A Matlab Tour of Second Generation Bandelets

Gabriel Peyré[*]
CMAP, École Polytechnique

Stéphane Mallat
CMAP, École Polytechnique

Abstract

The second generation bandelets [Peyré and Mallat 2005c] is a orthogonal multiscale transform that is able to capture the geometric content of images and surfaces. The goal of this note is to give some insight about a proof-of-concept MATLAB implementation. The compagnon source code (containing the functions described in this paper together with other tools and scripts) is freely available on MATLAB CENTRAL [The Mathworks 2005].

This MATLAB toolbox is a proof-of-concept implementation of the Bandelet Transform, which was originally presented in [Peyré and Mallat 2005c]. Image processing results are presented in [Peyré and Mallat 2005a], and mathematical proofs of the results in [Peyré and Mallat 2005b]. The homepage for the project is [Peyré 2005]

Instalation Unzip the package in a directory from where you will run MATLAB. Put the scripts contained in `toolbox/` into you MATLAB path (these are useful function, not part of the bandelet transform itself). If you are under linux/macos, recompile the mex file `perform_haar_transform.cpp` using the `compile_mex` script. If you want to use wavelet transforms more efficient than the Haar basis, you will also need to install WAVELAB [Donoho et al. 1999]. If you want to use your own wavelet transform, then you should edit the file `perform_wavelet_transform.m`.

Important This code is only intended to explain the basics behind the bandelet transform, and to be able to play with the major concepts and properties of these orthogonal basis. It does not contain any coding scheme, nor a fast implementation.

Overview See the `text_*` scripts to check the main features of the algorihtms. Remember you have to run these scripts *line by line* to really understand the internal (there is no fancy user interface).

1 Implementation of the Transform at a Fixed Scale

In this section we explain how to compute the bandelet transform of a given image at a fixed scale. This is the key ingredient of the whole bandelet algorithm, that uses first a wavelet transform and then a bandelet transform on each scale and orientation of the wavelet domain. In the following, the transform at fixed scale will be called the *bandelet* transform (current section), and the whole transform the *wavelet-bandelet* transform (see next section).

Note that this mono-resolution *bandelet* transform can be used directly on the image, and gives quite good results. The main drawback of this mono-resolution approach is that it produces blocking artifacts due to the piecewise-constant nature of the basis functions. In the next section, we will show how the wavelet-bandelet transform fixes this problem by performing the approximation in the wavelet domain.

Warped Haar Transform First of all, you will need a 1D haar transform. This is a basic code, and it should be as fast as possible. Listing 1 is a simple implementation in MATLAB. Function `perform_warped_haar`, listing 2, applies the haar transform in a given direction specified by its angle `theta`. This corresponds to the steps (5)-(7) of the algorithm in [Peyré and Mallat 2005c]. The parameter `dir` is +1 for forward transform, and -1 for backward transform.

Best Geometry Selection In order to find the best direction in a tiny square, we need to check a set of directions, this is step (8) of the algorithm presented in [Peyré and Mallat 2005c]. The sampling of the direction is exhaustive (we keep all possible directions joining pairs of sampling points). The function `compute_best_direction`, listing 3, first selects this set of directions and stores them in `Theta`, and then computes the Lagragian for each direction. It then chooses the direction which gives rise to the smallest Lagrangian (as shown in figure 3 of the paper [Peyré and Mallat 2005c]). Note that we also test the untransformed data (which is conveniently represented with the `Inf` token).

Quadtree Construction The quadtree is computed using the function `compute_quadtree`, listing 4. The quadtree is represented using two 2D images (of the same size as the original image `M`), see figure 1. The image `QT` represents the depth of the quadtree for each pixel in the image (it wastes some memory but it is very convenient). The image `Theta` stores the optimal direction for each pixel. Note that both `QT` and `Theta` are constant on each square of the final quadtree. To compute this quadtree, we first compute the Lagrangian for each dyadic square for the smallest size (the size of these squares is specified by the parameter `j_min`). Then the algorithm goes from the smallest size to the biggest, and tries to merge each group of four squares. To do so, it simply compute the Lagrangian of the merged square, and compares it to the sum of the 4 Lagrangians (plus 1 bit, which is the cost of coding a split in the tree). This is step (10) of the algorihtm presented in [Peyré and Mallat 2005c].

On figure 1, one can see a quadtree computed with this procedure for $T = 10$. One can note that even in squares with no

[*]e-mail:gabriel.peyre@polytechnique.fr

geometric features (on which the function is constant), the algorithm nevertheless choose some arbitrary orientation. This is because in these squares the function does not have zero mean, so a bandelet transform (with any direction) is better than leaving the data untransformed. This situation does not appear in the wavelet-bandelet algorithm (see next section), since in flat areas, a wavelet transform has zero mean.

