dtcwt Documentation

Release 0.11.0

Rich Wareham, Nick Kingsbury, Cian Shaffrey

March 09, 2016

Contents

1 Comparison with MATLAB toolbox **3**

2 Contents **5**
2.1 Getting Started 5
2.2 Performing the DTCWT 6
2.3 Multiple Backend Support 20
2.4 DTCWT-based algorithms 22
2.5 Example scripts 29
2.6 API Reference 31

Python Module Index **53**

The `dtcwt` library provides a Python implementation of the 1, 2 and 3-D dual-tree complex wavelet transform along with some associated algorithms. It contains a pure CPU implementation which makes use of NumPy along with an accelerated GPU implementation using OpenCL.

CHAPTER 1

Comparison with MATLAB toolbox

The canonical implementation of the DT-CWT is that provided by Professor Nick Kingsbury on his website. This library aims to have near-identical output (to within a small multiple of machine precision). Significant deviation is a bug and should be reported. Cross-verification of the transform output is part of the test suite and each and every change is checked against that test suite automatically.

It is hoped that testing this will allow confidence in this library being suitable for porting existing MATLAB scripts over to Python. To that end there is a `dtcwt.compat` module which provides an API similar to the original MATLAB toolbox.

CHAPTER 2

Contents

2.1 Getting Started

This library provides support for computing 1D, 2D and 3D dual-tree complex wavelet transforms and their inverse in Python along with some signal processing algorithms which make use of the DTCWT.

This section will guide you through using the `dtcwt` library. See API Reference for full details on the library's API.

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:

```
$ pip install -r tests/requirements.txt
$ py.test
```

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
```

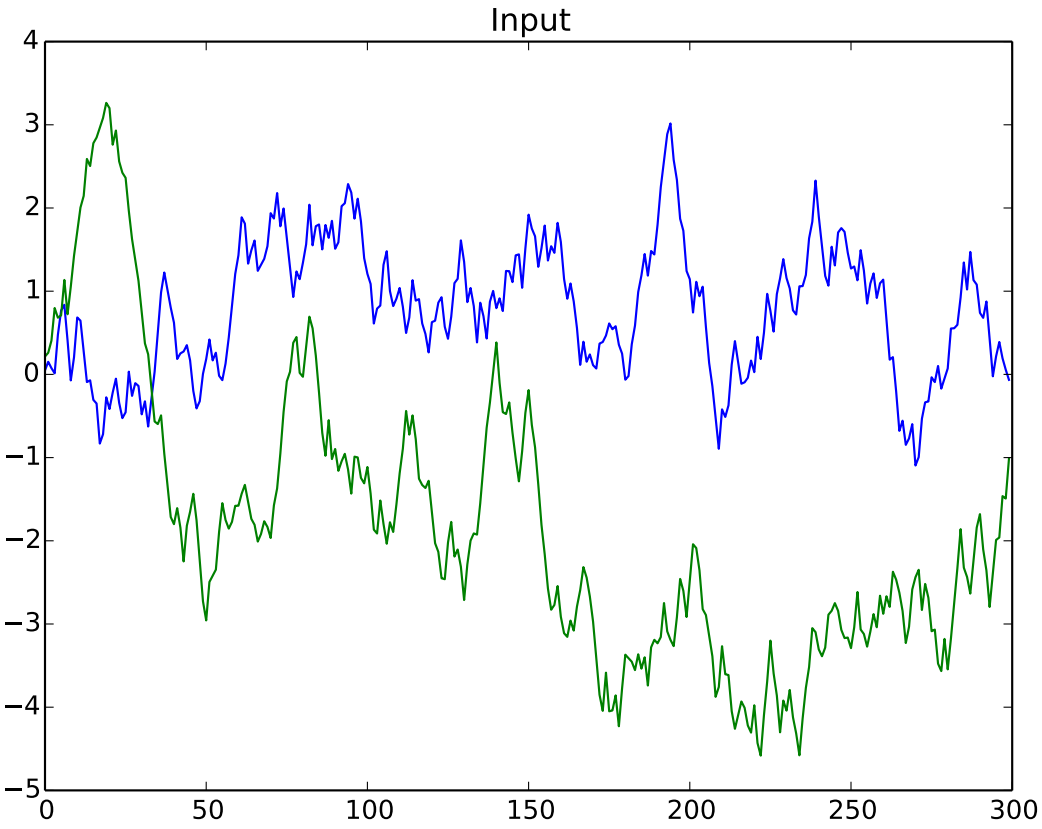
Compiled documentation may be found in `build/docs/html/`.

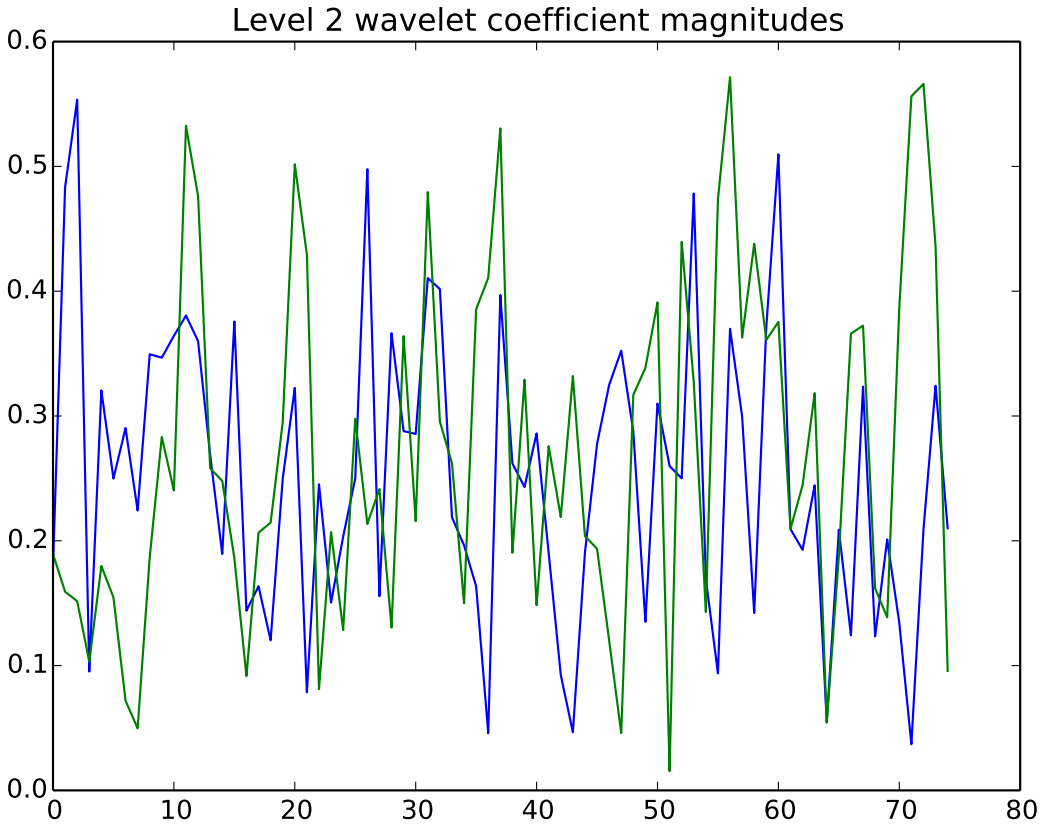
2.2 Performing the DTCWT

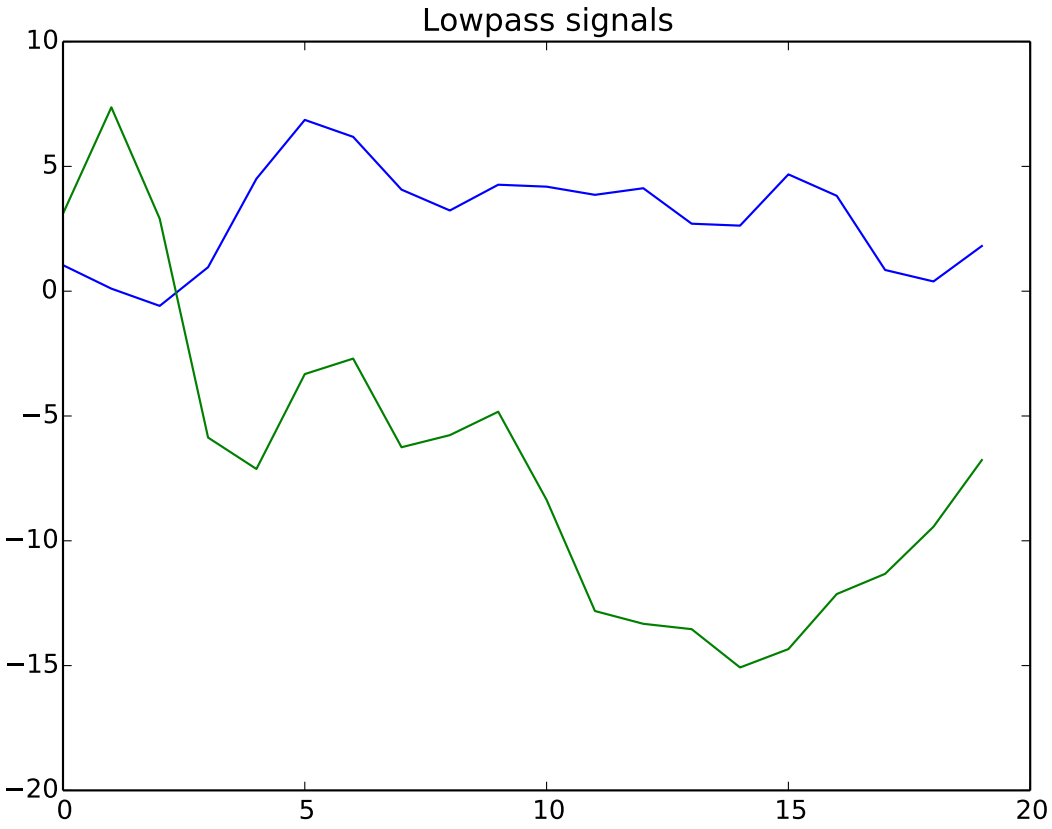
2.2.1 1D transform

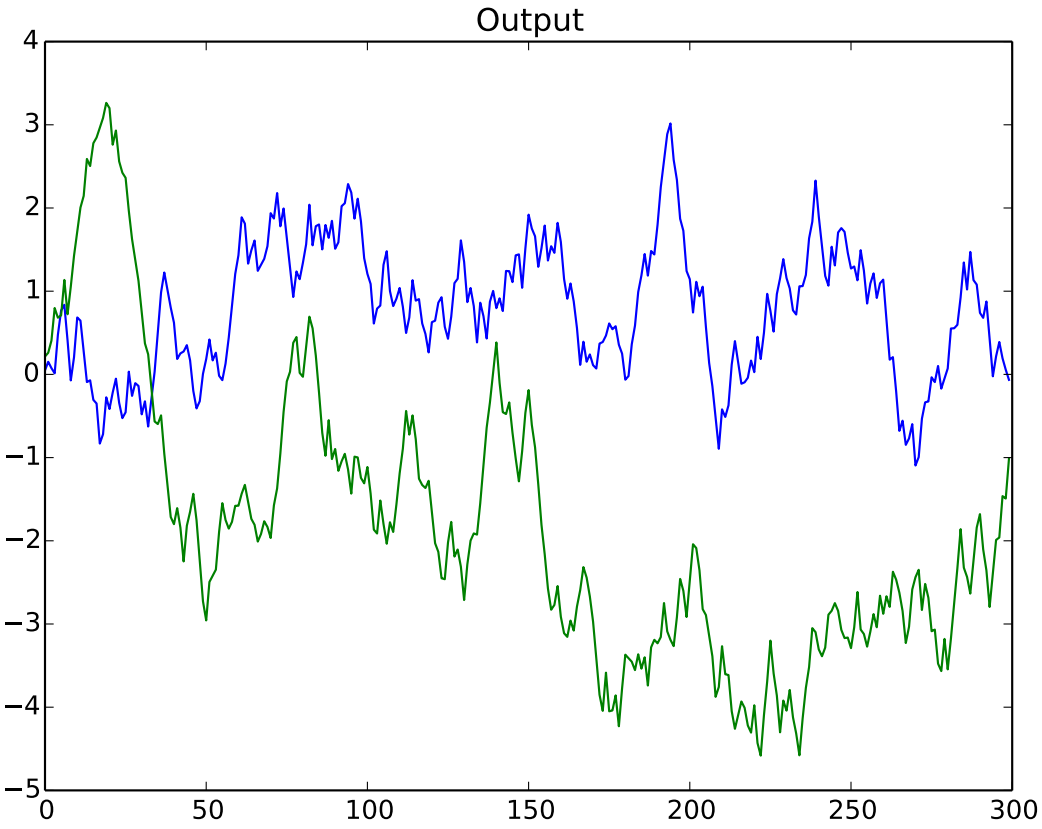
This example generates two 1D random walks and demonstrates reconstructing them using the forward and inverse 1D transforms. Note that :py:func‘dtcwt.Transform1d.forward‘ and *dtcwt.Transform1d.inverse()* will transform columns of an input array independently

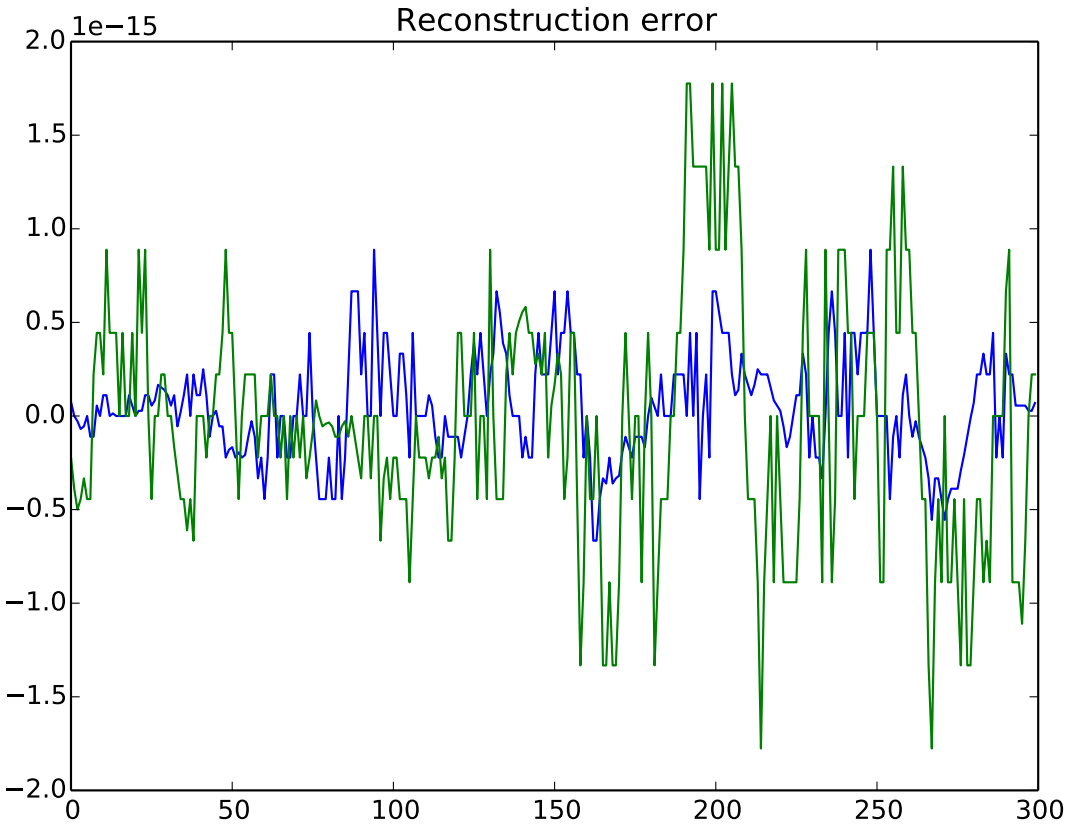
```
from matplotlib.pylab import *
import dtcwt

