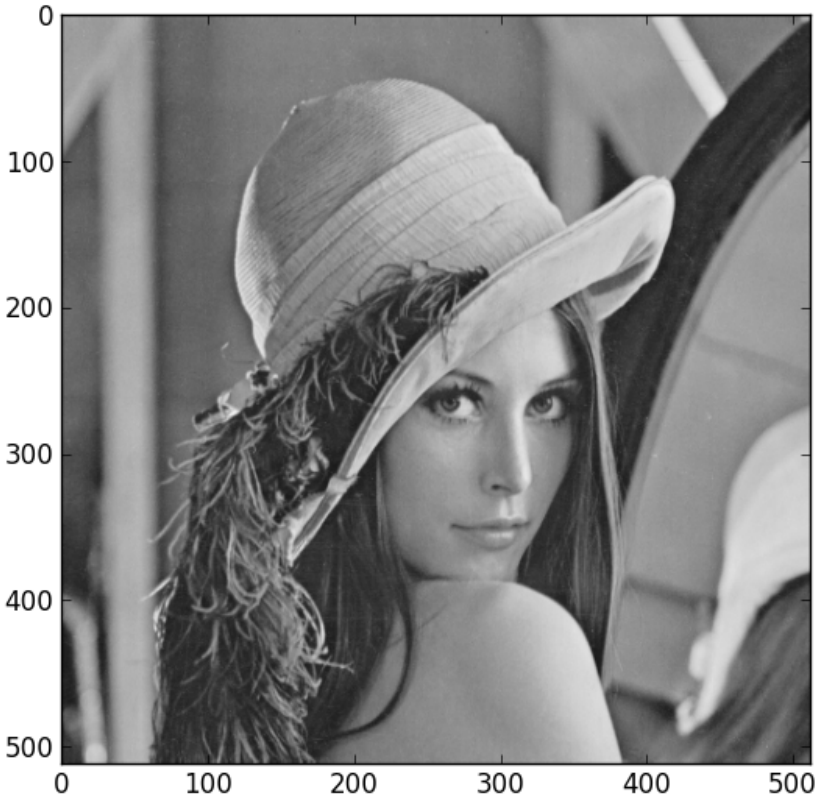
In the examples below I assume you've imported pyplot and numpy and, of course, the `dtcwt` library itself:

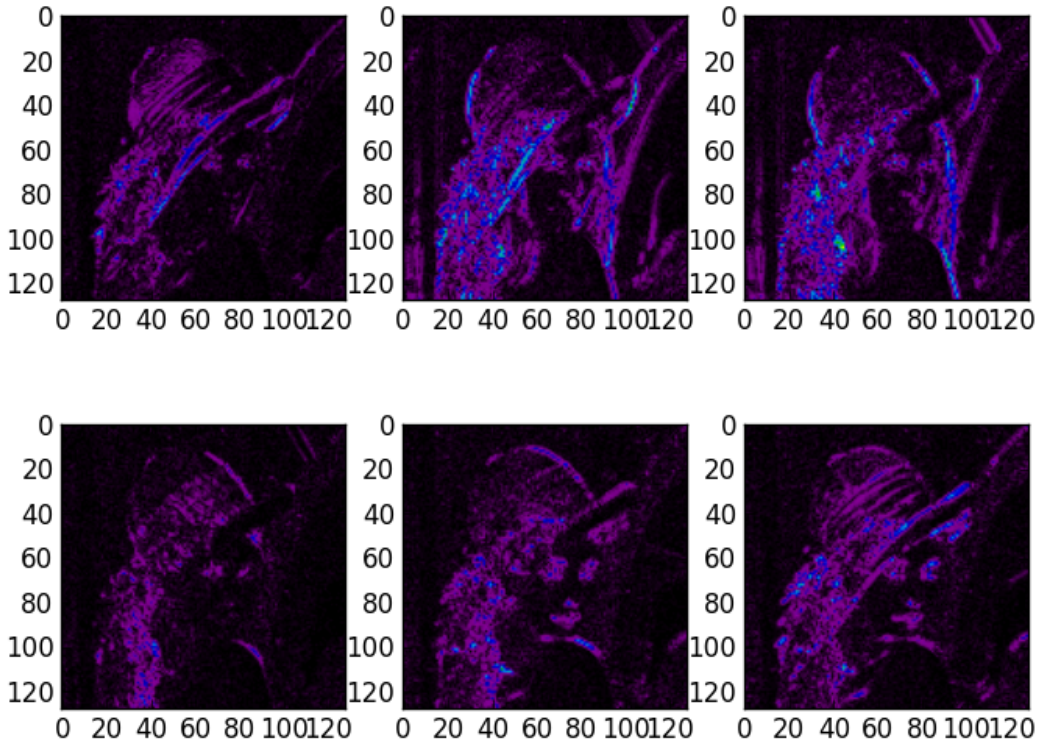
```
import numpy as np
from matplotlib.pyplot import *
from dtcwt import *
```