Bandelet Transform Forward and backward transform use exactly the same MATLAB code, by simply setting `dir` to either +1 or -1. The function `perform_bandelet_transform`, listing 5, does the job by transforming each leaf of the quadtree. This function also returns `m_geom` which is the number of geometric parameters needed to describe the bandelet basis. It is the sum of the geometry coefficients (one coefficient per leaf with geometry) and the quadtree description coefficients (one bit per 1:4 split rule, which can be replaced by approximately 1/7 th of a coefficient).

On figure 2 one can see the compression of a geometric image with various thresholds `T`. This clearly shows the limitation of the direct application of the bandelet transform in a mono-resolution setting: blocking artifacts are clearly visible at low bit rates.

2 Implementation of the Full Bandelet Scheme

In order to perform the full algorithm presented in [Peyré and Mallat 2005c], one must apply the bandelet transform presented in the previous section to each scale and orientation of a wavelet transform of the image.

Computing the Wavelet Transform You will need a 2D isotropic wavelet transform, such as the one implemented in WAVELAB [Donoho et al. 1999]. The function `perform_wavelet_transform` is a simple wrapper to WAVELAB (see script 6), but you can replace it by your own wavelet transform (orthogonal or biorthogonal), you just have to respect the same output layout.

Computing the Quadtree for Each Scale The function `compute_wavelet_quadtree` computes a single quadtree for each scale and for each orientation. Sharing the same quadtree for the three orientations leads to a little efficiency increase, at the price of a more complex implementation. The quadtree structure `QT` and geometry `Theta` are stored in a image that have the same size as the original image `M` (and they have the same hierarchical structure as the wavelet transformed image `MW`).

Performing the Wavelet-Bandelet transform As for the construction of the quadtree, the computation of the wavelet-bandelet transform is straightforward, it simply involves looping over each scale and each orientation, and calling the mono-resolution `perform_bandelet_transform` function. This is done in the script `perform_wav_band_transform`, see listing 8.

On figure 4 one can see the result of the compression of a geometric image using this wavelet-bandelet scheme. As one can see, the blocking artifacts have disappeared. On such a simple geometric image, the mono-scale implementation gives very similar PSNR results, but on complex natural images, the multi-scale procedure is by far more efficient and robust.

References

DONOHO, D., DUNCAN, M. R., HUO, X., AND LEVI, O., 1999. Wavelab web site. http://www-stat.stanford.edu/~wavelab/.

PEYRÉ, G., AND MALLAT, S. 2005. Discrete bandelets with geometric orthogonal filters. *Proc. of ICIP05* (Sept.).

PEYRÉ, G., AND MALLAT, S. 2005. Orthogonal bandelet basis for geometric image approximation. *To be published*.

PEYRÉ, G., AND MALLAT, S. 2005. Surface compression with geometric bandelets. *ACM Transactions on Graphics, Vol. 24(3), (Proc. of SIGGRAPH'05) 24*, 3 (Aug.).

PEYRÉ, G., 2005. Bandelets homepage. http://www.cmap.polytechnique.fr/~peyre/bandelets/.

THE MATHWORKS, 2005. Matlab central web site. http://www.mathworks.com/matlabcentral/.


```
function x = perform_haar_transform(x, dir);
x = x(:);      % to be sure we have a column vector
J = floor( log2(length(x)) );   % number of scales
if dir==1      % forward transform
  for j=1:J
    c = x(1:2^(j-1):end);    % previous coarse signal
    x(1:2^j:end) = ...       % new coarse signal
      ( c(1:2:end) + c(2:2:end) )/sqrt(2);
    x(1+2^(j-1):2^j:end) = ...  % new details
      (c(1:2:end)-c(2:2:end))/sqrt(2);
  end
else           % backward transform
  for j=J:-1:1
    y = x(1:2^(j-1):end);
    x(1:2^j:end) = ...
      ( y(1:2:end) + y(2:2:end) )/sqrt(2);
    x(1+2^(j-1):2^j:end) = ...
      ( y(1:2:end) - y(2:2:end) )/sqrt(2);
  end
end
```