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

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

# 1D transform, 5 levels
transform = dtcwt.Transform1d()
vecs_t = transform.forward(vecs, nlevels=5)

# Show level 2 highpass coefficient magnitudes
figure()
plot(np.abs(vecs_t.highpasses[1]))
title('Level 2 wavelet coefficient magnitudes')

# Show last level lowpass image
figure()
plot(vecs_t.lowpass)
title('Lowpass signals')

# Inverse
vecs_recon = transform.inverse(vecs_t)

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

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

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


Input
4
3
2
1
0
−1
−2
−3
−4
−5
0
50
100
150
200
250
300


Level 2 wavelet coefficient magnitudes
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
10
20
30
40
50
60
70
80


Lowpass signals
10
5
0
−5
−10
−15
−20
0
5
10
15
20


Output
4
3
2
1
0
−1
−2
−3
−4
−5
0
50
100
150
200
250
300

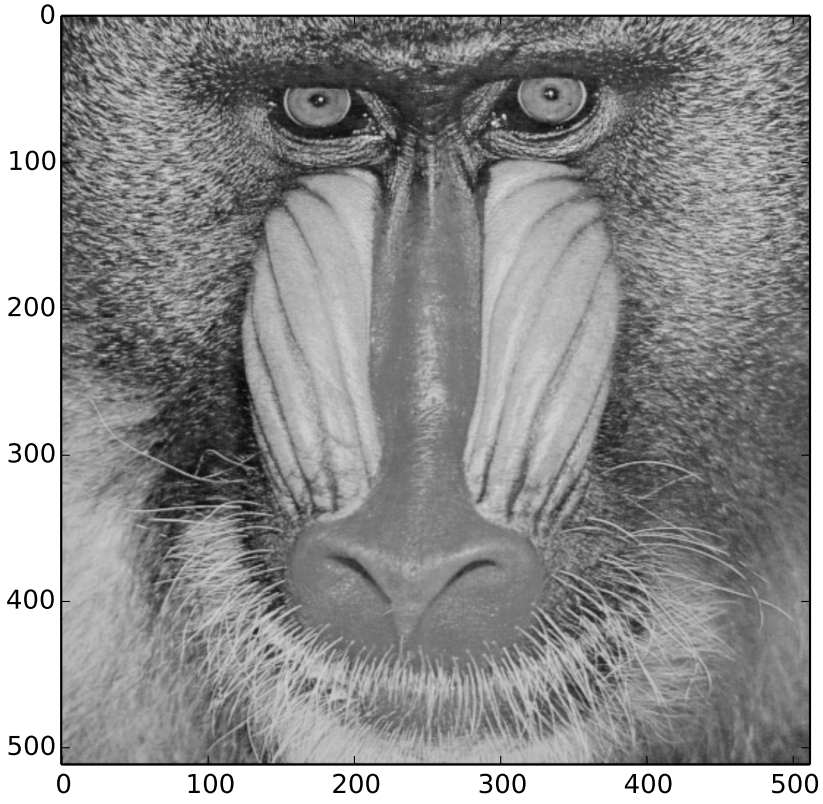


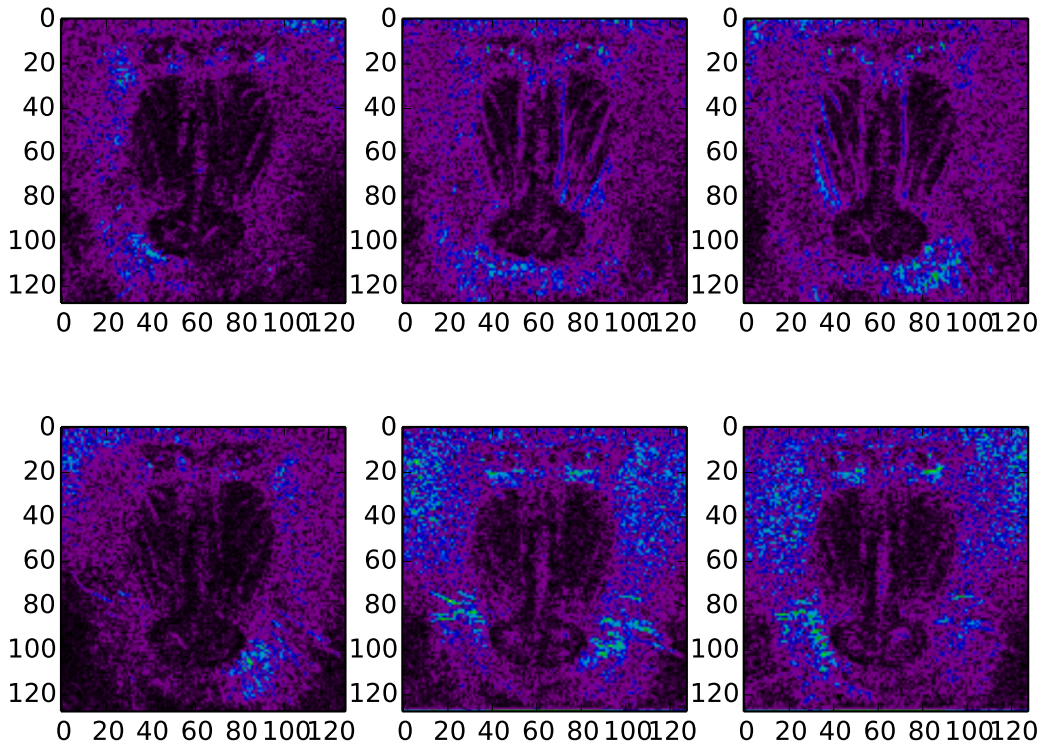
2.2.2 2D transform

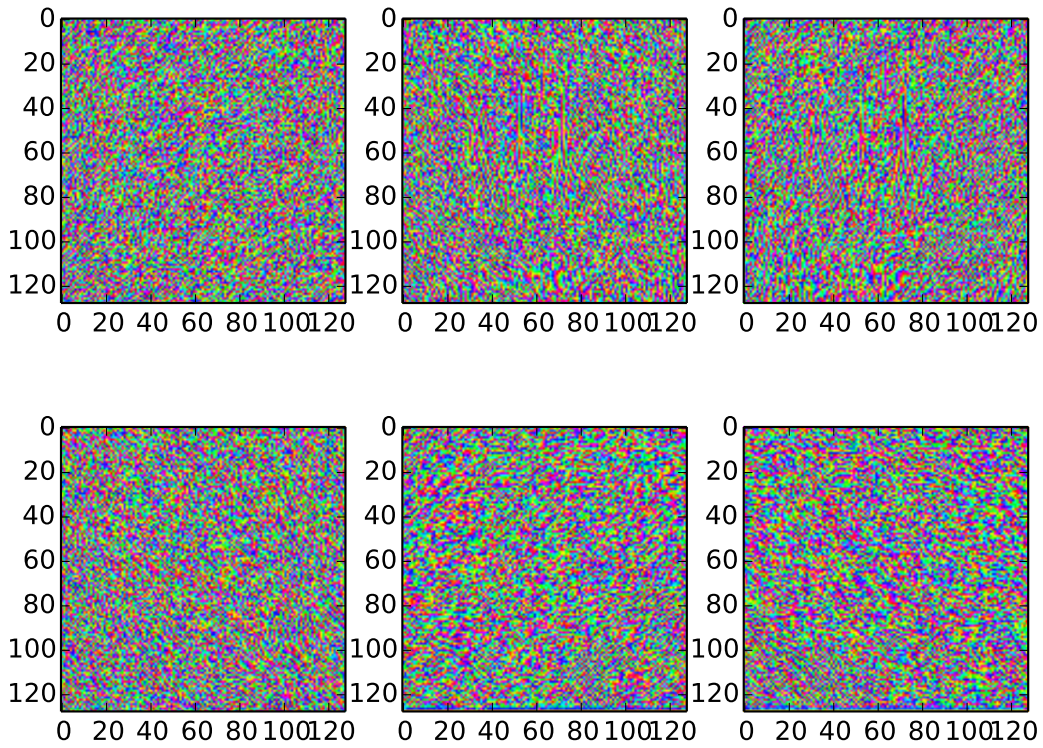
Using the pylab environment (part of matplotlib) we can perform a simple example where we transform the standard 'mandrill' image and show the level 2 wavelet coefficients:

```
# Load the mandrill image
mandrill = datasets.mandrill()

# Show mandrill
figure(1)
imshow(mandrill, cmap=cm.gray, clim=(0,1))

import dtcwt
transform = dtcwt.Transform2d()

# Compute two levels of dtcwt with the defaul wavelet family
mandrill_t = transform.forward(mandrill, nlevels=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 range(mandrill_t.highpasses[1].shape[2]):
    subplot(2, 3, slice_idx)
    imshow(np.abs(mandrill_t.highpasses[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 range(mandrill_t.highpasses[1].shape[2]):
    subplot(2, 3, slice_idx)
    imshow(np.angle(mandrill_t.highpasses[1][:,:,slice_idx]), cmap=cm.hsv, clim=(-np.pi, np.pi))
```




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



2.2.3 3D transform

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

```
from matplotlib.pylab import *
import dtcwt
```

