Image Processing
lab 3

Klaas Kliffen Jan Kramer

December 11, 2015

Exercise 1 – 1-D wavelet transforms

a. The algorithm given in the assignment can be represented as a filter bank based on the Haar scaling and wavelet vectors. Normally the whole input would be convolved with these vectors. However since the two-point sums and differences are taken, this convolution is combined with downscaling. Our implementation of a j-scale DWT is based on this algorithm by applying this "filter bank" j times. Otherwise its implementation is rather trivial.

```
% function to perform 1D Haar wavelet transform
function retval = IPdwt(x,s)
sqrt2 = sqrt(2);
out = x;

initl = length(out);

for i = 1 : s
  % Get the odd and even elements
  odds = out(1:2:initl);
  evens = out(2:2:initl);
   % Calculate the means and details
  sums = (odds + evens);
  diffs = (odds - evens);
  % Put the new values
  out(1:initl) = [sums, diffs] / sqrt2;
  initl /= 2;
end

retval = out;
end
```

b. The inverse of the wavelet transform is basicly doing all steps from the `IPdwt` in reverse. The inital length is set to the end of the length of the last sum vector. This can be determined by taking the lenght and divide it by a power of 2 with the scale. Then for each step the sum and difference vectors are retrieved from the input vector by taking halves of the input vector. Two new component vectors are created for the values. These need to be scaled again to retrieve the right results. The component vectors are then interleaved and replace their original values in the input vector. This is repeated until the orignal scale of the transformation is reached.

```
% function to perform 1D Haar wavelet transform
function retval = IPidwt(x,s)
sqrt2 = sqrt(2);
out = x;

% Determine the initial length
initl = length(x) / (2^(s-1));

for i = 1 : s
  % Retrieve the sums and the differences
  sums = out(1:initl/2);
  diffs = out(initl/2+1:initl);
  % Calculate and scale the result
  plus = (sums+diffs)/sqrt2;
  mins = (sums-diffs)/sqrt2;
  % Combine the new values
  combined = zeros(initl,1);
  combined(1:2:end) = plus;
  combined(2:2:end) = mins;
  % Store them in the output matrix
  out(1:initl) = combined;
  % Increase the length or the next iteration
  initl *= 2;
end

retval = out;
end
```

Exercise 2 – 2-D wavelet transforms

a. According to the Section 7.5 in the book the extension of 1D DWT to 2D is simple, because of the separable scaling and wavelet functions. It also mentions that the 2D DWT can be computed by first doing a 1D DWT of the columns and then doing a 1D DWT of the rows. Note that one could also calculate the DWT first for rows and then for columns. Our implementation uses this fact. In each iteration j the algorithm of exercise 1 is applied to each row and then to each column to get the approximation, the horizontal detail, the vertical detail and the diagonal detail.

```
% function to perform 2D Haar wavelet transform
function retval = IPdwt2(x, j)
% note that x should be double instead of uint, because
% the result can get negative
out = x;
coef = 1/sqrt(2);

initrow = size(out, 1);
initcol = size(out, 2);

for i = 1 : j
  % 1D DWT along the rows
  odds_c = out(1:initrow, 1:2:initcol);
  evens_c = out(1:initrow, 2:2:initcol);
  sums = (odds_c + evens_c);
```

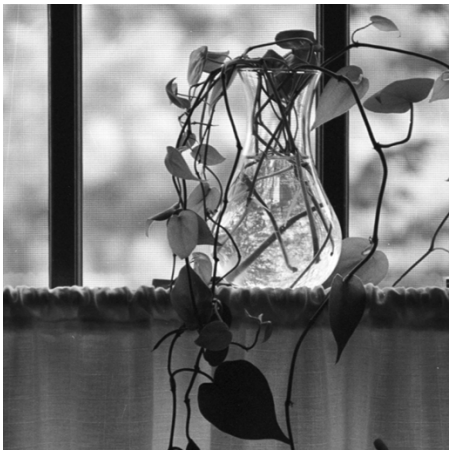
```
16    diffs = (odds_c - evens_c);
17    out(1:initrow, 1:initcol) = [sums, diffs] * coef;
18
19    % 1D DWT along the columns
20    odds_r = out(1:2:initrow, 1:initcol);
21    evens_r = out(2:2:initrow, 1:initcol);
22    sums = (odds_r + evens_r) * coef;
23    diffs = (odds_r - evens_r) * coef;
24
25    mid_r = initrow / 2;
26    mid_c = initcol / 2;
27
28    % save the parts as in the figures in the book
29    % approximation image
30    out(1:mid_r, 1:mid_c) = sums(:, 1:mid_c);
31    % vertical detail
32    out(mid_r+1:initrow, 1:mid_c) = sums(:, mid_c+1:initcol);
33    % horizontal detail
34    out(1:mid_r, mid_c+1:initcol) = diffs(:, 1:mid_c);
35    % detail detail
36    out(mid_r+1:initrow, mid_c+1:initcol) = diffs(:, mid_c+1:initcol);
37
38    initrow = mid_r;
39    initcol = mid_c;
40  end
41
42  retval = out;
43
44  end
```