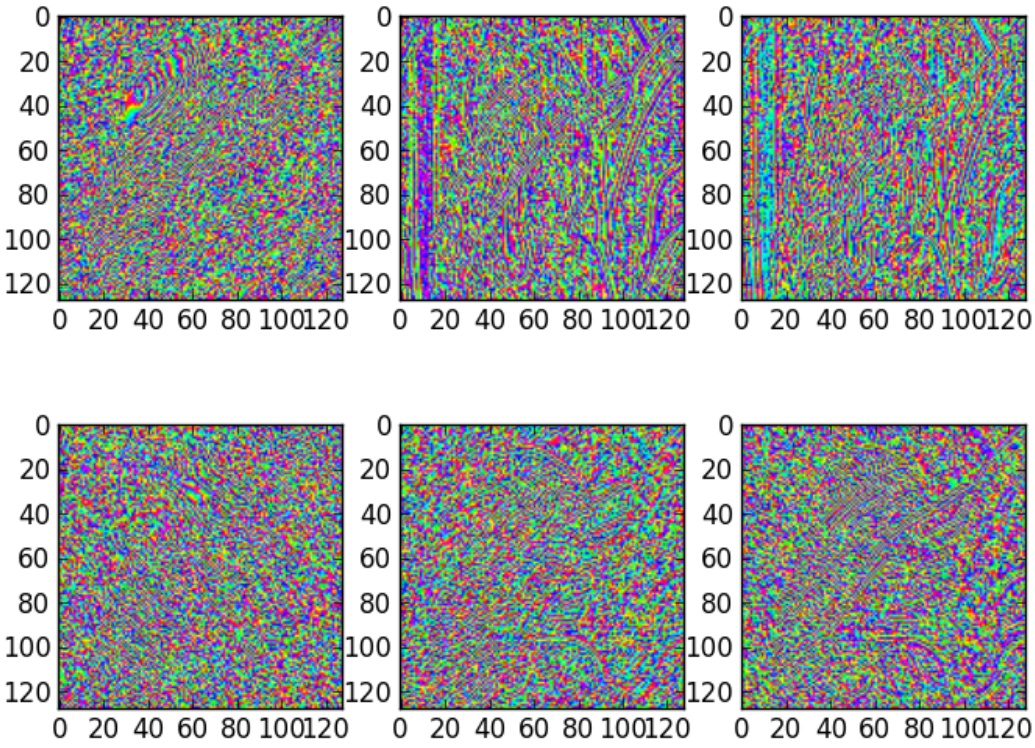
We can demonstrate the 3D transform by generating a 64x64x64 array which contains the image of a sphere:

```
GRID_SIZE = 64
SPHERE_RAD = int(0.45 * GRID_SIZE) + 0.5

grid = np.arange(-(GRID_SIZE>>1), GRID_SIZE>>1)
X, Y, Z = np.meshgrid(grid, grid, grid)
r = np.sqrt(X*X + Y*Y + Z*Z)

sphere = 0.5 + 0.5 * np.clip(SPHERE_RAD-r, -1, 1)
```

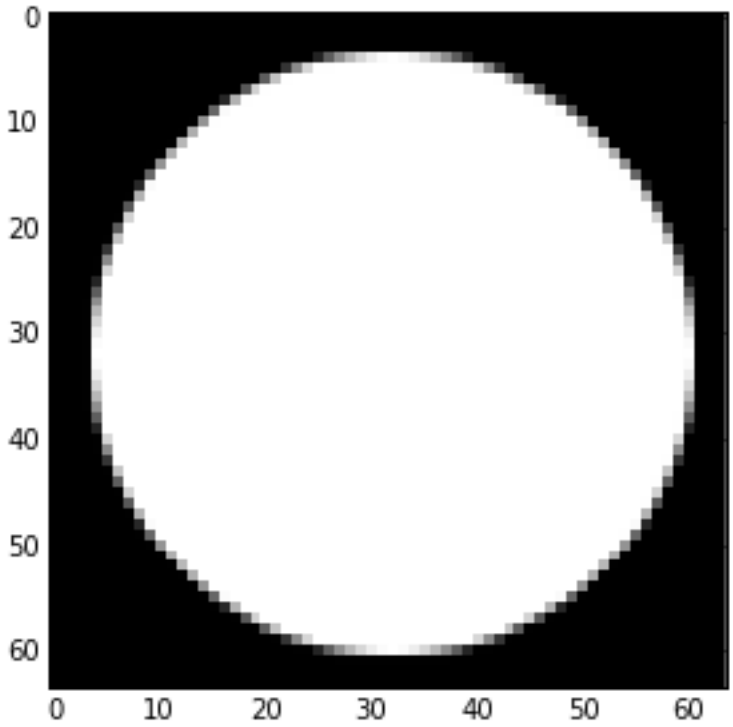
If we look at the central slice of this image, it looks like a circle:


0
100
200
300
400
500
0
100
200
300
400
500


0
20
40
60
80
100
120
0 20 40 60 80 100120


0
20
40
60
80
100
120
0 20 40 60 80 100120

```
imshow(sphere[:,:,GRID_SIZE>>1], interpolation='none', cmap=cm.gray)
```



Performing the 3 level DT-CWT with the defaul wavelet selection is easy:

```
Yl, Yh = dtwavexfm3(sphere, 3)
```

The function returns the lowest level low pass image and a tuple of complex subband coefficients:

```
>>> print(Yl.shape)
(16, 16, 16)
>>> for subbands in Yh:
...     print(subbands.shape)
(32, 32, 32, 28)
(16, 16, 16, 28)
(8, 8, 8, 28)
```

Performing the inverse transform should result in perfect reconstruction:

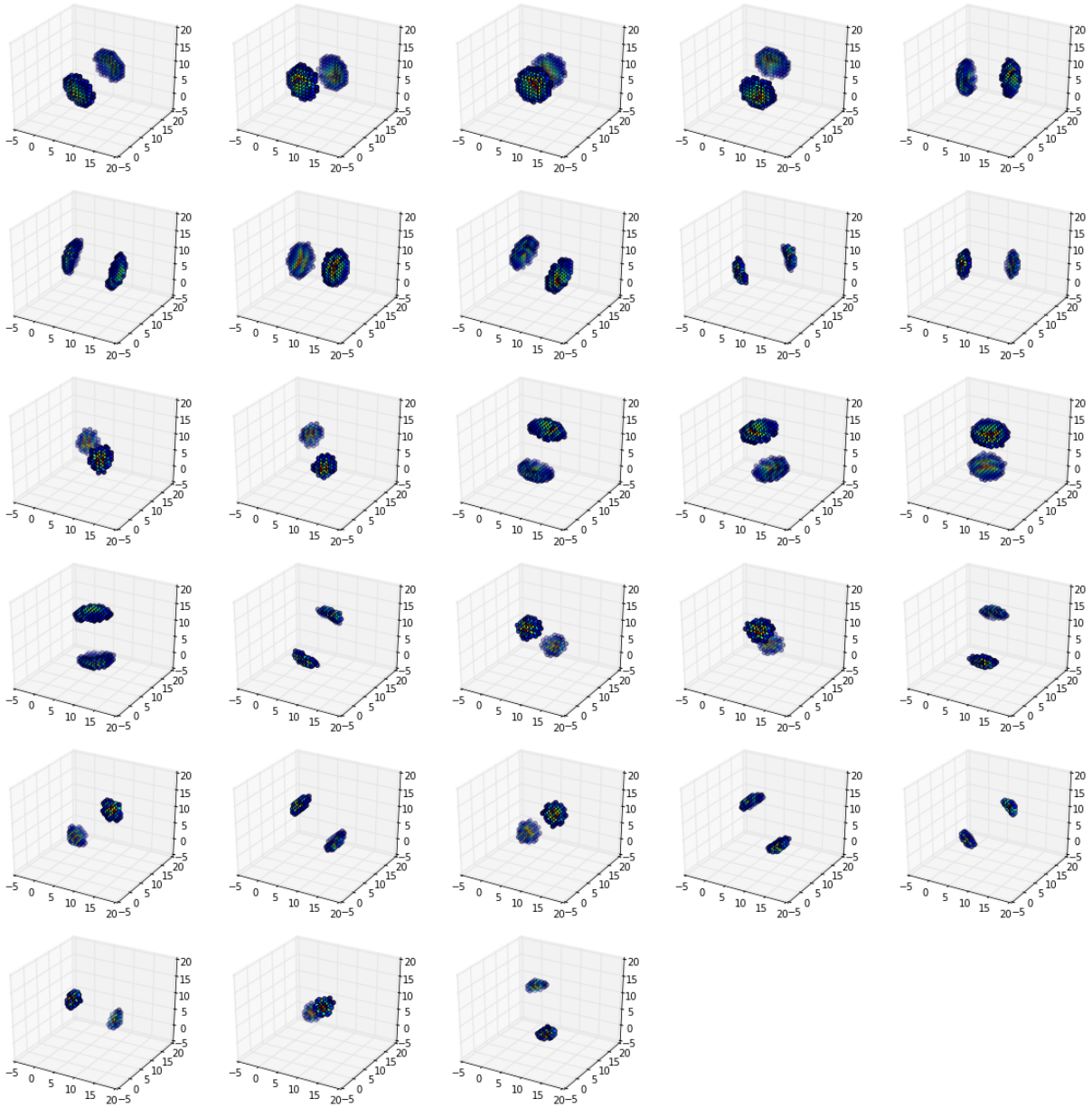
```
>>> Z = dtwaveifm3(Yl, Yh)
>>> print(np.abs(Z - ellipsoid).max()) # Should be < 1e-12
8.881784197e-15
```

If you plot the locations of the large complex coefficients, you can see the directional sensitivity of the transform:

```
from mpl_toolkits.mplot3d import Axes3D

figure(figsize=(16,16))
nplts = Yh[-1].shape[3]
nrows = np.ceil(np.sqrt(nplts))
ncols = np.ceil(nplts / nrows)
W = np.max(Yh[-1].shape[:3])
for idx in xrange(Yh[-1].shape[3]):
```

```
    C = np.abs(Yh[-1][:,:,:,idx])
    ax = gcf().add_subplot(nrows, ncols, idx+1, projection='3d')
    ax.set_aspect('equal')
    good = C > 0.2*C.max()
    x,y,z = np.nonzero(good)
    ax.scatter(x, y, z, c=C[good].ravel())
    ax.auto_scale_xyz((0,W), (0,W), (0,W))

tight_layout()
```