Listing 1: Function `perform_haar_transform`

```
function M = perform_warped_haar(M,theta,dir)
if theta==Inf    % special token : no geometry
  return;        % nothing to do
end
n = size(M,1);
% sampling location
[Y,X] = meshgrid(1:n,1:n);
% projection on orthogonal direction
t = -sin(theta)*X(:) + cos(theta)*Y(:);
% order points in increasing order
[tmp,I] = sort(t);
M(I) = perform_haar_transform(M(I),dir);
```

Listing 2: Function `perform_warped_haar`

```
function [MW,theta,L] = compute_best_direction(M,T)
% samples the direction
[Y,X] = meshgrid(0:n-1, 0:n-1); X = X(:); Y = Y(:);
X(1) = []; Y(1) = [];
Theta = atan2(Y(:),X(:)); Theta = unique(Theta);
Theta = [-Theta(end-1:-1:2); Theta]';
% take mid points
Theta = ( Theta + [Theta(2:end),Theta(1)+pi] )/2;
Theta = [Theta, Inf]; % add 'no geometry' token
% now check for each direction
n = size(M,1);
t = pi/(2*n*s);
Theta = [t/2:t:pi-t/2, Inf];
% compute the lagrangian
L = []; % to store the
for theta = Theta
  MW = perform_warped_haar(M,theta,1);
  % compute the error
  MWt = MW .* (abs(MW)<T);  % residual
  m = sum( abs(MW(:))>T );  % number of coef above T
  % compute the lagrangian
  l = sum(MWt(:).^2) + m*T^2;
  if theta~=Inf
    % here we need to store a geometric coefficient
    l = l + T^2;
  end
  L = [L; l];
end
% find minimum of lagrangian
[L,I] = min(L);
theta = Theta(I(end));
MW = perform_warped_haar(M,theta,1);
```

Listing 3: Function `compute_best_direction`

```
function [QT,Theta] = ...
    compute_quadtree(M,T,j_min,j_max)
n = size(M);
% we assume that a split in the QT is
% coded with 1/7 th of coefficients.
gamma = 1/7;
QT = zeros(n)+j_min; Theta = zeros(n);
L = zeros(n/2^j_min);    % the current lagrangian
% compute bandelet approximation for each square
for kx=0:n/2^j_min-1
  for ky=0:n/2^j_min-1
    selx = kx*2^j_min+1:(kx+1)*2^j_min;
    sely = ky*2^j_min+1:(ky+1)*2^j_min;
    % compute the optimal direction on this square
    [tmp,Theta(selx,sely),L(kx+1,ky+1)] = ...
compute_best_direction(M(selx,sely),T);
  end
end

% perform the bottom-up procedure, trying to merge
% 4 squares into 1 if it decreases the  lagrangian
for j=j_min+1:j_max
  % new lagrangian for this size of squares
  L1 = zeros(n/2^j);
  for kx=0:n/2^j-1
    for ky=0:n/2^j-1
      selx = kx*2^j+1:(kx+1)*2^j;
      sely = ky*2^j+1:(ky+1)*2^j;
      % the lagrangian of the 4 squares splited (add
      % gamma penalty because of the split)
      l_sum = L(2*kx+1,2*ky+1) + ...
          L(2*kx+2,2*ky+1) + L(2*kx+1,2*ky+2) + ...
          L(2*kx+2,2*ky+2) + gamma*T^2;
      % the lagrangian of the 4 squares once merged
      [tmp,theta,l] = ...
          compute_best_direction(M(selx,sely),T);
      % perform the conditional merging
      if l < l_sum
        L1(kx+1,ky+1) = l; QT(selx,sely) = j;
        Theta(selx,sely) = theta;
      else
        L1(kx+1,ky+1) = l_sum;
      end
    end
  end
  L = L1;
end
```

Listing 4: Function `compute_quadtree`

```
function [MB,m_geom] = ...
    perform_bandelet_transform(M,QT,Theta,dir)
n = size(M,1); MB = zeros(n); m_geom = 0;
gamma = 1/7;    % 4:1 split cost
j_min = min(QT(:)); j_max = max(QT(:));
% display subdivision
for j=j_max:-1:j_min
  w = 2^j/n;
  for kx=0:n/2^j-1
    for ky=0:n/2^j-1
      selx = kx*2^j+1:(kx+1)*2^j;
      sely = ky*2^j+1:(ky+1)*2^j;
      if QT(kx*2^j+1, ky*2^j+1)==j
        % this is a leaf, transform it
        theta = Theta(kx*2^j+1, ky*2^j+1);
        MB(selx,sely) = ...
          perform_warped_haar(M(selx,sely),theta,dir);
        % one coefficient for geometry
        m_geom = m_geom + (theta~=Inf);
      else % add split cost
        m_geom = m_geom + gamma;
      end
    end
  end
end
```