b. The result of the IPdwt2 by itself is too dark to see details clearly, since the values in the detail parts of the resulting image are around zero(black). Therefore our implementation shifts the input image to the 2d DWT such that the resulting details are centered around a gray value. After that we can constrast-stretch the various detail images by iterating over the j-scale and at the end the approximation image is stretched.

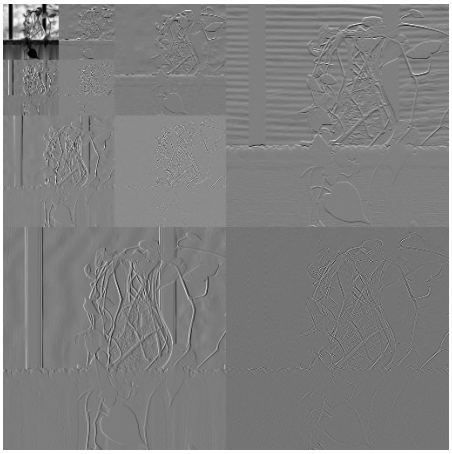
```
1  function retval = IPdwt2scale(x, j)
2  % calculate the dwt with a shifted image around 0 (assumes doubles)
3  out = x - 0.5;
4  out = IPdwt2(out, j);
5  out = out + 0.5;
6
7  initrow = size(out, 1);
8  initcol = size(out, 2);
9
10 % iterate trough the levels in the image
11 for i = 1 : j
12   mid_r = initrow / 2;
13   mid_c = initcol / 2;
14
15   % contrast stretch the horizontal details
16   w = out(1:mid_r, mid_c+1:initcol);
17   out(1:mid_r, mid_c+1:initcol) = (w - min(min(w))) * (1 / (max(max(w
         )) - min(min(w))));
```

```
18
19    % contrast stretch the vertical details
20    w = out(mid_r+1:initrow, 1:mid_c);
21    out(mid_r+1:initrow, 1:mid_c) = (w - min(min(w))) * (1 / (max(max(w
          )) - min(min(w))));
22
23    % contrast stretch the diagonal details
24    w = out(mid_r+1:initrow, mid_c+1:initcol);
25    out(mid_r+1:initrow, mid_c+1:initcol) = (w - min(min(w))) * (1 / (
          max(max(w)) - min(min(w))));
26
27    initrow = mid_r;
28    initcol = mid_c;
29  end
30
31  % contrast stretch the approximation image
32  w = out(1:initrow, 1:initcol);
33  out(1:initrow, 1:initcol) = (w - min(min(w))) * (1 / (max(max(w)) -
        min(min(w))));
34
35  retval = out;
36  end
```

```
1
2  x = im2double(imread('../images/vase.tif'));
3
4  %% lab 3 ex 2abcd
5  y = IPdwt2(x, 3);
6
7  imwrite(y, 'unscaled.png');
8  imwrite(im2uint8(IPdwt2scale(x, 3)), 'scaled.png');
9  imwrite(im2uint8(IPidwt2(y,3)), 'output.png');
10 % the difference
11 sum(sum(im2uint8(x - IPidwt2(y,3))))
```



Original image



3 scale wavelet transform

Figure 1: 3 scale wavelet transform of the orignal image

d. According to Section 7.5 the 2D inverse DWT can also be computed by using a 1D inverse DWT function. So similarly on how our 2D DWT implementation is based on our 1D DWT implemention, the 2D inverse DWT is also based on the 1D DWT implementation. Note however that the order of applying 1D DWT first to rows and then columns in the 2D DWT, has to be inverted in the 2D inverse DWT. Hence our iplementation first applies a 1D inverse DWT to the columns and the one to the rows.

```
% inverse discrete wavelet transform
function retval = IPidwt2(x, s);
out = x;
coef = 1/sqrt(2);

[height, width] = size(x);
initrow = height / (2^(s-1));
initcol = width / (2^(s-1));

for i = 1 : s
  mid_r = initrow / 2;
  mid_c = initcol / 2;

  % make sure we swap horizontal and vertical details
  rowsums = [out(1:mid_r, 1:mid_c), out(mid_r+1:initrow, 1:mid_c)];
  rowdiffs = [out(1:mid_r, mid_c+1:initcol), out(mid_r+1:initrow,
      mid_c+1:initcol)];

  % Calculate and scale the result
  plus = (rowsums+rowdiffs);
  mins = (rowsums-rowdiffs);
  % Combine the new values
  combined = zeros(initrow,initcol);
  combined(1:2:end,:) = plus * coef;
  combined(2:2:end,:) = mins * coef;
```