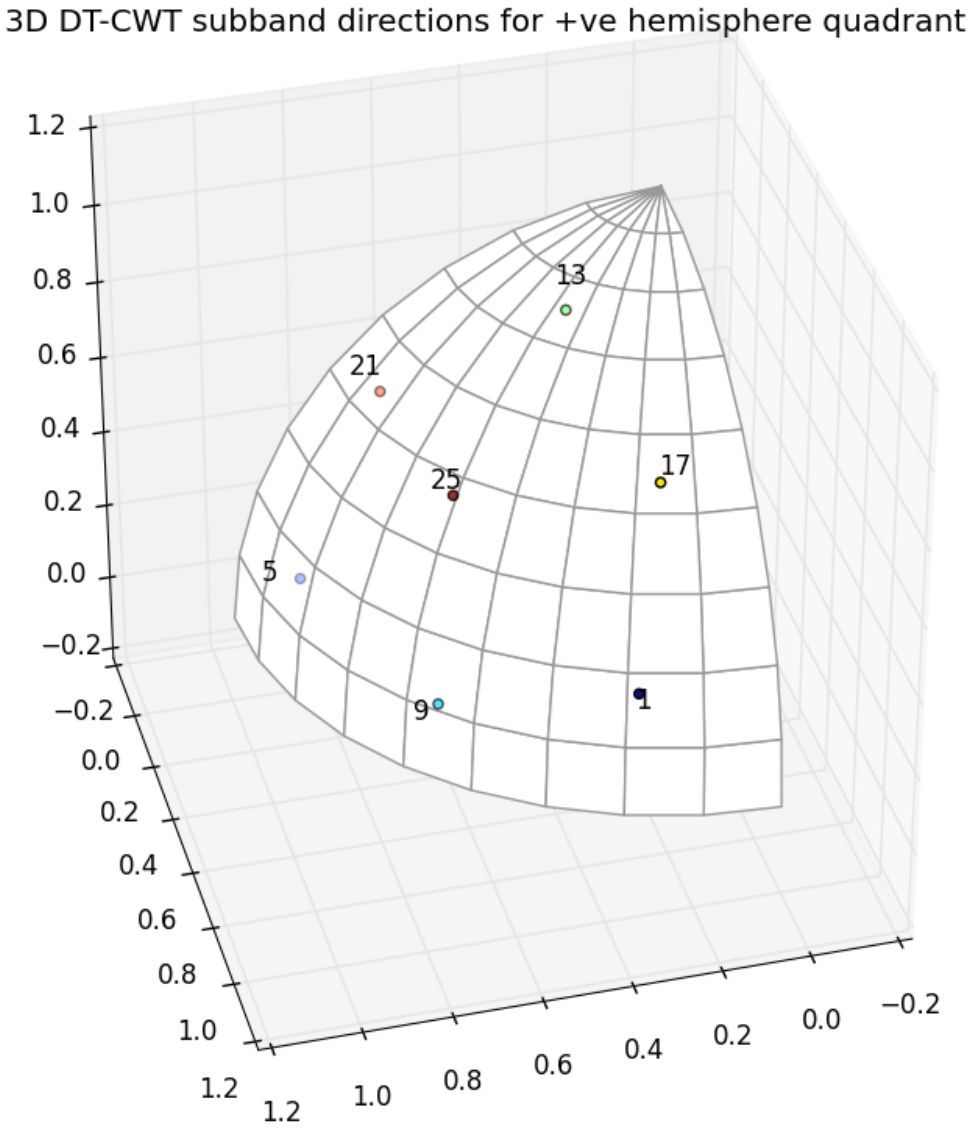
For a further directional sensitivity example, see *Showing 3D Directional Sensitivity*.



2.2 Example scripts

2.2.1 Showing 3D Directional Sensitivity

The `3d_dtcwt_directionality.py` script in the examples directory shows how one may demonstrate the directional sensitivity of the 3D DT-CWT complex subband coefficients. It computes empirically the maximally sensitive directions for each subband and plots them in an interactive figure using matplotlib. A screenshot is reproduced below:



There are some points to note about this diagram. Each subband is labeled sich that '1' refers to the first subband, '5' the fifth and so forth. On this diagram the subbands are all four apart reflecting the fact that, for example, subbands 2, 3 and 4 are positioned in the other four quadrants of the upper hemisphere reflecting the position of subband 1. There are seven visible subband directions in the +ve quadrant of the hemisphere and hence there are 28 directions in total over all four quadrants.

The source for the script is shown below:

```
#!/bin/python

"""
An example of the directional selectivity of 3D DT-CWT coefficients.

This example creates a 3D array holding an image of a sphere and performs the
3D DT-CWT transform on it. The locations of maxima (and their images about the
mid-point of the image) are determined for each complex coefficient at level 2.
These maxima points are then shown on a single plot to demonstrate the
directions in which the 3D DT-CWT transform is selective.

"""

# Import the libraries we need
from matplotlib.pyplot import *
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from dtcwt import dtwavexfm3, dtwaveifm3, biort, qshift

# Specify details about sphere and grid size
GRID_SIZE = 128
SPHERE_RAD = int(0.45 * GRID_SIZE) + 0.5

# Compute an image of the sphere
grid = np.arange(-(GRID_SIZE>>1), GRID_SIZE>>1)
X, Y, Z = np.meshgrid(grid, grid, grid)
r = np.sqrt(X*X + Y*Y + Z*Z)
sphere = (0.5 + np.clip(SPHERE_RAD-r, -0.5, 0.5)).astype(np.float32)

# Specify number of levels and wavelet family to use
nlevels = 2
b = biort('near_sym_a')
q = qshift('qshift_a')

# Form the DT-CWT of the sphere. We use discard_level_1 since we're
# uninterested in the inverse transform and this saves us some memory.
Yl, Yh = dtwavexfm3(sphere, nlevels, b, q, discard_level_1=True)

# Plot maxima
figure(figsize=(8,8))

ax = gcf().add_subplot(1,1,1, projection='3d')
ax.set_aspect('equal')
ax.view_init(35, 75)

# Plot unit sphere +ve octant
thetas = np.linspace(0, np.pi/2, 10)
phis = np.linspace(0, np.pi/2, 10)

def sphere_to_xyz(r, theta, phi):
    st, ct = np.sin(theta), np.cos(theta)
    sp, cp = np.sin(phi), np.cos(phi)
    return r * np.asarray((st*cp, st*sp, ct))

tris = []
rad = 0.99 # so that points plotted latter are not z-clipped
```

```
for t1, t2 in zip(thetas[:-1], thetas[1:]):
    for p1, p2 in zip(phis[:-1], phis[1:]):
        tris.append([
            sphere_to_xyz(rad, t1, p1),
            sphere_to_xyz(rad, t1, p2),
            sphere_to_xyz(rad, t2, p2),
            sphere_to_xyz(rad, t2, p1),
        ])

sphere = Poly3DCollection(tris, facecolor='w', edgecolor=(0.6,0.6,0.6))
ax.add_collection3d(sphere)

locs = []
scale = 1.1
for idx in xrange(Yh[-1].shape[3]):
    Z = Yh[-1][:,:,:,idx]
    C = np.abs(Z)
    max_loc = np.asarray(np.unravel_index(np.argmax(C), C.shape)) - np.asarray(C.shape)*0.5
    max_loc /= np.sqrt(np.sum(max_loc * max_loc))

    # Only record directions in the +ve octant (or those from the -ve quadrant
    # which can be flipped).
    if np.all(np.sign(max_loc) == 1):
        locs.append(max_loc)
        ax.text(max_loc[0] * scale, max_loc[1] * scale, max_loc[2] * scale, str(idx+1))
    elif np.all(np.sign(max_loc) == -1):
        locs.append(-max_loc)
        ax.text(-max_loc[0] * scale, -max_loc[1] * scale, -max_loc[2] * scale, str(idx+1))

# Plot all directions as a scatter plot
locs = np.asarray(locs)
ax.scatter(locs[:,0], locs[:,1], locs[:,2], c=np.arange(locs.shape[0]))

w = 1.1
ax.auto_scale_xyz([0, w], [0, w], [0, w])

legend()
title('3D DT-CWT subband directions for +ve hemisphere quadrant')
tight_layout()

show()

# vim:sw=4:sts=4:et
```