Listing 5: Function `perform_bandelet_transform`

```
function MW = perform_wavelet_transform(M,Jmin, dir)
% retrieve the 7-9 CDF biorthogonal filters
[qmf,dqmf] = MakeBSFilter( 'CDF', [4,4] );
% compute biorthogonal wavelet transform
if dir==1
  MW = FWT2_SBS(M,Jmin,qmf,dqmf);
else
  MW = IWT2_SBS(M,Jmin,qmf,dqmf);
end
```

Listing 6: Function `perform_wavelet_transform`

```
function [QT,Theta] = ...
    compute_wavelet_quadtree(M,Jmin,T,j_min,j_max)
% perform the wavelet transform
MW = perform_wavelet_transform(M,Jmin, 1);
n = size(M,1); Jmax = log2(n)-1;
QT = zeros(n); Theta = zeros(n);
% compute the transform for each scale and direction
for j=Jmax:-1:Jmin  % for each scale
  j_max = min(j_max, j);
  for q=1:3    % for each orientation
    if q==1 % 1st quadrant
      selx = 1:2^j; sely = (2^j+1):2^(j+1);
    elseif q==2
      selx = (2^j+1):2^(j+1); sely = 1:2^j;
    else
      selx = (2^j+1):2^(j+1); sely = (2^j+1):2^(j+1);
    end
    [QT(selx,sely),Theta(selx,sely)] = ...
      compute_quadtree(MW(selx,sely),T,j_min,j_max);
  end
end
```

Listing 7: Function `compute_wavelet_quadtree`

```
function [MB,m_geom] = ...
    perform_wav_band_transform(M,Jmin,QT,Theta,dir)
MB = M;
if dir==1
  % perform the wavelet transform
  MB = perform_wavelet_transform(M,Jmin, 1);
end
n = size(M,1); Jmax = log2(n)-1;
m_geom = 0;
% compute the transform for each scale and direction
for j=Jmax:-1:Jmin  % for each scale
  for q=1:3   % for each orientation
    if q==1 % 1st quadrant
      selx = 1:2^j; sely = (2^j+1):2^(j+1);
    elseif q==2
      selx = (2^j+1):2^(j+1); sely = 1:2^j;
    else
      selx = (2^j+1):2^(j+1); sely = (2^j+1):2^(j+1);
    end
    [MB(selx,sely),m] = ...
      perform_bandelet_transform(MB(selx,sely), ...
      QT(selx,sely),Theta(selx,sely),dir);
    m_geom = m_geom + m;
  end
end
if dir==-1
  % perform the inverse wavelet transform
  MB = perform_wavelet_transform(MB,Jmin, -1);
end
```

Listing 8: Function `perform_wav_band_transform`

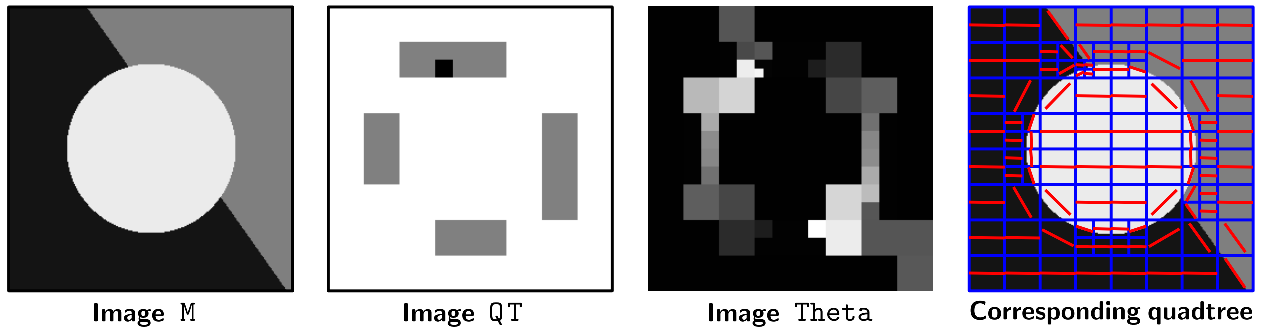

Figure 1: *An example of quadtree computed using the function* `compute_quadtree`.