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)

trans = dtcwt.Transform3d()
sphere_t = trans.forward(sphere, nlevels=2)
```

The function returns a *dtcwt.Pyramid* instance containing the lowpass image and a tuple of complex highpass coefficients

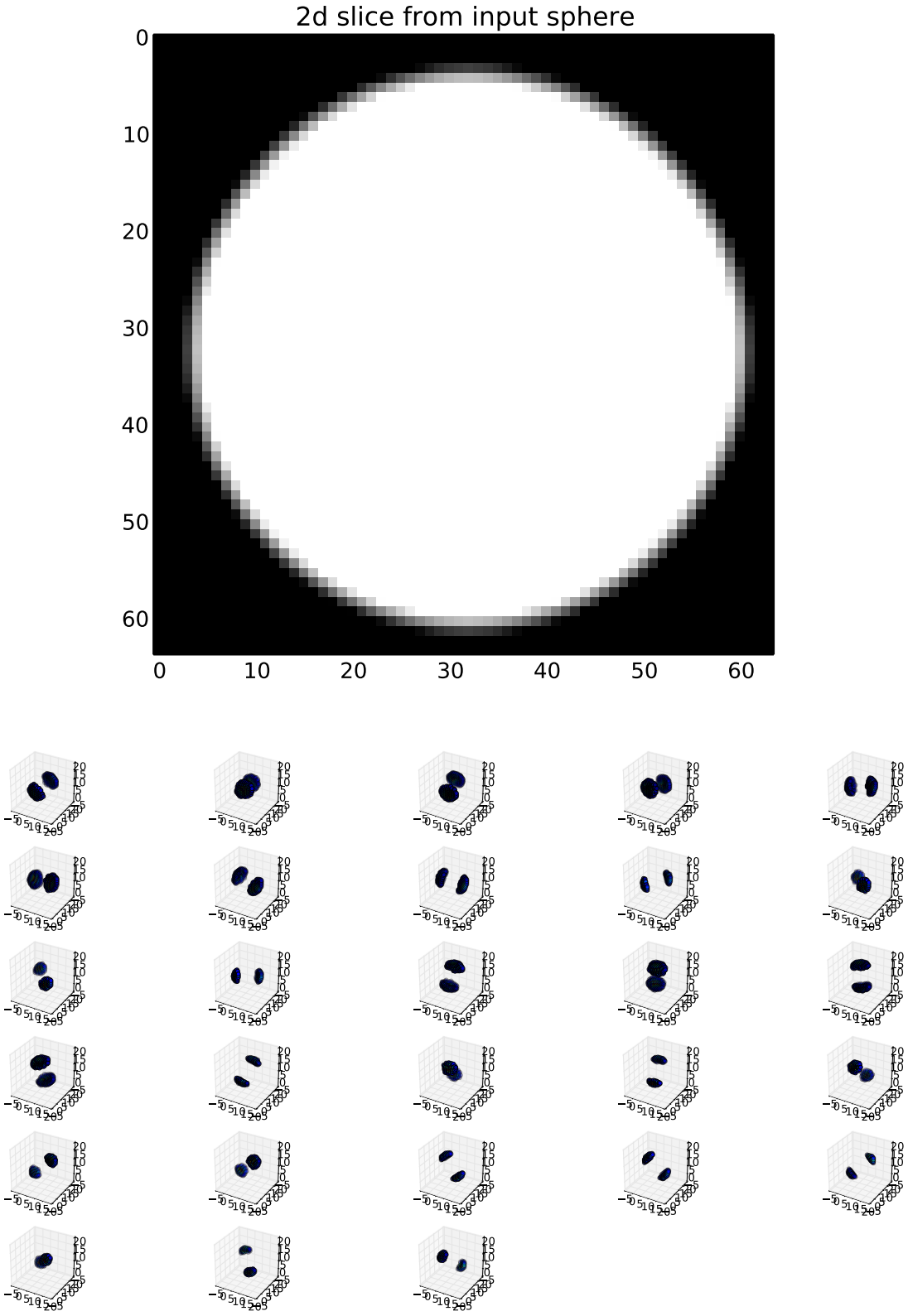
```
>>> print(sphere_t.lowpass.shape)
(16, 16, 16)
>>> for highpasses in sphere_t.highpasses:
...     print(highpasses.shape)
(32, 32, 32, 28)
(16, 16, 16, 28)
(8, 8, 8, 28)
```

Performing the inverse transform should result in perfect reconstruction

```
>>> Z = trans.inverse(sphere_t)
>>> print(np.abs(Z - sphere).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()
imshow(sphere[:,:,GRID_SIZE>>1], interpolation='none', cmap=cm.gray)
title('2d slice from input sphere')

# Plot large magnitude wavelet coefficients' position in 3D.

figure(figsize=(16,9))
Yh = sphere_t.highpasses
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 range(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*.

2.2.4 Variant transforms

In addition to the basic 1, 2 and 3 dimensional DT-CWT, this library also supports a selection of variant transforms.

Rotational symmetry modified wavelet transform

For some applications, one may prefer the subband responses to be more rotationally similar.

In the original 2-D DTCWT, the 45 and 135 degree subbands have passbands whose centre frequencies are somewhat further from the origin than those of the other four subbands. This results from the combination of two highpass 1-D wavelet filters to produce 2-D wavelets. The remaining subbands combine highpass and lowpass 1-D filters, and hence their centre frequencies are a factor of approximately sqrt(1.8) closer to the origin of the frequency plane.

The dtwavexfm2b() function employs an alternative bandpass 1-D filter in place of the highpass filter for the appropriate subbands. The image below illustrates the relevant differences in impulse and frequency responses[1].

Usage is very similar to the standard 2-D transform function, but one uses the 'near_sym_b_bp' and 'qshift_b_bp' wavelets.

```
import dtcwt
transform = dtcwt.Transform2d(biort='near_sym_bp', qshift='qshift_bp')

# .. load image and select number of levels ...

image_t = transform.foward(image, nlevels=nlevels)
```

While the Hilbert transform property of the DTCWT is preserved, perfect reconstruction is lost. However, in applications such as machine vision, where all subsequent operations on the image take place in the transform domain, this is of relatively minor importance.

For full details, refer to:

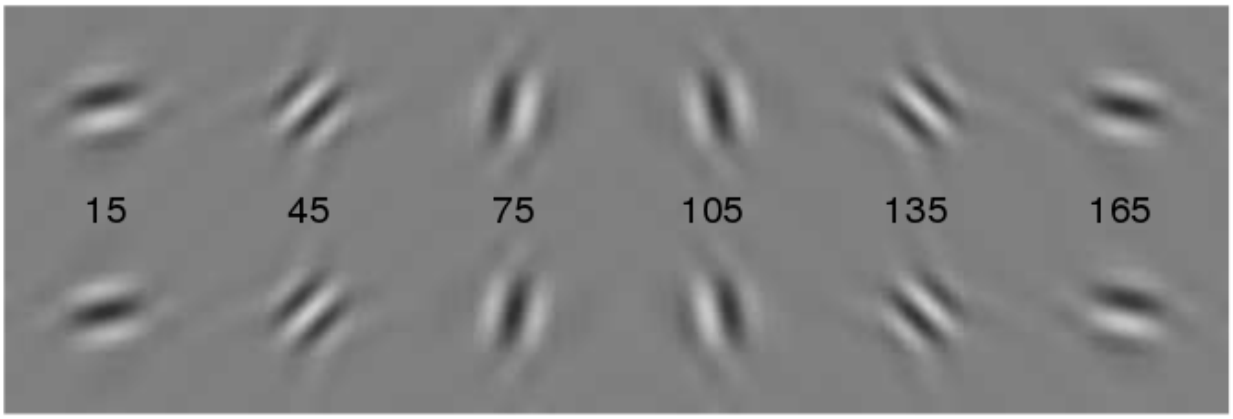
[1] N. G. Kingsbury. Rotation-invariant local feature matching with complex wavelets. *In Proc. European Conference on Signal Processing (EUSIPCO)*, pages 901–904, 2006. 2, 18, 21

Example

Working on the Lena image, the standard 2-D DTCWT achieves perfect reconstruction:

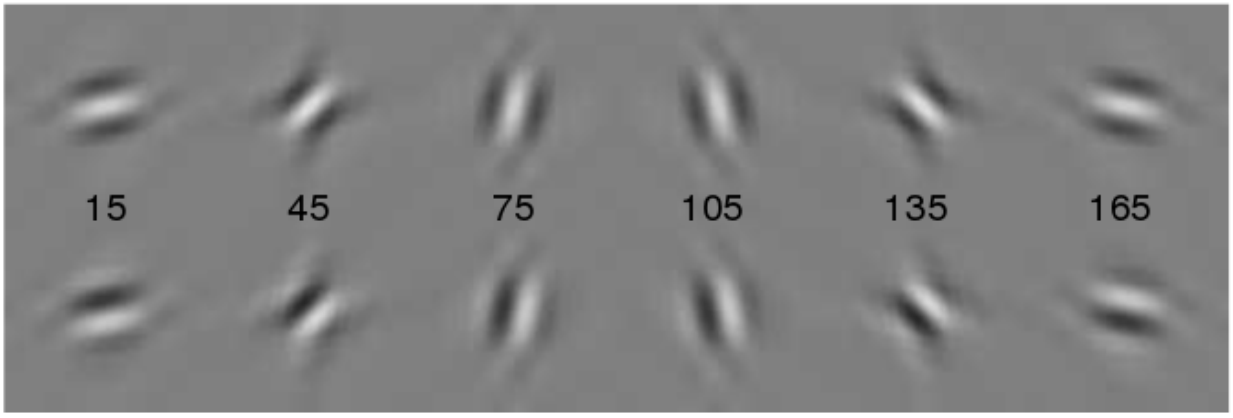
```
import dtcwt

# Use the standard 2-D DTCWT
transform = dtcwt.Transform2d(biort='near_sym_b', qshift='qshift_b')

# Forward transform
image = datasets.mandrill()
image_t = transform.forward(image)

# Inverse transform
Z = transform.inverse(image_t)

# Show the error
imshow(Z-image, cmap=cm.gray)
colorbar()
```

(a) Dual–Tree Complex Wavelets: Real Part



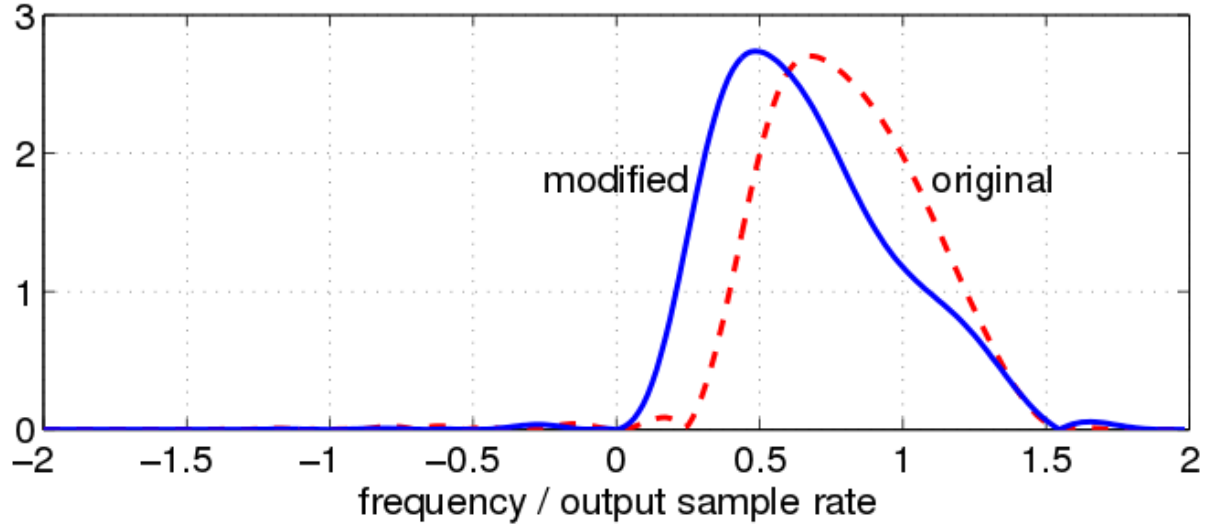
Imaginary Part

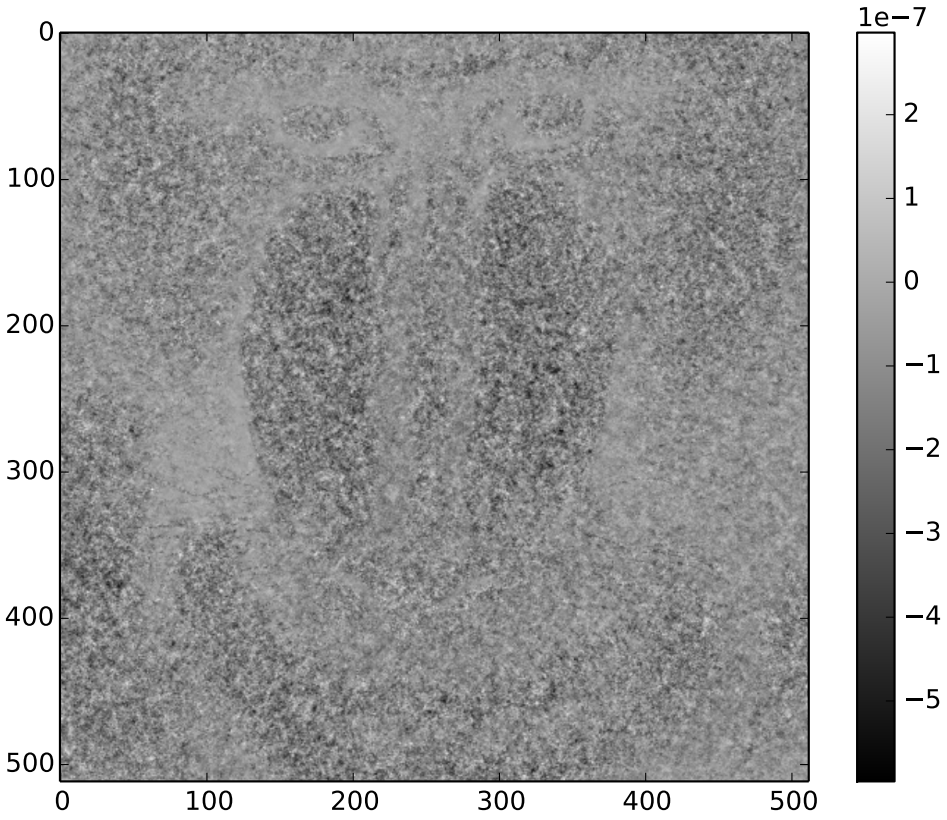
(b) Modified Complex Wavelets: Real Part