2.3 API Reference

2.3.1 Computing the DT-CWT

dtcwt.**dtwavexfm**(*X*, *nlevels=3*, *biort='near_sym_a'*, *qshift='qshift_a'*, *include_scale=False*)

Perform a *n*-level DTCWT decompostion on a 1D column vector *X* (or on the columns of a matrix *X*).

Parameters

- **X** – 1D real array or 2D real array whose columns are to be transformed
- **nlevels** – Number of levels of wavelet decomposition

- **biort** – Level 1 wavelets to use. See `biort()`.
- **qshift** – Level >= 2 wavelets to use. See `qshift()`.

Returns Yl The real lowpass image from the final level

Returns Yh A tuple containing the (N, M, 6) shape complex highpass subimages for each level.

Returns Yscale If *include_scale* is True, a tuple containing real lowpass coefficients for every scale.

If *biort* or *qshift* are strings, they are used as an argument to the `biort()` or `qshift()` functions. Otherwise, they are interpreted as tuples of vectors giving filter coefficients. In the *biort* case, this should be (h0o, g0o, h1o, g1o). In the *qshift* case, this should be (h0a, h0b, g0a, g0b, h1a, h1b, g1a, g1b).

Example:

```
# Performs a 5-level transform on the real image X using the 13,19-tap
# filters for level 1 and the Q-shift 14-tap filters for levels >= 2.
Yl, Yh = dtwavexfm(X,5,'near_sym_b','qshift_b')
```

dtcwt.**dtwaveifm**(*Yl*, *Yh*, *biort='near_sym_a'*, *qshift='qshift_a'*, *gain_mask=None*)

Perform an *n*-level dual-tree complex wavelet (DTCWT) 1D reconstruction.

Parameters

- **Yl** – The real lowpass subband from the final level
- **Yh** – A sequence containing the complex highpass subband for each level.
- **biort** – Level 1 wavelets to use. See `biort()`.
- **qshift** – Level >= 2 wavelets to use. See `qshift()`.
- **gain_mask** – Gain to be applied to each subband.

Returns Z Reconstructed real array.

The *l*-th element of *gain_mask* is gain for wavelet subband at level l. If gain_mask[l] == 0, no computation is performed for band *l*. Default *gain_mask* is all ones. Note that *l* is 0-indexed.

If *biort* or *qshift* are strings, they are used as an argument to the `biort()` or `qshift()` functions. Otherwise, they are interpreted as tuples of vectors giving filter coefficients. In the *biort* case, this should be (h0o, g0o, h1o, g1o). In the *qshift* case, this should be (h0a, h0b, g0a, g0b, h1a, h1b, g1a, g1b).

Example:

```
# Performs a reconstruction from Yl,Yh using the 13,19-tap filters
# for level 1 and the Q-shift 14-tap filters for levels >= 2.
Z = dtwaveifm(Yl, Yh, 'near_sym_b', 'qshift_b')
```

dtcwt.**dtwavexfm2**(*X*, *nlevels=3*, *biort='near_sym_a'*, *qshift='qshift_a'*, *include_scale=False*)

Perform a *n*-level DTCWT-2D decompostion on a 2D matrix *X*.

Parameters

- **X** – 2D real array
- **nlevels** – Number of levels of wavelet decomposition
- **biort** – Level 1 wavelets to use. See `biort()`.
- **qshift** – Level >= 2 wavelets to use. See `qshift()`.

Returns Yl The real lowpass image from the final level

Returns Yh A tuple containing the complex highpass subimages for each level.

Returns Yscale If *include_scale* is True, a tuple containing real lowpass coefficients for every scale.

If *biort* or *qshift* are strings, they are used as an argument to the `biort()` or `qshift()` functions. Otherwise, they are interpreted as tuples of vectors giving filter coefficients. In the *biort* case, this should be (h0o, g0o, h1o, g1o). In the *qshift* case, this should be (h0a, h0b, g0a, g0b, h1a, h1b, g1a, g1b).

Example:

```
# Performs a 3-level transform on the real image X using the 13,19-tap
# filters for level 1 and the Q-shift 14-tap filters for levels >= 2.
Yl, Yh = dtwavexfm2(X, 3, 'near_sym_b', 'qshift_b')
```

dtcwt.**dtwaveifm2**(*Yl*, *Yh*, *biort='near_sym_a'*, *qshift='qshift_a'*, *gain_mask=None*)

Perform an *n*-level dual-tree complex wavelet (DTCWT) 2D reconstruction.