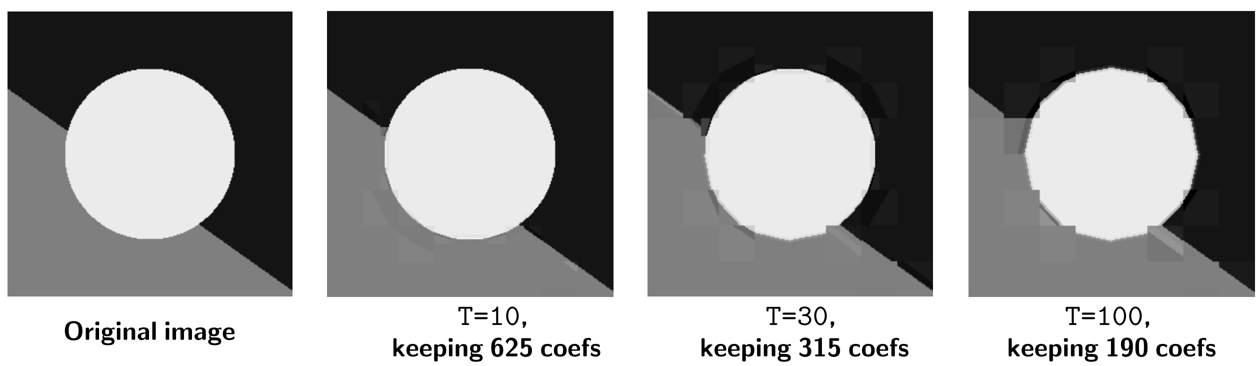

Figure 2: *Image compression using mono-resolution bandelets.*

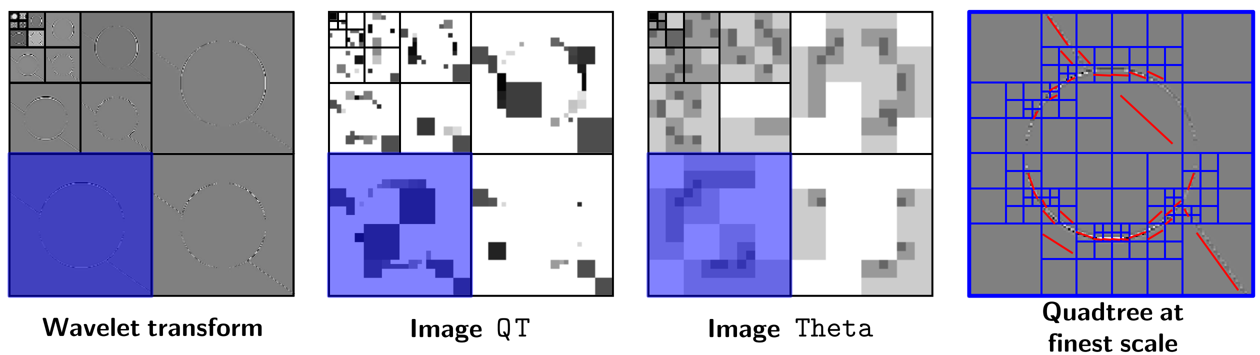

Figure 3: *An example of quadtree computed using the function* `compute_wavelet_quadtree`.

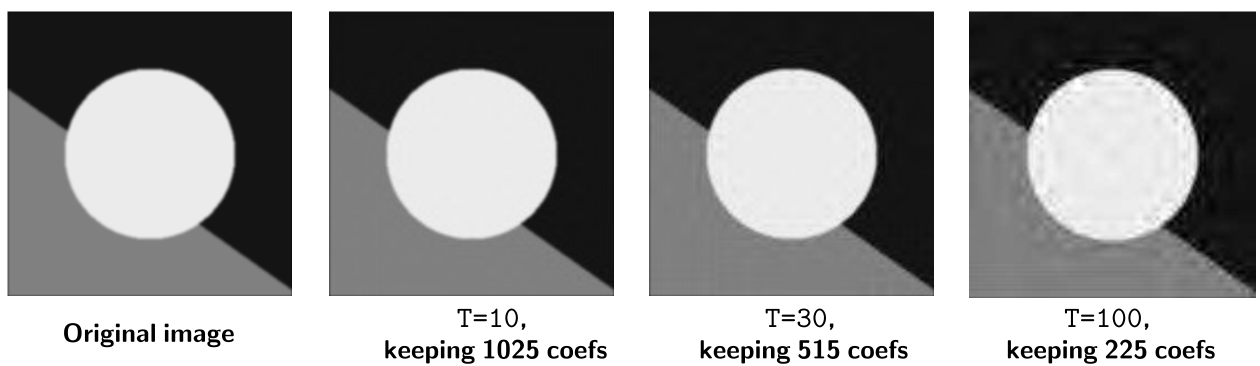

Figure 4: *Image compression using multi-resolution bandelets.*