Imaginary Part

(c) Frequency responses of original and modified 1–D filters

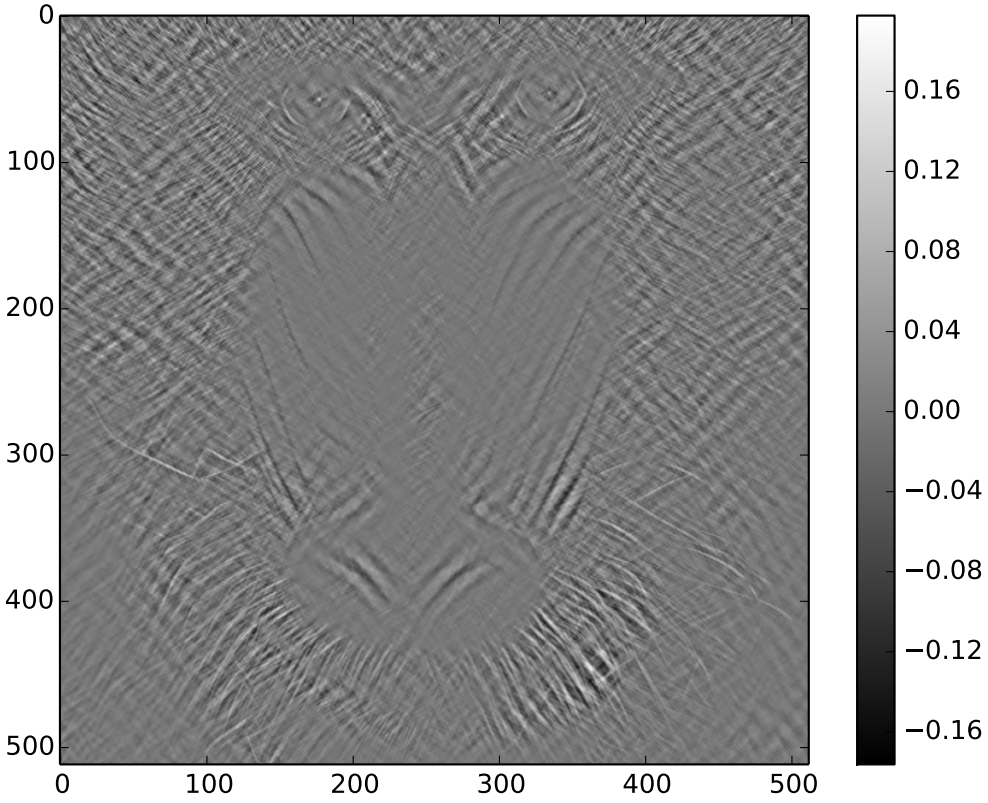




The error signal appears to be just noise, which we can attribute to floating-point precision.

Using the modified wavelets yields the following result:

```
import dtcwt

# Use the modified 2-D DTCWT
transform = dtcwt.Transform2d(biort='near_sym_b_bp', qshift='qshift_b_bp')

# Forward transform
image = datasets.mandrill()
image_t = transform.forward(image)

# Inverse transform
Z = transform.inverse(image_t)

# Show the error
imshow(Z-image, cmap=cm.gray)
colorbar()
```



As we would expect, the error is more significant, but only near 45 and 135 degree edge features.

2.3 Multiple Backend Support

The `dtcwt` library currently provides two backends for computing the wavelet transform: a NumPy based implementation and an OpenCL implementation which uses the PyOpenCL bindings for Python.

2.3.1 NumPy

The NumPy backend is the reference implementation of the transform. All algorithms and transforms will have a NumPy backend. NumPy implementations are written to be efficient but also clear in their operation.

2.3.2 OpenCL

Some transforms and algorithms implement an OpenCL backend. This backend, if present, will provide an identical API to the NumPy backend. NumPy-based input may be passed in and out of the backends but if OpenCL-based input is passed in, a copy back to the host may be avoided in some cases. Not all transforms or algorithms have an OpenCL-based implementation and the implementation itself may not be full-featured.

OpenCL support depends on the PyOpenCL package being installed and an OpenCL implementation being installed on your machine. Attempting to use an OpenCL backen without both of these being present will result in a runtime

(but not import-time) exception.

2.3.3 Which backend should I use?

The top-level transform routines, such as :py:class‘dtcwt.Transform2d‘, will automatically use the NumPy backend. If you are not primarily focussed on speed, this is the correct choice since the NumPy backend has the fullest feature support, is the best tested and behaves correctly given single- and double-precision input.

If you care about speed and need only single-precision calculations, the OpenCL backend can provide significant speed-up. On the author's system, the 2D transform sees around a times 10 speed improvement.

2.3.4 Using a backend

The NumPy and OpenCL backends live in the *dtcwt.numpy* and *dtcwt.opencl* modules respectively. Both provide implementations of some subset of the DTCWT library functionality.

Access to the 2D transform is via a *dtcwt.Transform2d* instance. For example, to compute the 2D DT-CWT of the 2D real array in X:

```
>>> from dtcwt.numpy import Transform2d
>>> trans = Transform2d()           # You may optionally specify which wavelets to use here
>>> Y = trans.forward(X, nlevels=4) # Perform a 4-level transform of X
>>> imshow(Y.lowpass)               # Show coarsest scale low-pass image
>>> imshow(Y.highpasses[-1][:,:,0])   # Show first coarsest scale subband
```

In this case Y is an instance of a class which behaves like *dtcwt.Pyramid*. Backends are free to return whatever result they like as long as the result can be used like this base class. (For example, the OpenCL backend returns a *dtcwt.opencl.Pyramid* instance which keeps the device-side results available.)

The default backend used by *dtcwt.Transform2d*, etc can be manipulated using the *dtcwt.push_backend()* function. For example, to switch to the OpenCL backend

```
dtcwt.push_backend('opencl')
# ... Transform2d, etc now use OpenCL ...
```

As is suggested by the name, changing the backend manipulates a stack behind the scenes and so one can temporarily switch backend using *dtcwt.push_backend()* and *dtcwt.pop_backend()*

```
# Run benchmark with NumPy
my_benchmarking_function()

# Run benchmark with OpenCL
dtcwt.push_backend('opencl')
my_benchmarking_function()
dtcwt.pop_backend()
```

It is safer to use the *dtcwt.preserve_backend_stack()* function. This returns a guard object which can be used with the `with` statement to save the state of the backend stack

```
with dtcwt.preserve_backend_stack():
    dtcwt.push_backend('opencl')
    my_benchmarking_function()

# Outside of the 'with' clause the backend is reset to numpy.
```

Finally the default backend may be set via the `DTCWT_BACKEND` environment variable. This is useful to run scripts with different backends without having to modify their source.

2.4 DTCWT-based algorithms

2.4.1 Image Registration

The `dtcwt.registration` module provides an implementation of a DTCWT-based image registration algorithm. The output is similar, but not identical, to "optical flow". The underlying assumption is that the source image is a smooth locally-affine warping of a reference image. This assumption is valid in some classes of medical image registration and for video sequences with small motion between frames.

Algorithm overview

This section provides a brief overview of the algorithm itself. The algorithm is a 2D version of the 3D registration algorithm presented in Efficient Registration of Nonrigid 3-D Bodies. The motion field between two images is a vector field whose elements represent the direction and distance of displacement for each pixel in the source image required to map it to a corresponding pixel in the reference image. In this algorithm the motion is described via the affine transform which can represent rotation, translation, shearing and scaling. An advantage of this model is that if the motion of two neighbouring pixels are from the same model then they will share affine transform parameters. This allows for large regions of the image to be considered as a whole and helps mitigate the aperture problem.

The model described below is based on the model in Phase-based multidimensional volume registration with changes designed to allow use of the DTCWT as a front end.

Motion constraint

The three-element homogeneous displacement vector at location $\mathbf{x}$ is defined to be

$$\tilde{\mathbf{v}}(\mathbf{x}) \equiv \begin{bmatrix} \mathbf{v}(\mathbf{x}) \\ 1 \end{bmatrix}$$

where $\mathbf{v}(\mathbf{x})$ is the motion vector at location $\mathbf{x} = [x\, y]^T$. A motion constraint is a three-element vector, $\mathbf{c}(\mathbf{x})$ such that

$$\mathbf{c}^T(\mathbf{x})\, \tilde{\mathbf{v}}(\mathbf{x}) = 0.$$

In the two-dimensional DTCWT, the phase of each complex highpass coefficient has an approximately linear relationship with the local shift vector. We can therefore write

$$\frac{\partial \theta_d}{\partial t} = \nabla_{\mathbf{x}} \theta_d \cdot \mathbf{v}(\mathbf{x})$$

where $\nabla_{\mathbf{x}} \theta_d \equiv [(\partial \theta_d / \partial x)\, (\partial \theta_d / \partial y)]^T$ and represents the phase gradient at $\mathbf{x}$ for subband d in both of the x and y directions.

Numerical estimation of the partial derivatives of θ_d can be performed by noting that multiplication of a subband pixels's complex coefficient by the conjugate of its neighbour subtracts phase whereas multiplication by the neighbour adds phase. We can thus construct equivalents of forward-, backward- and central difference algorithms for phase gradients.

Comparing the relations above, it is clear that the motion constraint vector, $\mathbf{c}_d(\mathbf{x})$, corresponding to subband d at location $\mathbf{x}$ satisfies the following:

$$\mathbf{c}_d(\mathbf{x}) = C_d(\mathbf{x}) \begin{bmatrix} \nabla_{\mathbf{x}} \theta_d \\ -\frac{\partial \theta_d}{\partial t} \end{bmatrix}$$

where $C_d(\mathbf{x})$ is some weighting factor which we can interpret as a measure of the confidence we have of subband d specifying the motion at $\mathbf{x}$.

This confidence measure can be heuristically designed. The measure used in this implementation is:

$$C_d(\mathbf{x}) = \frac{\left|\sum_{k=1}^{4} u_k^* v_k\right|^2}{\sum_{k=1}^{4}(|u_k|^3 + |v_k|^3) + \epsilon}.$$

where u_k and v_k are the wavelet coefficients in the reference and source transformed images, subscripts $k = 1 \ldots 4$ denote the four diagonally neighbouring coefficients and ϵ is some small value to avoid division by zero when the wavelet coefficients are small. It is beyond the scope of this documentation to describe the design of this metric. Refer to the original paper for more details.