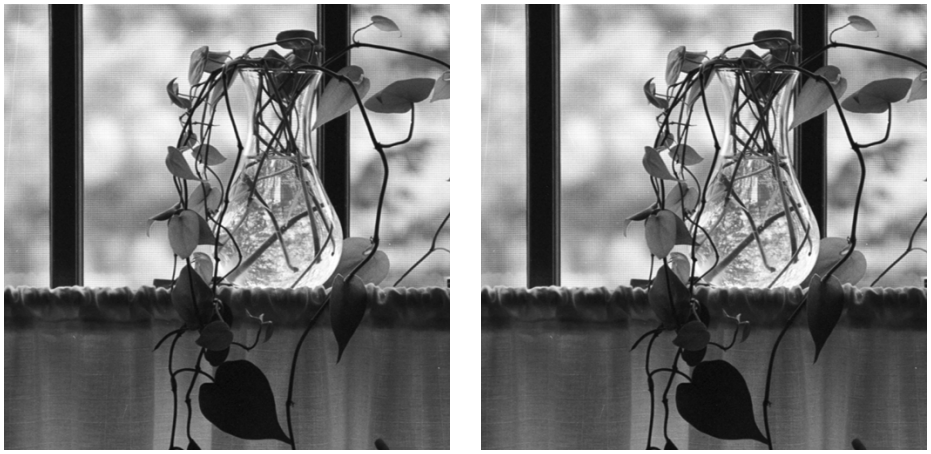
```
    % Replace the values in the image
    out(1:initrow,1:initcol) = combined;

    % Do the same with the columns
    colsums = out(1:initrow,1:mid_c);
    coldiffs = out(1:initrow,mid_c+1:initcol);

    % Calculate and scale the result
    plus = (colsums+coldiffs);
    mins = (colsums-coldiffs);
    % Combine the new values
    combined(:,1:2:end) = plus * coef;
    combined(:,2:2:end) = mins * coef;
    % Replace the values in the image
    out(1:initrow,1:initcol) = combined;

    initrow *= 2;
    initcol *= 2;
end

retval = out;
end
```



Original image | Restoration of the image form figure 1

Figure 2: 3 scale wavelet transform restoration

Exercise 3 – Image Compression

a. First the wavelet transform is performed on the input image. A threshold matrix consisting solely of the threshold value is used to find all pixels larger than the threshold in the transform. The thresholded image is then retrieved by pointwise multiplication of the results of comparing the threshold matrix with the wavelet transform. Since the approximation of the original matrix is also passed by the threshold, it needs to be reconstructed. This is done by copying the values from the wavelet transform back to the thresholded image. The compressed image is then retrieved by applying the inverse wavelet transform.

The root mean square error and the mean square signal to noise ratio are calculated from the given formulas. For the compression ratio the the build-in function of entropy is used to calculate the entropy of the compressed and original image. The entropy is a value for how complex an image is and the minium amount of data which is needed to store the image. Compressing the image lowers the complexity. Dividing the original entropy by the compressed entropy yields a compression ratio.

```
% Function to compress an image using wavelet transform
function retval = IPwaveletcompress(img, scale, threshold)

[width,height] = size(img);
wl = width / (2^scale);
hl = height / (2^scale);

% Wavelet transform
wtrans = IPdwt2(img,scale);

% Construct a matrix for the threshold
thresholdmat = threshold * ones(size(img));
% Matrix containing 1 for pixels above the threshold
results = abs(wtrans) > thresholdmat;
% Perform the thresholding by elementwise multiplying
threshed = zeros(size(img));
threshed = wtrans .*results;

% Copy the original image part (for dark values in the original
threshed(1:hl,1:wl) = wtrans(1:hl,1:wl);

% Convert it back
compressed = IPidwt2(threshed, scale);

printf("Scale: %d Threshold: %f\n", scale, threshold);
error = compressed - img;
errorsq = error .* error;
rmse = sqrt(mean(mean(errorsq)));
printf("Root mean square error: %f\n",rmse);

squared = compressed .* compressed;
snr = sum(sum(squared)) / sum(sum(errorsq));
printf("Mean square signal to noise: %f\n",snr);

% Calculatute the compression
orig = entropy(im2uint8(img));
comp = entropy(im2uint8(threshed));
printf("Compress ratio: %f:1\n\n",orig/comp);


retval = compressed;

end
```