Parameters

- **Yl** – The real lowpass subband from the final level
- **Yh** – A sequence containing the complex highpass subband for each level.
- **biort** – Level 1 wavelets to use. See `biort()`.
- **qshift** – Level >= 2 wavelets to use. See `qshift()`.
- **gain_mask** – Gain to be applied to each subband.

Returns Z Reconstructed real array

The (*d*, *l*)-th element of *gain_mask* is gain for subband with direction *d* at level *l*. If gain_mask[d,l] == 0, no computation is performed for band (d,l). Default *gain_mask* is all ones. Note that both *d* and *l* are zero-indexed.

If *biort* or *qshift* are strings, they are used as an argument to the `biort()` or `qshift()` functions. Otherwise, they are interpreted as tuples of vectors giving filter coefficients. In the *biort* case, this should be (h0o, g0o, h1o, g1o). In the *qshift* case, this should be (h0a, h0b, g0a, g0b, h1a, h1b, g1a, g1b).

Example:

```
# Performs a 3-level reconstruction from Yl,Yh using the 13,19-tap
# filters for level 1 and the Q-shift 14-tap filters for levels >= 2.
Z = dtwaveifm2(Yl, Yh, 'near_sym_b', 'qshift_b')
```

dtcwt.**dtwavexfm3**(*X*, *nlevels=3*, *biort='near_sym_a'*, *qshift='qshift_a'*, *ext_mode=4*, *discard_level_1=False*)

Perform a *n*-level DTCWT-3D decompostion on a 3D matrix *X*.

Parameters

- **X** – 3D real array-like object
- **nlevels** – Number of levels of wavelet decomposition
- **biort** – Level 1 wavelets to use. See `biort()`.
- **qshift** – Level >= 2 wavelets to use. See `qshift()`.
- **ext_mode** – Extension mode. See below.
- **discard_level_1** – True if level 1 high-pass bands are to be discarded.

Returns Yl The real lowpass image from the final level

Returns Yh A tuple containing the complex highpass subimages for each level.

Each element of *Yh* is a 4D complex array with the 4th dimension having size 28. The 3D slice `Yh[l][:,:,:,d]` corresponds to the complex higpass coefficients for direction d at level l where d and l are both 0-indexed.

If *biort* or *qshift* are strings, they are used as an argument to the `biort()` or `qshift()` functions. Otherwise, they are interpreted as tuples of vectors giving filter coefficients. In the *biort* case, this should be (h0o, g0o, h1o, g1o). In the *qshift* case, this should be (h0a, h0b, g0a, g0b, h1a, h1b, g1a, g1b).

There are two values for *ext_mode*, either 4 or 8. If *ext_mode* = 4, check whether 1st level is divisible by 2 (if not we raise a `ValueError`). Also check whether from 2nd level onwards, the coefs can be divided by 4. If any dimension size is not a multiple of 4, append extra coefs by repeating the edges. If *ext_mode* = 8, check whether 1st level is divisible by 4 (if not we raise a `ValueError`). Also check whether from 2nd level onwards, the coeffs can be divided by 8. If any dimension size is not a multiple of 8, append extra coeffs by repeating the edges twice.

If *discard_level_1* is True the highpass coefficients at level 1 will be discarded. (And, in fact, will never be calculated.) This turns the transform from being 8:1 redundant to being 1:1 redundant at the cost of no-longer allowing perfect reconstruction. If this option is selected then *Yh[0]* will be *None*. Note that `dtwaveifm3()` will accepts *Yh[0]* being *None* and will treat it as being zero.

Example:

```
# Performs a 3-level transform on the real 3D array X using the 13,19-tap
# filters for level 1 and the Q-shift 14-tap filters for levels >= 2.
Yl, Yh = dtwavexfm3(X, 3, 'near_sym_b', 'qshift_b')
```

dtcwt.**dtwaveifm3**(*Yl*, *Yh*, *biort='near_sym_a'*, *qshift='qshift_a'*, *ext_mode=4*)

Perform an *n*-level dual-tree complex wavelet (DTCWT) 3D reconstruction.

Parameters

- **Yl** – The real lowpass subband from the final level
- **Yh** – A sequence containing the complex highpass subband for each level.
- **biort** – Level 1 wavelets to use. See `biort()`.
- **qshift** – Level >= 2 wavelets to use. See `qshift()`.
- **ext_mode** – Extension mode. See below.

Returns Z Reconstructed real image matrix.

If *biort* or *qshift* are strings, they are used as an argument to the `biort()` or `qshift()` functions. Otherwise, they are interpreted as tuples of vectors giving filter coefficients. In the *biort* case, this should be (h0o, g0o, h1o, g1o). In the *qshift* case, this should be (h0a, h0b, g0a, g0b, h1a, h1b, g1a, g1b).

There are two values for *ext_mode*, either 4 or 8. If *ext_mode* = 4, check whether 1st level is divisible by 2 (if not we raise a `ValueError`). Also check whether from 2nd level onwards, the coefs can be divided by 4. If any dimension size is not a multiple of 4, append extra coefs by repeating the edges. If *ext_mode* = 8, check whether 1st level is divisible by 4 (if not we raise a `ValueError`). Also check whether from 2nd level onwards, the coeffs can be divided by 8. If any dimension size is not a multiple of 8, append extra coeffs by repeating the edges twice.

Example:

```
# Performs a 3-level reconstruction from Yl,Yh using the 13,19-tap
# filters for level 1 and the Q-shift 14-tap filters for levels >= 2.
Z = dtwaveifm3(Yl, Yh, 'near_sym_b', 'qshift_b')
```

dtcwt.**biort**(*name*)

Load level 1 wavelet by name.

Parameters **name** – a string specifying the wavelet family name

Returns a tuple of vectors giving filter coefficients

Name	Wavelet
antonini	Antonini 9,7 tap filters.
legall	LeGall 5,3 tap filters.
near_sym_a	Near-Symmetric 5,7 tap filters.
near_sym_b	Near-Symmetric 13,19 tap filters.