Cost function

The model is represented via the six parameters $a_1 \ldots a_6$ such that

$$\mathbf{v}(\mathbf{x}) = \begin{bmatrix} 1 & 0 & x & 0 & y & 0 \\ 0 & 1 & 0 & x & 0 & y \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_6 \end{bmatrix} \equiv \mathbf{K}(\mathbf{x})\,\mathbf{a}.$$

We then make the following definitions:

$$\tilde{\mathbf{K}}(\mathbf{x}) \equiv \begin{bmatrix} \mathbf{K}(\mathbf{x}) & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix}, \quad \tilde{\mathbf{a}} \equiv \begin{bmatrix} \mathbf{a} \\ 1 \end{bmatrix}$$

and then the homogenous motion vector is given by

$$\tilde{\mathbf{v}}(\mathbf{x}) = \tilde{\mathbf{K}}(\mathbf{x})\,\tilde{\mathbf{a}}.$$

Considering all size subband directions, we have:

$$\mathbf{c}_d(\mathbf{x})\,\tilde{\mathbf{K}}(\mathbf{x})\,\tilde{\mathbf{a}} = 0, \quad \forall\, d \in \{1, \ldots, 6\}\,.$$

Each location $\mathbf{x}$ has six constraint equations for six unknown affine parameters in $\tilde{\mathbf{a}}$. We can solve for $\tilde{\mathbf{a}}$ by minimising squared error $\epsilon(\mathbf{x})$:

$$\begin{aligned} \epsilon(\mathbf{x}) &= \sum_{d=1}^{6} \left\| \mathbf{c}_d^T(\mathbf{x}) \tilde{\mathbf{K}}(\mathbf{x}) \tilde{\mathbf{a}} \right\|^2 \\ &= \sum_{d=1}^{6} \tilde{\mathbf{a}}^T\, \tilde{\mathbf{K}}^T(\mathbf{x})\, \mathbf{c}_d(\mathbf{x}) \mathbf{c}_d^T(\mathbf{x})\, \tilde{\mathbf{K}}(\mathbf{x})\, \tilde{\mathbf{a}} \\ &= \tilde{\mathbf{a}}^T \tilde{\mathbf{Q}}(\mathbf{x}) \tilde{\mathbf{a}} \end{aligned}$$

where

$$\tilde{\mathbf{Q}}(\mathbf{x}) \equiv \sum_{d=1}^{6} \tilde{\mathbf{K}}^T(\mathbf{x})\, \mathbf{c}_d(\mathbf{x}) \mathbf{c}_d^T(\mathbf{x})\, \tilde{\mathbf{K}}(\mathbf{x}).$$

In practice, in order to handle the registration of dissimilar image features and also to handle the aperture problem, it is helpful to combine $\tilde{\mathbf{Q}}(\mathbf{x})$ matrices across more than one level of DTCWT and over a slightly wider area within each level. This results in better estimates of the affine parameters and reduces the likelihood of obtaining singular matrices. We define locality χ to represent this wider spatial and inter-scale region, such that

$$\tilde{\mathbf{Q}}_\chi = \sum_{\mathbf{x} \in \chi} \tilde{\mathbf{Q}}(\mathbf{x}).$$

The $\tilde{\mathbf{Q}}_\chi$ matrices are symmetric and so can be written in the following form:

$$\tilde{\mathbf{Q}}_\chi = \begin{bmatrix} \mathbf{Q}_\chi & \mathbf{q}_\chi \\ \mathbf{q}_\chi^T & q_{0,\chi} \end{bmatrix}$$

where $\mathbf{q}_\chi$ is a six-element vector and $q_{0,\chi}$ is a scalar. Substituting into our squared error function gives

$$\epsilon_\chi = \mathbf{a}^T \mathbf{Q}_\chi \mathbf{a} + 2\mathbf{a}^T \mathbf{q}_\chi + q_{0,\chi}.$$

To minimize, we differentiate and set to zero. Hence,

$$\nabla_{\mathbf{a}} \epsilon_\chi = 2\mathbf{Q}_\chi \mathbf{a} + 2\mathbf{q}_\chi = 0$$

and so the local affine parameter vector satisfies

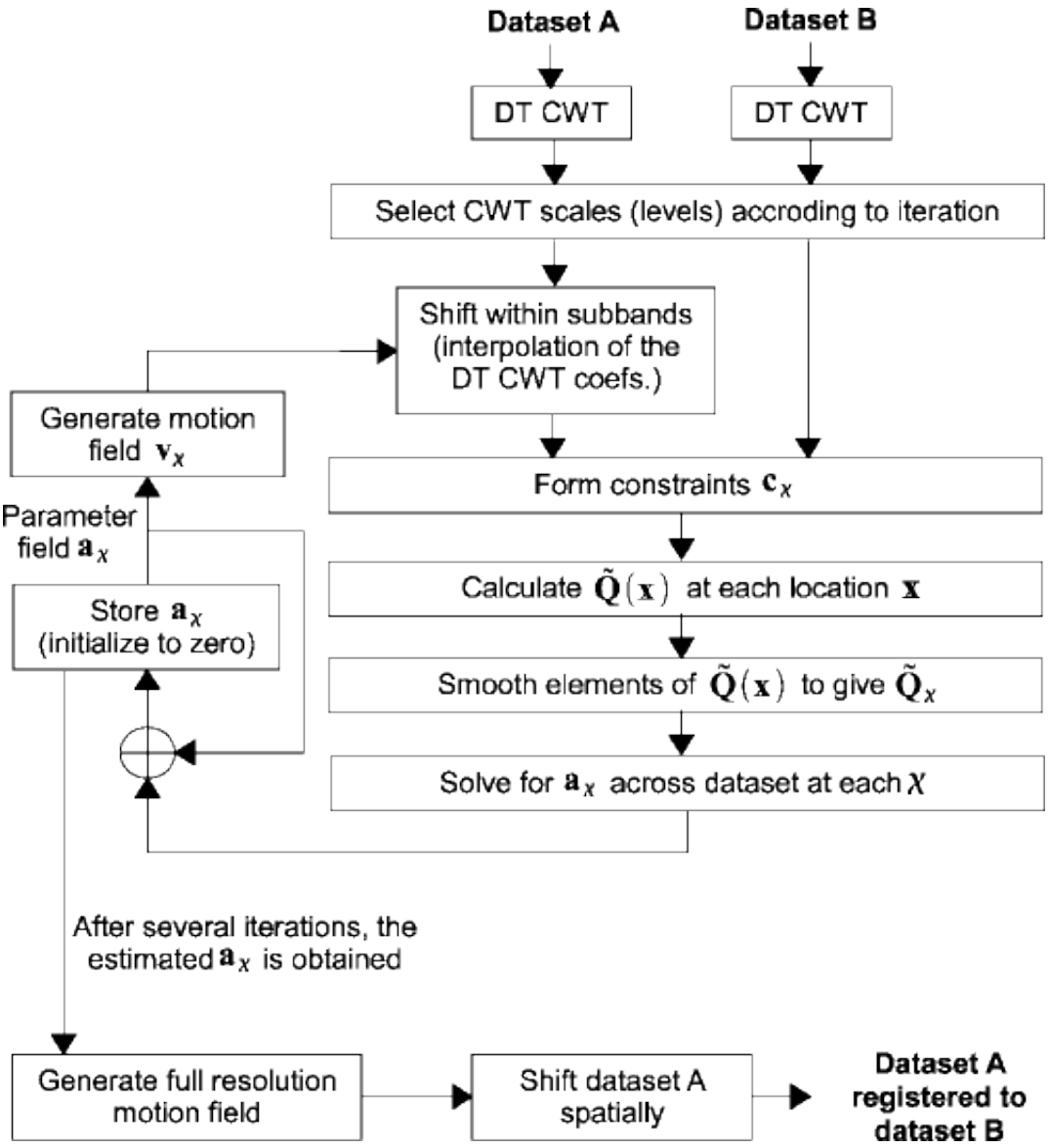
$$\mathbf{Q}_\chi \mathbf{a}_\chi = -\mathbf{q}_\chi.$$

In our implementation, we avoid calculating the inverse of $\mathbf{Q}_\chi$ directly and solve this equation by eigenvector decomposition.

Iteration

There are three stres in the full registration algorithm: transform the images to the DTCWT domain, perform motion estimation and register the source image. We do this via an iterative process where coarse-scale estimates of $\mathbf{a}_\chi$ are estimated from coarse-scale levels of the transform and progressively refined with finer-scale levels.

The following flow diagram, taken from the paper, illustrates the algorithm.



The pair of images to be registered are first transformed by the DTCWT and levels to be used for motion estimation are selected. The subband coefficients of the source image are shifted according to the current motion field estimate. These shifted coefficients together with those of the reference image are then used to generate motion constraints. From these the $\tilde{\mathbf{Q}}_\chi$ matrices are calculated and the local affine distortion parameters updated. After a few iterations, the distortion parameters are used to warp the source image directly.

Using the implementation

The implementation of the image registration algorithm is accessed via the *dtcwt.registration* module's functions. The two functions of most interest at *dtcwt.registration.estimatereg()* and *dtcwt.registration.warp()*. The former will estimate $\mathbf{a}_\chi$ for each 8x8 block in the image and *dtcwt.registration.warp()* will take these affine parameter vectors and warp an image with them.

As an example, we will register two frames from a video of road traffic. Firstly, as boilerplate, import plotting command from pylab and also the `datasets` module which is part of the test suite for *dtcwt*.

```
from pylab import *
import datasets
```

If we show one image in the red channel and one in the green, we can see where the images are incorrectly registered by looking for red or green fringes:

```
ref, src = datasets.regframes('traffic')

figure()
imshow(np.dstack((ref, src, np.zeros_like(ref))))
title('Registration input images')
```

To register the images we first take the DTCWT:

```
import dtcwt

transform = dtcwt.Transform2d()
ref_t = transform.forward(ref, nlevels=6)
src_t = transform.forward(src, nlevels=6)
```

Registration is now performed via the *dtcwt.registration.estimatereg()* function. Once the registration is estimated, we can warp the source image to the reference using the *dtcwt.registration.warp()* function.

```
import dtcwt.registration as registration

reg = registration.estimatereg(src_t, ref_t)
warped_src = registration.warp(src, reg, method='bilinear')
```

Plotting the warped and reference image in the green and red channels again shows a marked reduction in colour fringes.

```
figure()

imshow(np.dstack((ref, warped_src, np.zeros_like(ref))))
title('Source image warped to reference')
```

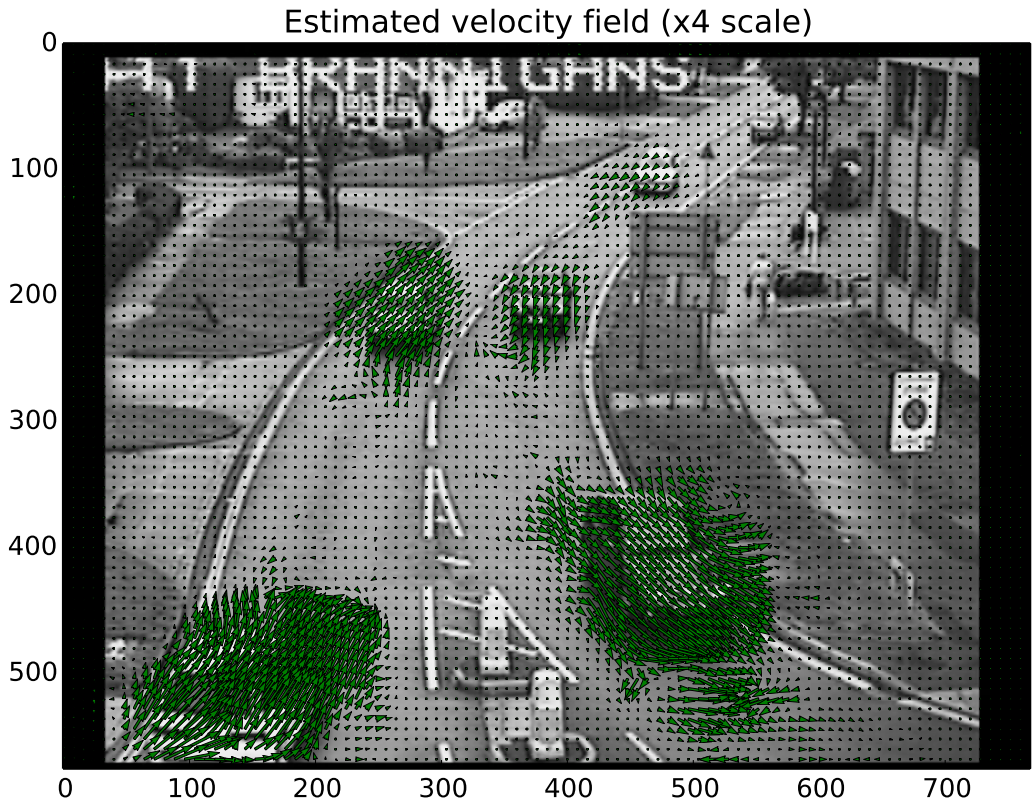
The velocity field, in units of image width/height, can be calculated by the *dtcwt.registration.velocityfield()* function. We need to scale the result by the image width and height to get a velocity field in pixels.

```
vxs, vys = registration.velocityfield(reg, ref.shape[:2], method='bilinear')
vxs = vxs * ref.shape[1]
vys = vys * ref.shape[0]
```