b. The quantitative properties for several different scales and thresholds can be seen in table 1. Globally the signal to noise ratio decreases and the errors and compression ratio increase

while increasing scale and threshold. Although for this image the increasing the scale past 9 was not possible, due to its size being a square of 512 pixels. It would seem that increasing it further would not compress the image any further past a compression ratio of 37. Increasing the threshold will yield in higher compression ratios. Although the signal to noise ratio decreases exponentiall, while the errors increase almost linearly.

Scale	**Threshold**	ϵ_{rms}	**SNR**	**Compression ratio**
1	0.02	0.0086	1277.34	2.59:1
3	0.02	0.0157	385.47	17.42:1
3	0.05	0.0249	152.09	24.87:1
3	0.10	0.0322	90.33	29.14:1
5	0.02	0.0220	194.71	34.00:1
7	0.02	0.0272	127.44	36.40:1
7	0.05	0.0575	27.627	113.18:1
7	0.10	0.1035	7.837	460.20:1
9	0.02	0.0321	91.02	36.55:1

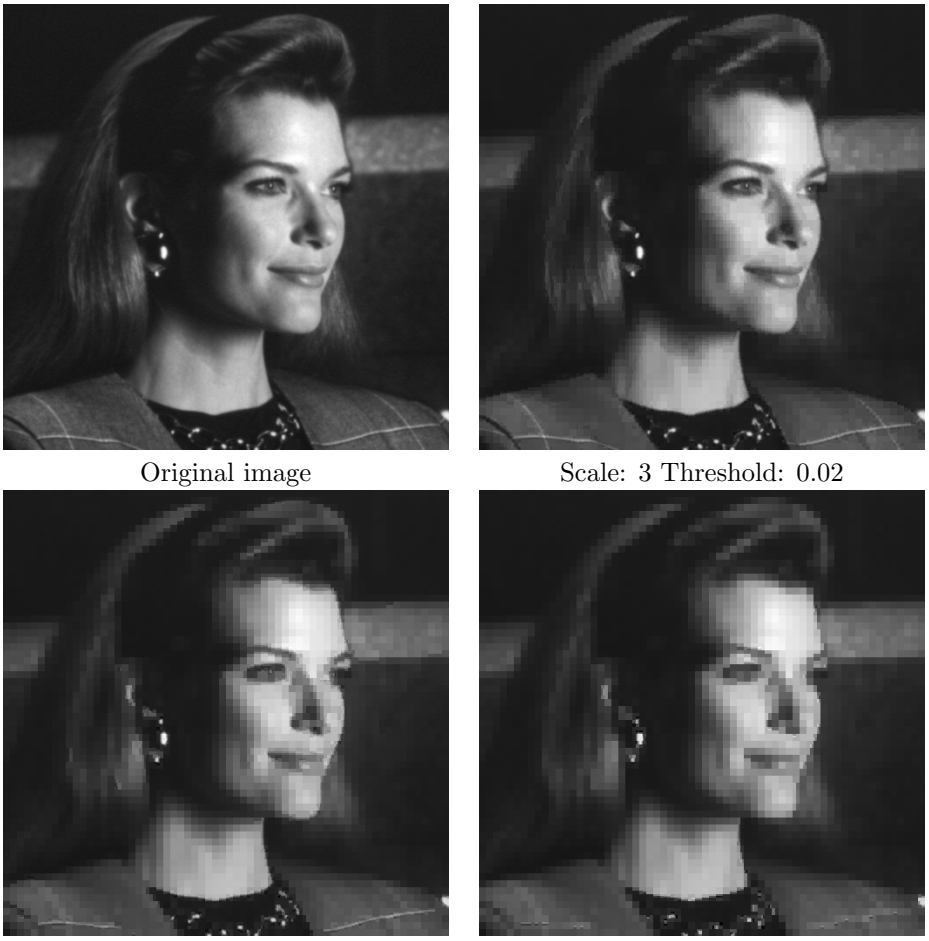
Table 1: Quantitative compression quality for different scales and threshold

Original image | Scale: 1 Threshold: 0.02

Scale: 5 Threshold: 0.02 | Scale: 9 Threshold: 0.02

Figure 3: Increasing wavelet compression scale on an image

Figure 3 show the original image and the 3 compression with different scales. Increasing the scale will cause square like artifacts of a single gray value. The large spots are created because in with the high scale image, coarse details, such as the upper left corner become small differences in the detail compents of that scale. The Thresholding will remove this detail, resulting in a monotone block.