Return a tuple whose elements are a vector specifying the h0o, g0o, h1o and g1o coefficients.

Raises

- **IOError** – if name does not correspond to a set of wavelets known to the library.
- **ValueError** – if name specifies a `qshift()` wavelet.

`dtcwt.`**`qshift`**(*name*)

Load level >=2 wavelet by name,

Parameters **name** – a string specifying the wavelet family name

Returns a tuple of vectors giving filter coefficients

Name	Wavelet
qshift_06	Quarter Sample Shift Orthogonal (Q-Shift) 10,10 tap filters, (only 6,6 non-zero taps).
qshift_a	Q-shift 10,10 tap filters, (with 10,10 non-zero taps, unlike qshift_06).
qshift_b	Q-Shift 14,14 tap filters.
qshift_c	Q-Shift 16,16 tap filters.
qshift_d	Q-Shift 18,18 tap filters.

Return a tuple whose elements are a vector specifying the h0a, h0b, g0a, g0b, h1a, h1b, g1a and g1b coefficients.

Raises

- **IOError** – if name does not correspond to a set of wavelets known to the library.
- **ValueError** – if name specifies a `biort()` wavelet.

2.3.2 Keypoint analysis

`dtcwt.keypoint.`**`find_keypoints`**(*highpass_subbands*, *method=None*, *alpha=1.0*, *beta=0.4*, *kappa=0.16666666666666666*, *threshold=None*, *max_points=None*, *upsample_keypoint_energy=None*, *upsample_subbands=None*, *refine_positions=True*, *skip_levels=1*)

Parameters

- **highpass_subbands** – (NxMx6) matrix of highpass subband images
- **method** – *(optional)* string specifying which keypoint energy method to use
- **alpha** – *(optional)* scale parameter for `'fauqueur'` method
- **beta** – *(optional)* shape parameter for `'fauqueur'` method
- **kappa** – *(optiona)* suppression parameter for `'kingsbury'` method
- **threshold** – *(optional)* minimum keypoint energy of returned keypoints
- **max_points** – *(optional)* maximum number of keypoints to return
- **upsample_keypoint_energy** – is non-None, a string specifying a method used to upscale the keypoint energy map before finding keypoints
- **upsample_subands** – is non-None, a string specifying a method used to upscale the subband image before finding keypoints

- **refine_positions** – *(optional)* should the keypoint positions be refined to sub-pixel accuracy
- **skip_levels** – *(optional)* number of levels of the transform to ignore before looking for keypoints

Returns (Px4) array of P keypoints in image co-ordinates

Warning: The interface and behaviour of this function is the subject of an open research project. It is provided in this release as a preview of forthcoming functionality but it is subject to change between releases.

The rows of the returned keypoint array give the x co-ordinate, y co-ordinate, scale and keypoint energy. The rows are sorted in order of decreasing keypoint energy.

If *refine_positions* is `True` then the positions (and energy) of the keypoints will be refined to sub-pixel accuracy by fitting a quadratic patch. If *refine_positions* is `False` then the keypoint locations will be those corresponding directly to pixel-wise maxima of the subband images.

The *max_points* and *threshold* parameters are cumulative: if both are specified then the *max_points* greatest energy keypoints with energy greater than *threshold* will be returned.

Usually the keypoint energies returned from the finest scale level are dominated by noise and so one usually wants to specify *skip_levels* to be 1 or 2. If *skip_levels* is 0 then all levels will be used to compute keypoint energy.

The *upsample_subbands* and *upsample_keypoint_energy* parameters are used to control whether the individual subband coefficients and/org the keypoint energy map are upscaled by 2 before finding keypoints. If these parameters are None then no corresponding upscaling is performed. If non-None they specify the upscale method as outlined in `dtcwt.sampling.upsample()`.

If *method* is `None`, the default `'fauqueur'` method is used.

Name	Description	Parameters used
fauqueur	Geometric mean of absolute values[1]	*alpha*, *beta*
bendale	Minimum absolute value[2]	none
kingsbury	Cross-product of orthogonal subbands	*kappa*

[1] Julien Fauqueur, Nick Kingsbury, and Ryan Anderson. *Multiscale Keypoint Detection using the Dual-Tree Complex Wavelet Transform*. 2006 International Conference on Image Processing, pages 1625-1628, October 2006. ISSN 1522-4880. doi: 10.1109/ICIP.2006.312656. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4106857.

[2] Pashmina Bendale, Bill Triggs, and Nick Kingsbury. *Multiscale Keypoint Analysis based on Complex Wavelets*. In British Machine Vision Con-ference (BMVC), 2010. http://www-sigproc.eng.cam.ac.uk/~pb397/publications/BTK_BMVC_2010_abstract.pdf.

2.3.3 Image sampling

Rescaling and re-sampling high- and low-pass subbands.

`dtcwt.sampling.`**`sample`**(*im*, *xs*, *ys*, *method=None*)

Sample image at (x,y) given by elements of *xs* and *ys*. Both *xs* and *ys* must have identical shape and output will have this same shape. The location (x,y) refers to the *centre* of `im[y,x]`. Samples at fractional locations are calculated using the method specified by *method* (or `'lanczos'` if *method* is `None`.)

Parameters

- **im** – array to sample from
- **xs** – x co-ordinates to sample

- **ys** – y co-ordinates to sample
- **method** – one of 'bilinear', 'lanczos' or 'nearest'

Raises ValueError if `xs` and `ys` have differing shapes

`dtcwt.sampling.`**`sample_highpass`**(*im*, *xs*, *ys*, *method=None*)
As `sample()` except that the highpass image is first phase shifted to be centred on approximately DC.