We can plot the result as a quiver map overlaid on the reference image:

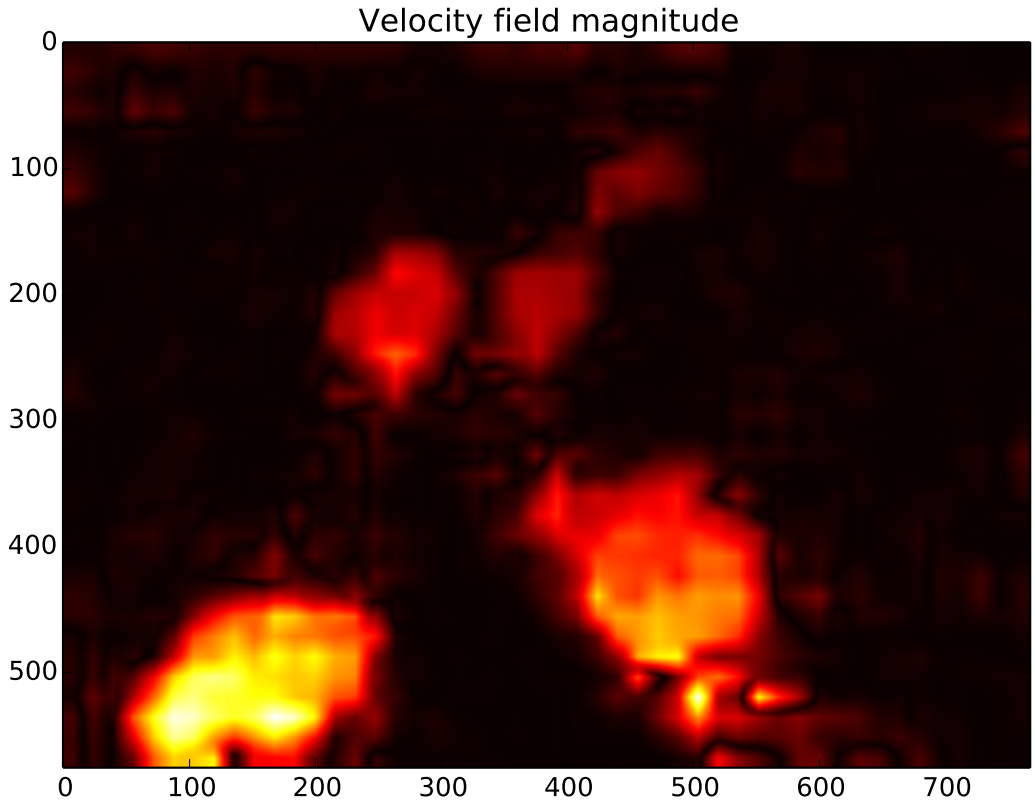
```
figure()

X, Y = np.meshgrid(np.arange(ref.shape[1]), np.arange(ref.shape[0]))

imshow(ref, cmap=cm.gray, clim=(0,1))

step = 8

quiver(X[::step,::step], Y[::step,::step],
```

```
        vxs[::step,::step], vys[::step,::step],
        color='g', angles='xy', scale_units='xy', scale=0.25)

title('Estimated velocity field (x4 scale)')
```



We can also plot the magnitude of the velocity field which clearly shows the moving cars:

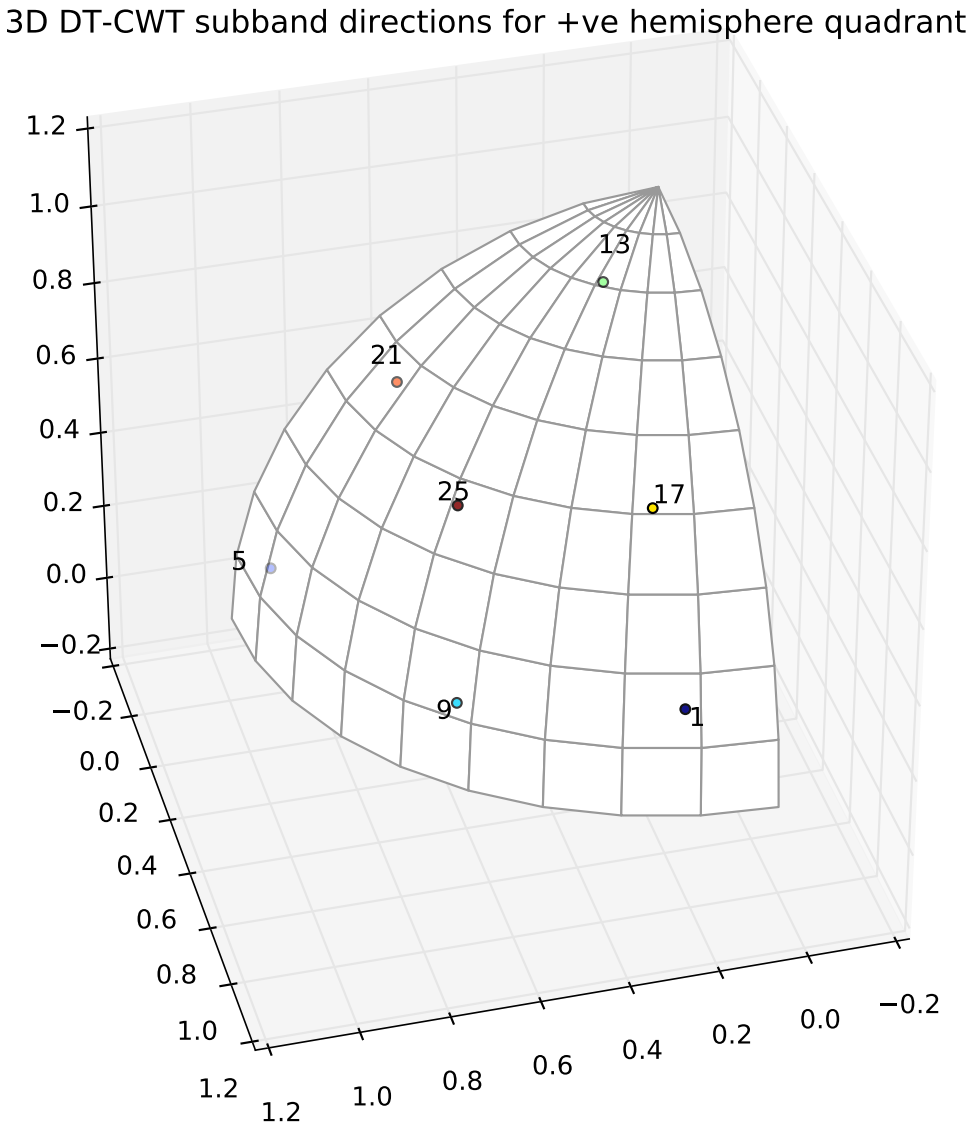
```
figure()
imshow(np.abs(vxs + 1j*vys), cmap=cm.hot)
title('Velocity field magnitude')
```



2.5 Example scripts

2.5.1 Showing 3D Directional Sensitivity

The `3d_dtcwt_directionality.py` script in the `docs/` 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:



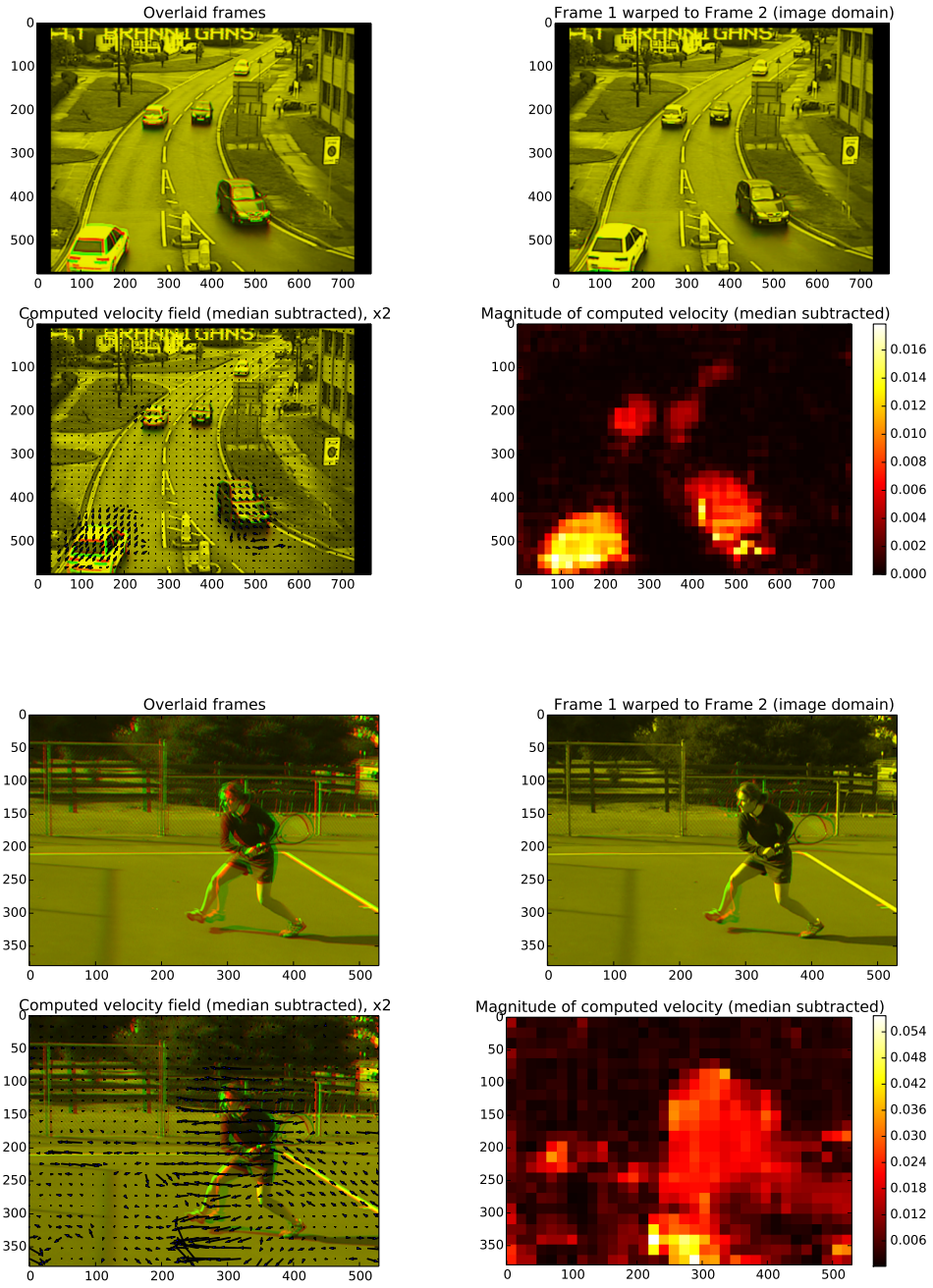
There are some points to note about this diagram. Each subband is labeled sich that '1' refers to the first subband, '5' the fifth and so forth. On this diagram the highpasses are all four apart reflecting the fact that, for example, highpasses 2, 3 and 4 are positioned in the other four quadrants of 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.

2.5.2 2D Image Registration

This library includes support for 2D image registration modelled after the 3D algorithm outlined in the paper Efficient Registration of Nonrigid 3-D Bodies. The `image-registration.py` script in the `docs/` directory shows a complete worked example of using the registration API using two sets of source images: a woman playing tennis and some traffic at a road junction.

It will attempt to register two image pairs: a challenging sequence from a video sequence and a sequence from a traffic

camera. The result is shown below.



2.6 API Reference

2.6.1 Main interface

class dtcwt.**Transform1d**(*biort='near_sym_a'*, *qshift='qshift_a'*)

An implementation of the 1D DT-CWT in NumPy.

Parameters

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

forward(*X*, *nlevels=3*, *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

Returns A *dtcwt.Pyramid*-like object representing the transform result.

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).

inverse(*pyramid*, *gain_mask=None*)

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

Parameters

- **pyramid** – A *dtcwt.Pyramid*-like object containing the transformed signal.
- **gain_mask** – Gain to be applied to each subband.

Returns 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.

class `dtcwt.`**Transform2d**(*biort='near_sym_a'*, *qshift='qshift_a'*)

An implementation of the 2D DT-CWT via NumPy. *biort* and *qshift* are the wavelets which parameterise the transform.

If *biort* or *qshift* are strings, they are used as an argument to the *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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).

forward(*X*, *nlevels=3*, *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

Returns A *dtcwt.Pyramid* compatible object representing the transform-domain signal

inverse(*pyramid*, *gain_mask=None*)

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

Parameters

- **pyramid** – A *dtcwt.Pyramid*-like class holding the transform domain representation to invert.
- **gain_mask** – Gain to be applied to each subband.