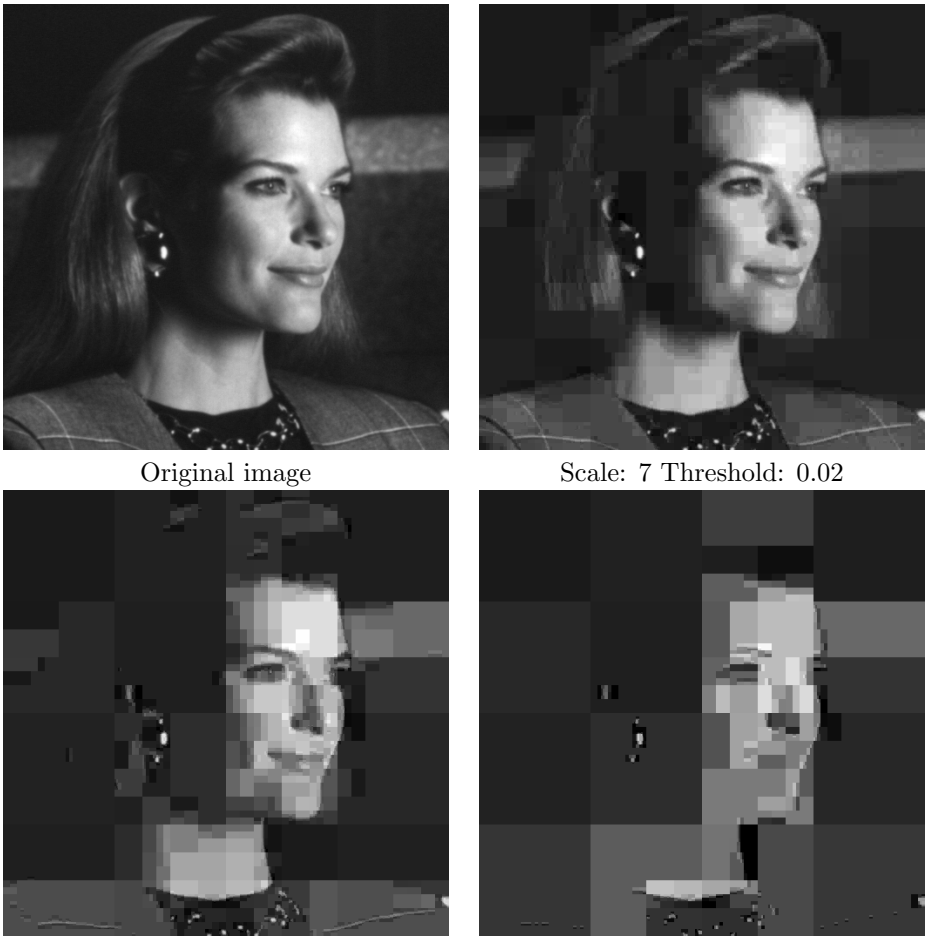


Original image | Scale: 3 Threshold: 0.02

Scale: 3 Threshold: 0.05 | Scale: 3 Threshold: 0.10

Figure 4: Increasing wavelet compression threshold on scale 3 on an image

Figure 4 show the compression on the orignal image with an increasing threshold. The scale is quite low, so course details are not filtered. Increasing the threshold will remove the fine details. This can be seen in a pixelated region around the face. This is caused by the lack of the detail component. Essentially, the approximation image is just resized without increasing the detail, resulting in "large" pixels.



Original image | Scale: 7 Threshold: 0.02

Scale: 7 Threshold: 0.05 | Scale: 7 Threshold: 0.10

Figure 5: Increasing wavelet compression threshold on scale 7 on an image

Figure 5 shows the orignal image being compressed on a high scale with an increasing threshold. The background is almost not visible anymore, only parts of the face contain a bit more detail. This is caused by the higher scale. Course details on the higher scale do not have a large difference, thus are removed by the threshold algorithm. This results in the large blocks in the lower right image. The face contains still some detail, because this is detail is captured on a lower level and the amount of change is higher. Thus a higher threshold is needed to remove that detail from the image.

Task distribution

ex1	design	implementation	answers questions	writing report
Klaas	60%	90%	n.a.	50%
Jan	40%	10%	n.a.	50%

ex2	design	implementation	answers questions	writing report
Klaas	50%	30%	25%	25%
Jan	50%	70%	75%	75%

ex3	design	implementation	answers questions	writing report
Klaas	50%	75%	50%	75%
Jan	50%	25%	50%	25%