`dtcwt.sampling.`**`rescale`**(*im*, *shape*, *method=None*)
Return a resampled version of *im* scaled to *shape*.

Since the centre of pixel (x,y) has co-ordinate (x,y) the extent of *im* is actually $x \in (-0.5, w - 0.5]$ and $y \in (-0.5, h - 0.5]$ where (y,x) is `im.shape`. This returns a sampled version of *im* that has the same extent as a *shape*-sized array.

`dtcwt.sampling.`**`rescale_highpass`**(*im*, *shape*, *method=None*)
As `rescale()` except that the highpass image is first phase shifted to be centred on approximately DC.

`dtcwt.sampling.`**`upsample`**(*image*, *method=None*)
Specialised function to upsample an image by a factor of two using a specified sampling method. If *image* is an array of shape (NxMx...) then the output will have shape (2Nx2Mx...). Only rows and columns are upsampled, depth axes and greater are interpolated but are not upsampled.

Parameters

- **image** – an array containing the image to upsample
- **method** – if non-None, a string specifying the sampling method to use.

If *method* is `None`, the default sampling method `'lanczos'` is used. The following sampling methods are supported:

Name	Description
nearest	Nearest-neighbour sampling
bilinear	Bilinear sampling
lanczos	Lanczos sampling with window radius of 3

`dtcwt.sampling.`**`upsample_highpass`**(*im*, *method=None*)
As `upsample()` except that the highpass image is first phase rolled so that the filter has approximate DC centre frequency. The upshot is that this is the function to use when re-sampling complex subband images.

2.3.4 Low-level support functions

A normal user should not need to call these functions but they are documented here just in case you do.

`dtcwt.lowlevel.`**`appropriate_complex_type_for`**(*X*)
Return an appropriate complex data type depending on the type of X. If X is already complex, return that, if it is floating point return a complex type of the appropriate size and if it is integer, choose an complex floating point type depending on the result of `numpy.asfarray()`.

`dtcwt.lowlevel.`**`as_column_vector`**(*v*)
Return *v* as a column vector with shape (N,1).

`dtcwt.lowlevel.`**`asfarray`**(*X*)
Similar to `numpy.asfarray()` except that this function tries to preserve the original datatype of X if it is already a floating point type and will pass floating point arrays through directly without copying.

`dtcwt.lowlevel.`**`coldfilt`**(*X*, *ha*, *hb*)
Filter the columns of image X using the two filters ha and hb = reverse(ha). ha operates on the odd samples of

X and hb on the even samples. Both filters should be even length, and h should be approx linear phase with a quarter sample advance from its mid pt (i.e. $|h(m/2)| > |h(m/2+1)|$).

```
                  ext        top edge                    bottom edge       ext
Level 1:        !              |              !              |              !
odd filt on .    b   b   b   b   a   a   a   a   a   a   a   a   b   b   b   b
odd filt on .      a   a   a   a   b   b   b   b   b   b   b   b   a   a   a   a
Level 2:        !               |              !               |              !
+q filt on x      b       b       a       a       a       a       b       b
-q filt on o          a       a       b       b       b       b       a       a
```

The output is decimated by two from the input sample rate and the results from the two filters, Ya and Yb, are interleaved to give Y. Symmetric extension with repeated end samples is used on the composite X columns before each filter is applied.

Raises ValueError if the number of rows in X is not a multiple of 4, the length of ha does not match hb or the lengths of ha or hb are non-even.

dtcwt.lowlevel.**colfilter**(*X*, *h*)

Filter the columns of image *X* using filter vector *h*, without decimation. If len(h) is odd, each output sample is aligned with each input sample and *Y* is the same size as *X*. If len(h) is even, each output sample is aligned with the mid point of each pair of input samples, and Y.shape = X.shape + [1 0].

Parameters

- **X** – an image whose columns are to be filtered
- **h** – the filter coefficients.

Returns Y the filtered image.

dtcwt.lowlevel.**colifilt**(*X*, *ha*, *hb*)

Filter the columns of image X using the two filters ha and hb = reverse(ha). ha operates on the odd samples of X and hb on the even samples. Both filters should be even length, and h should be approx linear phase with a quarter sample advance from its mid pt (i.e *:math:'|h(m/2)| > |h(m/2 + 1)|*).

```
                  ext       left edge                   right edge       ext
Level 2:        !               |              !               |              !
+q filt on x      b       b       a       a       a       a       b       b
-q filt on o          a       a       b       b       b       b       a       a
Level 1:        !               |              !               |              !
odd filt on .    b   b   b   b   a   a   a   a   a   a   a   a   b   b   b   b
odd filt on .      a   a   a   a   b   b   b   b   b   b   b   b   a   a   a   a
```

The output is interpolated by two from the input sample rate and the results from the two filters, Ya and Yb, are interleaved to give Y. Symmetric extension with repeated end samples is used on the composite X columns before each filter is applied.

dtcwt.lowlevel.**reflect**(*x*, *minx*, *maxx*)

Reflect the values in matrix *x* about the scalar values *minx* and *maxx*. Hence a vector *x* containing a long linearly increasing series is converted into a waveform which ramps linearly up and down between *minx* and *maxx*. If *x* contains integers and *minx* and *maxx* are (integers + 0.5), the ramps will have repeated max and min samples.

CHAPTER 3

Licence

The original toolbox is copyrighted and there are some restrictions on use which are outlined in the file `ORIGINAL_README.txt`. Aside from portions directly derived from the original MATLAB toolbox, any additions in this library and this documentation are licensed under the 2-clause BSD licence as documented in the file `COPYING.txt`.

Python Module Index

d