Returns A numpy-array compatible instance with the reconstruction.

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.

class dtcwt.**Transform3d**(*biort='near_sym_a'*, *qshift='qshift_a'*, *ext_mode=4*)

An implementation of the 3D DT-CWT via NumPy. *biort* and *qshift* are the wavelets which parameterise the transform. Valid values are documented in *dtcwt.coeffs.biort()* and *dtcwt.coeffs.qshift()*.

forward(*X*, *nlevels=3*, *include_scale=False*, *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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.qshift()*.
- **discard_level_1** – True if level 1 high-pass bands are to be discarded.

Returns a *dtcwt.Pyramid* instance

Each element of the Pyramid *highpasses* tuple is a 4D complex array with the 4th dimension having size 28. The 3D slice `[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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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 not 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 the first element of the *highpasses* tuple will be *None*. Note that *dtcwt.Transform3d.inverse()* will accept the first element being *None* and will treat it as being zero.

inverse(*pyramid*)

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

Parameters

- **pyramid** – The *dtcwt.Pyramid*-like instance representing the transformed signal.
- **biort** – Level 1 wavelets to use. See `biort()`.
- **qshift** – Level >= 2 wavelets to use. See `qshift()`.
- **ext_mode** – Extension mode. See below.

Returns Reconstructed real image matrix.

If *biort* or *qshift* are strings, they are used as an argument to the *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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.

class `dtcwt.`**`Pyramid`**(*lowpass*, *highpasses*, *scales=None*)

A representation of a transform domain signal.

Backends are free to implement any class which respects this interface for storing transform-domain signals. The inverse transform may accept a backend-specific version of this class but should always accept any class which corresponds to this interface.

`lowpass`

A NumPy-compatible array containing the coarsest scale lowpass signal.

`highpasses`

A tuple where each element is the complex subband coefficients for corresponding scales finest to coarsest.

`scales`

(optional) A tuple where each element is a NumPy-compatible array containing the lowpass signal for corresponding scales finest to coarsest. This is not required for the inverse and may be *None*.

`dtcwt.`**`backend_name = 'numpy'`**

A string providing a short human-readable name for the DTCWT backend currently being used. This corresponds to the *name* parameter passed to *`dtcwt.push_backend()`*. The *default* backend is `numpy` but can be overridden by setting the DTCWT_BACKEND environment variable to a valid backend name.

`dtcwt.`**`push_backend`**(*name*)

Switch backend implementation to *name*. Push the previous backend onto the backend stack. The previous backend may be restored via *`dtcwt.pop_backend()`*.

Parameters **`name`** – string identifying which backend to switch to

Raises **`ValueError`** – if *name* does not correspond to a known backend

name may take one of the following values:

- `numpy`: the default NumPy backend. See *`dtcwt.numpy`*.
- `opencl`: a backend which uses OpenCL where available. See *`dtcwt.opencl`*.

`dtcwt.`**`pop_backend`**()

Restore the backend after a call to *`push_backend()`*. Calls to *`pop_backend()`* and *`pop_backend()`* may be nested. This function will undo the most recent call to *`push_backend()`*.

Raises **`IndexError`** – if one attempts to pop more backends than one has pushed.

`dtcwt.`**`preserve_backend_stack`**()

Return a generator object which can be used to preserve the backend stack even when an exception has been raise. For example:

```
# current backend is NumPy
assert dtcwt.backend_name == 'numpy'

with dtcwt.preserve_backend_stack():
    dtcwt.push_backend('opencl')
    # ... things which may raise an exception
```

```
    # current backend is NumPy even if an exception was thrown
    assert dtcwt.backend_name == 'numpy'
```

Functions to load standard wavelet coefficients.

dtcwt.coeffs.**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.
near_sym_b_bp	Near-Symmetric 13,19 tap filters + BP filter

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

See *Rotational symmetry modified wavelet transform* for an explanation of the `near_sym_b_bp` wavelet filters.

Raises

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

dtcwt.coeffs.**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.
qshift_b_bp	Q-Shift 18,18 tap filters + BP

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

See *Rotational symmetry modified wavelet transform* for an explanation of the `qshift_b_bp` wavelet filters.

Raises

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

2.6.2 Keypoint analysis

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

Parameters

- **highpass_highpasses** – (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_highpasses* 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 highpasses	*kappa*

[1] Julien Fauqueur, Nick Kingsbury, and Ryan Anderson. *Multiscale Keypoint Detection using the Dual-Tree Complex Wavelet Transform*. 2006 International Conference on Image Pro-

cessing, 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.

2.6.3 Image sampling

This module contains function for rescaling and re-sampling high- and low-pass highpasses.

Note: All of these functions take an integer co-ordinate (x, y) to be the *centre* of the corresponding pixel. Therefore the upper-left pixel notionally covers the interval (-0.5, 0.5) in x and y. An image with N rows and M columns, therefore, has an extent (-0.5, M-0.5) on the x-axis and an extent of (-0.5, N-0.5) on the y-axis. The rescale and upsample functions in this module will use this region as the extent of the image.

`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*, *sbs=None*)

As *sample()* except that the highpass image is first phase shifted to be centred on approximately DC, and has additional 'sbs' input allowing selection and re-ordering of subbands. This is useful mainly when the exact locations one wishes to sample from vary by subband.

'sbs' should ordinarily be a numpy array of subband indices, in ascending order, e.g., np.array([0,2,3,5]) for just subbands 0, 2, 3 and 5; The returned array will be flattened to just 4 subbands. Pass [0,1,2,3,4,5] for all subbands.

Take care when re-ordering, preferably keeping the 'sbs' array outside the scope of this function to keep track of the new indices.

19. (a)Forshaw, Feb 2014.

`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*, *sbs=None*)

As *rescale()* except that the highpass image is first phase shifted to be centred on approximately DC, and has additional 'sbs' input allowing selection and re-ordering of subbands. This is useful mainly when the exact locations one wishes to sample from vary by subband.

'sbs' should ordinarily be a list of subband indices, in ascending order, e.g., np.array([0,2,3,5]) for just subbands 0, 2, 3 and 5; The returned array will be flattened to just 4 subbands. Pass [0,1,2,3,4,5] for all subbands.

Take care when re-ordering, preferably keeping the 'sbs' array outside the scope of this function to keep track of the new indices.

19. (a)Forshaw, Feb 2014.

`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.

2.6.4 Image registration

Note: This module is experimental. It's API may change between versions.

This module implements function for DTCWT-based image registration as outlined in [1]. These functions are 2D-only for the moment.

`dtcwt.registration.`**`estimatereg`**(*source*, *reference*, *regshape=None*, *levels=None*)

Estimate registration from which will map *source* to *reference*.

Parameters

- **`source`** – transformed source image
- **`reference`** – transformed reference image

The *reference* and *source* parameters should support the same API as *dtcwt.Pyramid*.

The local affine distortion is estimated at at 8x8 pixel scales. Return a NxMx6 array where the 6-element vector at (N,M) corresponds to the affine distortion parameters for the 8x8 block with index (N,M).

Use the *velocityfield()* function to convert the return value from this function into a velocity field.

If not-*None*, *levels* is a sequence of sequences of 0-based level indices to use when calculating the registration. If *None* then a default set of levels are used.

`dtcwt.registration.`**`velocityfield`**(*avecs*, *shape*, *method=None*)

Given the affine distortion parameters returned from *estimatereg()*, return a tuple of 2D arrays giving the

x- and y- components of the velocity field. The shape of the velocity component field is *shape*. The velocities are measured in terms of normalised units where the image has width and height of unity.

The *method* parameter is interpreted as in `dtcwt.sampling.rescale()` and is the sampling method used to resize *avecs* to *shape*.

`dtcwt.registration.`**`warp`**(*I*, *avecs*, *method=None*)
A convenience function to warp an image according to the velocity field implied by *avecs*.

`dtcwt.registration.`**`warptransform`**(*t*, *avecs*, *levels*, *method=None*)
Return a warped version of a transformed image acting only on specified levels.

Parameters

- **`t`** – a transformed image
- **`avecs`** – an array of affine distortion parameters
- **`levels`** – a sequence of 0-based indices specifying which levels to act on

t should be a `dtcwt.Pyramid`-compatible instance.

The *method* parameter is interpreted as in `dtcwt.sampling.rescale()` and is the sampling method used to resize *avecs* to *shape*.

Note: This function will clone the transform *t* but it is a shallow clone where possible. Only the levels specified in *levels* will be deep-copied and warped.

2.6.5 Plotting functions

Convenience functions for plotting DTCWT-related objects.

`dtcwt.plotting.`**`overlay_quiver`**(*image*, *vectorField*, *level*, *offset*)
Overlays nicely coloured quiver plot of complex coefficients over original full-size image, providing a useful phase visualisation.

Parameters

- **`image`** – array holding grayscale values on the interval [0, 255] to display
- **`vectorField`** – a single [MxNx6] numpy array of DTCWT coefficients
- **`level`** – the transform level (1-indexed) of *vectorField*.
- **`offset`** – Offset for DTCWT coefficients (typically 0.5)

Note: The *level* parameter is 1-indexed meaning that the third level has index "3". This is unusual in Python but is kept for compatibility with similar MATLAB routines.

Should also work with other types of complex arrays (e.g., SLP coefficients), as long as the format is the same.

Usage example:

import dtcwt import dtcwt.plotting as plotting

mandrill = datasets.mandrill()

transform2d = dtcwt.Transform2d() mandrill_t = transform2d.forward(mandrill, nlevels=5)

plotting.overlay_quiver(mandrill*255, mandrill_t.highpasses[-1], 5, 0.5)

2.6.6 Miscellaneous and low-level support functions

Useful utilities for testing the 2-D DTCWT with synthetic images

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

dtcwt.utils.**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.utils.**drawcirc**(*r*, *w*, *du*, *dv*, *N*)
Generate an image of size N*N pels, containing a circle radius r pels and centred at du,dv relative to the centre of the image. The edge of the circle is a cosine shaped edge of width w (from 10 to 90% points).

Python implementation by S. C. Forshaw, November 2013.

dtcwt.utils.**drawedge**(*theta*, *r*, *w*, *N*)
Generate an image of size N * N pels, of an edge going from 0 to 1 in height at theta degrees to the horizontal (top of image = 1 if angle = 0). r is a two-element vector, it is a coordinate in ij coords through which the step should pass. The shape of the intensity step is half a raised cosine w pels wide (w>=1).

T. E . Gale's enhancement to drawedge() for MATLAB, transliterated to Python by S. C. Forshaw, Nov. 2013.

dtcwt.utils.**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.

dtcwt.utils.**stacked_2d_matrix_matrix_prod**(*mats1*, *mats2*)
Interpret *mats1* and *mats2* as arrays of 2D matrices. I.e. *mats1* has shape PxQxNxM and *mats2* has shape PxQxMxR. The result is a PxQxNxR array equivalent to:

```
result[i,j,:,:] = mats1[i,j,:,:].dot(mats2[i,j,:,:])
```

for all valid row and column indices *i* and *j*.

dtcwt.utils.**stacked_2d_matrix_vector_prod**(*mats*, *vecs*)
Interpret *mats* and *vecs* as arrays of 2D matrices and vectors. I.e. *mats* has shape PxQxNxM and *vecs* has shape PxQxM. The result is a PxQxN array equivalent to:

```
result[i,j,:] = mats[i,j,:,:].dot(vecs[i,j,:])
```

for all valid row and column indices *i* and *j*.

dtcwt.utils.**stacked_2d_vector_matrix_prod**(*vecs*, *mats*)
Interpret *mats* and *vecs* as arrays of 2D matrices and vectors. I.e. *mats* has shape PxQxNxM and *vecs* has shape PxQxN. The result is a PxQxM array equivalent to:

```
result[i,j,:] = mats[i,j,:,:].T.dot(vecs[i,j,:])
```

for all valid row and column indices *i* and *j*.

2.6.7 Compatibility with MATLAB

Functions for compatibility with MATLAB scripts. These functions are intentionally similar in name and behaviour to the original functions from the DTCWT MATLAB toolbox. They are included in the library to ease the porting of MATLAB scripts but shouldn't be used in new projects.

Note: The functionality of `dtwavexfm2b` and `dtwaveifm2b` has been folded into `dtwavexfm2` and `dtwaveifm2`. For convenience of porting MATLAB scripts, the original function names are available in the *dtcwt* module as aliases but they should not be used in new code.

---

dtcwt.compat.**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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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.compat.**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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.qshift()* functions. Otherwise, they are interpreted as tuples of vectors giving filter coef-

ficients. 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.compat.**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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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.compat.**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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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.compat.**dtwavexfm2b**(*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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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.compat.**dtwaveifm2b**(*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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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.compat.**dtwavexfm3**(*X*, *nlevels=3*, *biort='near_sym_a'*, *qshift='qshift_a'*, *include_scale=False*, *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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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.compat.**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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.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 `dtcwt.coeffs.biort()` or `dtcwt.coeffs.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')
```

2.6.8 Backends

The following modules provide backend-specific implementations. Usually you won't need to import these modules directly; the main API will use an appropriate implementation. Occasionally, however, you may want to benchmark one implementation against the other.

NumPy

A backend which uses NumPy to perform the filtering. This backend should always be available.

class `dtcwt.numpy.`**Pyramid**(*lowpass*, *highpasses*, *scales=None*)

A representation of a transform domain signal.

Backends are free to implement any class which respects this interface for storing transform-domain signals. The inverse transform may accept a backend-specific version of this class but should always accept any class which corresponds to this interface.

lowpass
A NumPy-compatible array containing the coarsest scale lowpass signal.

highpasses
A tuple where each element is the complex subband coefficients for corresponding scales finest to coarsest.

scales
(optional) A tuple where each element is a NumPy-compatible array containing the lowpass signal for corresponding scales finest to coarsest. This is not required for the inverse and may be *None*.

class `dtcwt.numpy.`**Transform1d**(*biort='near_sym_a'*, *qshift='qshift_a'*)

An implementation of the 1D DT-CWT in NumPy.

Parameters

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

forward(*X*, *nlevels=3*, *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

Returns A *dtcwt.Pyramid*-like object representing the transform result.

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).

inverse(*pyramid*, *gain_mask=None*)

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

Parameters

- **pyramid** – A *dtcwt.Pyramid*-like object containing the transformed signal.
- **gain_mask** – Gain to be applied to each subband.

Returns 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.

class dtcwt.numpy.**Transform2d**(*biort='near_sym_a'*, *qshift='qshift_a'*)

An implementation of the 2D DT-CWT via NumPy. *biort* and *qshift* are the wavelets which parameterise the transform.

If *biort* or *qshift* are strings, they are used as an argument to the *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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).

forward(*X*, *nlevels=3*, *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

Returns A *dtcwt.Pyramid* compatible object representing the transform-domain signal

inverse(*pyramid*, *gain_mask=None*)

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

Parameters

- **pyramid** – A *dtcwt.Pyramid*-like class holding the transform domain representation to invert.
- **gain_mask** – Gain to be applied to each subband.

Returns A numpy-array compatible instance with the reconstruction.

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.

class dtcwt.numpy.**Transform3d**(*biort='near_sym_a'*, *qshift='qshift_a'*, *ext_mode=4*)

An implementation of the 3D DT-CWT via NumPy. *biort* and *qshift* are the wavelets which parameterise the transform. Valid values are documented in *dtcwt.coeffs.biort()* and *dtcwt.coeffs.qshift()*.

forward(*X*, *nlevels=3*, *include_scale=False*, *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 *dtcwt.coeffs.biort()*.
- **qshift** – Level >= 2 wavelets to use. See *dtcwt.coeffs.qshift()*.
- **discard_level_1** – True if level 1 high-pass bands are to be discarded.

Returns a *dtcwt.Pyramid* instance

Each element of the Pyramid *highpasses* tuple is a 4D complex array with the 4th dimension having size 28. The 3D slice `[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 *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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 not 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 the first element of the *highpasses* tuple will be *None*. Note that *dtcwt.Transform3d.inverse()* will accept the first element being *None* and will treat it as being zero.

inverse(*pyramid*)

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

Parameters

- **pyramid** – The *dtcwt.Pyramid*-like instance representing the transformed signal.
- **biort** – Level 1 wavelets to use. See `biort()`.
- **qshift** – Level >= 2 wavelets to use. See `qshift()`.
- **ext_mode** – Extension mode. See below.

Returns Reconstructed real image matrix.

If *biort* or *qshift* are strings, they are used as an argument to the *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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.

`dtcwt.numpy.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.numpy.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.numpy.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.

OpenCL

Provide low-level OpenCL accelerated operations. This backend requires that PyOpenCL be installed.

class dtcwt.opencl.**Pyramid**(*lowpass*, *highpasses*, *scales=None*)

An interface-compatible version of *dtcwt.Pyramid* where the initialiser arguments are assumed to by pyopencl.array.Array instances.

The attributes defined in *dtcwt.Pyramid* are implemented via properties. The original OpenCL arrays may be accessed via the cl_... attributes.

Note: The copy from device to host is performed *once* and then memoized. This makes repeated access to the host-side attributes efficient but will mean that any changes to the device-side arrays will not be reflected in the host-side attributes after their first access. You should not be modifying the arrays once you return an instance of this class anyway but if you do, beware!

cl_lowpass
The CL array containing the lowpass image.

cl_highpasses
A tuple of CL arrays containing the subband images.

cl_scales
(optional) Either `None` or a tuple of lowpass images for each scale.

class `dtcwt.opencl.`**Transform2d**(*biort='near_sym_a'*, *qshift='qshift_a'*, *queue=None*)
An implementation of the 2D DT-CWT via OpenCL. *biort* and *qshift* are the wavelets which parameterise the transform.

If *queue* is non-*None* it is an instance of `pyopencl.CommandQueue` which is used to compile and execute the OpenCL kernels which implement the transform. If it is *None*, the first available compute device is used.

If *biort* or *qshift* are strings, they are used as an argument to the *dtcwt.coeffs.biort()* or *dtcwt.coeffs.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).

Note: At the moment *only* the **forward** transform is accelerated. The inverse transform uses the NumPy backend.

forward(*X*, *nlevels=3*, *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

Returns A *dtcwt.Pyramid* compatible object representing the transform-domain signal

Note: *X* may be a `pyopencl.array.Array` instance which has already been copied to the device. In which case, it must be 2D. (I.e. a vector will not be auto-promoted.)

inverse(*pyramid*, *gain_mask=None*)
Perform an *n*-level dual-tree complex wavelet (DTCWT) 2D reconstruction.

Parameters

- **pyramid** – A *dtcwt.Pyramid*-like class holding the transform domain representation to invert.
- **gain_mask** – Gain to be applied to each subband.

Returns A numpy-array compatible instance with the reconstruction.

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.

dtcwt.opencl.lowlevel.**axis_convolve**(*X*, *h*, *axis=0*, *queue=None*, *output=None*)

Filter along an of *X* using filter vector *h*. If *h* has odd length, each output sample is aligned with each input sample and *Y* is the same size as *X*. If *h* has even length, each output sample is aligned with the mid point of each pair of input samples, and the output matrix's shape is increased by one along the convolution axis.

After convolution, the `pyopencl.array.Array` instance holding the device-side output is returned. This may be accessed on the host via `to_array()`.

The axis of convolution is specified by *axis*. The default direction of convolution is column-wise.

If *queue* is non-`None`, it should be a `pyopencl.CommandQueue` instance which is used to perform the computation. If `None`, a default global queue is used.

If *output* is non-`None`, it should be a `pyopencl.array.Array` instance which the result is written into. If `None`, an output array is created.

dtcwt.opencl.lowlevel.**coldfilt**(*X*, *ha*, *hb*, *queue=None*)

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.opencl.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].

The filtering will be accelerated via OpenCL.

Parameters

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

Returns Y the filtered image.

dtcwt.opencl.lowlevel.**colifilt**(*X*, *ha*, *hb*, *queue=None*)

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.opencl.lowlevel.**get_default_queue**(**args*, ***kwargs*)

Return the default queue used for computation if one is not specified.

This function is memoized and so only one queue is created after multiple invocations.

d

dtcwt, 31
dtcwt.coeffs, 35
dtcwt.compat, 41
dtcwt.keypoint, 35
dtcwt.numpy, 45
dtcwt.numpy.lowlevel, 47
dtcwt.opencl, 48
dtcwt.opencl.lowlevel, 50
dtcwt.plotting, 39
dtcwt.registration, 38
dtcwt.sampling, 37
dtcwt.utils, 40

A

appropriate_complex_type_for() (in module dtcwt.utils), 40
as_column_vector() (in module dtcwt.utils), 40
asfarray() (in module dtcwt.utils), 40
axis_convolve() (in module dtcwt.opencl.lowlevel), 50

B

backend_name (in module dtcwt), 34
biort() (in module dtcwt.coeffs), 35

C

cl_highpasses (dtcwt.opencl.Pyramid attribute), 49
cl_lowpass (dtcwt.opencl.Pyramid attribute), 49
cl_scales (dtcwt.opencl.Pyramid attribute), 49
coldfilt() (in module dtcwt.numpy.lowlevel), 48
coldfilt() (in module dtcwt.opencl.lowlevel), 50
colfilter() (in module dtcwt.numpy.lowlevel), 47
colfilter() (in module dtcwt.opencl.lowlevel), 50
colifilt() (in module dtcwt.numpy.lowlevel), 48
colifilt() (in module dtcwt.opencl.lowlevel), 50

D

drawcirc() (in module dtcwt.utils), 40
drawedge() (in module dtcwt.utils), 40
dtcwt (module), 31
dtcwt.coeffs (module), 35
dtcwt.compat (module), 41
dtcwt.keypoint (module), 35
dtcwt.numpy (module), 45
dtcwt.numpy.lowlevel (module), 47
dtcwt.opencl (module), 48
dtcwt.opencl.lowlevel (module), 50
dtcwt.plotting (module), 39
dtcwt.registration (module), 38
dtcwt.sampling (module), 37
dtcwt.utils (module), 40
dtwaveifm() (in module dtcwt.compat), 41
dtwaveifm2() (in module dtcwt.compat), 42
dtwaveifm2b() (in module dtcwt.compat), 43
dtwaveifm3() (in module dtcwt.compat), 44
dtwavexfm() (in module dtcwt.compat), 41
dtwavexfm2() (in module dtcwt.compat), 42
dtwavexfm2b() (in module dtcwt.compat), 43
dtwavexfm3() (in module dtcwt.compat), 43

E

estimatereg() (in module dtcwt.registration), 38

F

find_keypoints() (in module dtcwt.keypoint), 35
forward() (dtcwt.numpy.Transform1d method), 45
forward() (dtcwt.numpy.Transform2d method), 46
forward() (dtcwt.numpy.Transform3d method), 46
forward() (dtcwt.opencl.Transform2d method), 49
forward() (dtcwt.Transform1d method), 32
forward() (dtcwt.Transform2d method), 32
forward() (dtcwt.Transform3d method), 33

G

get_default_queue() (in module dtcwt.opencl.lowlevel), 51

H

highpasses (dtcwt.numpy.Pyramid attribute), 45
highpasses (dtcwt.Pyramid attribute), 34

I

inverse() (dtcwt.numpy.Transform1d method), 46
inverse() (dtcwt.numpy.Transform2d method), 46
inverse() (dtcwt.numpy.Transform3d method), 47
inverse() (dtcwt.opencl.Transform2d method), 49
inverse() (dtcwt.Transform1d method), 32
inverse() (dtcwt.Transform2d method), 32
inverse() (dtcwt.Transform3d method), 33

L

lowpass (dtcwt.numpy.Pyramid attribute), 45
lowpass (dtcwt.Pyramid attribute), 34

O

overlay_quiver() (in module dtcwt.plotting), 39

P

pop_backend() (in module dtcwt), 34
preserve_backend_stack() (in module dtcwt), 34
push_backend() (in module dtcwt), 34
Pyramid (class in dtcwt), 34
Pyramid (class in dtcwt.numpy), 45
Pyramid (class in dtcwt.opencl), 48

Q

qshift() (in module dtcwt.coeffs), 35

R

reflect() (in module dtcwt.utils), 40
rescale() (in module dtcwt.sampling), 37
rescale_highpass() (in module dtcwt.sampling), 37

S

sample() (in module dtcwt.sampling), 37
sample_highpass() (in module dtcwt.sampling), 37
scales (dtcwt.numpy.Pyramid attribute), 45
scales (dtcwt.Pyramid attribute), 34
stacked_2d_matrix_matrix_prod() (in module dtcwt.utils), 40
stacked_2d_matrix_vector_prod() (in module dtcwt.utils), 40
stacked_2d_vector_matrix_prod() (in module dtcwt.utils), 40

T

Transform1d (class in dtcwt), 31
Transform1d (class in dtcwt.numpy), 45
Transform2d (class in dtcwt), 32
Transform2d (class in dtcwt.numpy), 46
Transform2d (class in dtcwt.opencl), 49
Transform3d (class in dtcwt), 33
Transform3d (class in dtcwt.numpy), 46

U

upsample() (in module dtcwt.sampling), 38
upsample_highpass() (in module dtcwt.sampling), 38

V

velocityfield() (in module dtcwt.registration), 38

W

warp() (in module dtcwt.registration), 39
warptransform() (in module dtcwt.registration), 39