Subband Coding of Still Images

May 2005

1 Introduction

1.1 Overview

A transform coder consists of three distinct parts: The *transform*, the *quantizer* and the *entropy coder*. In this laboration you will study all three parts and see how the choice of transform/quantizer/entropy coder affects the performance of the transform coder.

The laboration runs in a MATLAB environment. In MATLAB images are naturally represented as matrices. During the laboration certain test images will be compressed. The test images can be thought of as *original images* in the sense that they are stored as raw samples with 24 bits/pixel. This usually gives more colours than the human eye can discern on a computer screen.

1.2 Preparations

We assume that you have studied the chapters on transform coding, subband and wavelet coding, quantization and entropy coding in the course literature. Read through this document carefully before the laboration. We also suggest that you read the manual (`help <funcname>`) for each function before you use it.

2 Image manipulation in MATLAB

We will be working with colour images. An image can be read into MATLAB using the command `imread`. For future manipulation, we change the pixel values to be floating point values between 0 and 1:

```
>> im1=double(imread('image1.png'))/256;
```

The colour image is stored as a $512 \times 768 \times 3$ matrix, meaning that we have one 512×768 matrix for each of the three RGB colour components (red, green and blue). The image can be viewed using the command `imshow`:

```
>> imshow(im1)
```

We can also view each colour plane separately:

```
>> im1r=im1(:,:,1); im1g=im1(:,:,2); im1b=im1(:,:,3);
>> imshow(im1r), figure, imshow(im1g), figure, imshow(im1b)
```

When coding colour images, we usually use the YCbCr colour space (or another luminance/chrominance colour space), rather than the RGB colour space. To convert the image back and forth between the different colour spaces, use the functions `rgb2ycbcr` and `ycbcr2rgb`:

```
>> im1ycbcr=rgb2ycbcr(im1);
```

See what the luminance and chrominance components look like:

```
>> im1y=im1ycbcr(:,:,1); im1cb=im1ycbcr(:,:,2); im1cr=im1ycbcr(:,:,3);
>> imshow(im1y), figure, imshow(im1cb), figure, imshow(im1cr)
```

The luminance component is basically a greyscale version of the colour image, while the two chrominance components contain information about the colour of the image.

There are six images (named `image1.png` to `image6.png`) with varying content that you can use for your experiments. Take a look at the other images too.

3 Image transformations

The transform that we will use in this laboration is a *dyadic subband decomposition*. The image is filtered with a low-pass and a high-pass filter, both horizontally and vertically. The four filtered versions of the image are then subsampled. So, if the original image is of size 512×768 pixels, after one step of the decomposition we have 4 images of size 256×384. The filters are then applied recursively on the low-pass image. Typically, 4-5 such steps is sufficient. For example, see figures 14.10, 14.11 and 14.12(a) in Sayood's book.

The MATLAB function to do this subband decomposition is called `dsbt2`. As arguments, the function takes an image, the number of splitting steps and what set of filters to use. Read the manual about the function (`help dsbt2`) before using it.

Example:

```
>> im1ys = dsbt2(im1y, 2, 1);
>> figure, imshow(im1ys,[])
```

This performs a two-level subband split on the image, using the 9/7 tap filter pair

and displays the transformed image. As you can see, `dsbt2` returns the result of the subband split in a single matrix. We want to be able to treat each subband separately. The function `sbdivide` can be used to divide the transformed image into a number of separate subbands, and the function `sbmerge` can be used to put them back into a single matrix again.

```
>> im1ysd=sbdivide(im1ys, 2);
>> for k=1:length(im1ysd),figure,imshow(im1ysd{k},[]),end
```

Note that `sbdivide` returns a cell array, which is indexed using curly braces, and that you need to give the number of subband splits performed as an argument.

Inverse transformation is performed with the function `idsbt2`.

Example:

```
>> im1yr = idsbt2(im1ys, 2, 1);
```

Without quantization, `im1y` and `im1yr` should be identical. Due to rounding errors in the computer, there might be a slight (but negligible) difference.

4 Quantization

The second part of a transform image coder is the quantizer. We will only be looking at uniform quantization, using the function `sbquant`. This function takes a transformed image (either as a single matrix, or a cell array from `sbdivide`) and a quantization parameter as arguments. The quantization parameter is the stepsize of the uniform quantizer. To perform inverse quantization (reconstruction), use the function `sbrec`.

Example:

```
>> Q1=0.2;
>> im1ysq = sbquant(im1ys, Q1);
>> im1ysr = sbrec(im1ysq, Q1);
>> im1yr = idsbt2(im1ysr, 2, 1);
>> figure, imshow(im1yr, []);
```

It is also possible to have a separate stepsizes for every subband. The quantization parameter should then be given as a vector of quantization steps, one for each subband in the transformed image.

The distortion (the mean square error) between the original image and a reconstructed image can be calculated as

```
>> dist = mean((im1y(:)-im1yr(:)).^2)
```

Usually, distortion in images is measured by the PSNR (*Peak-to-peak Signal to Noise Ratio*). For an image where the pixels take values between 0 and 1, it is given by

```
>> psnr = 10*log10(1/dist)
```

There is no standard way of defining distortion and PSNR for colour images. The simplest way is to just average the distortions of the three colour components (red, green and blue).

By varying the quantization step (`Q1`), you get reconstructed images with different quality. Try quantizing with different quantization steps and see how the reconstructed image is affected.

5 Entropy coding

The third part of a transform image coder is the entropy coder (or source coder). In this lab we're going to study two methods: Memoryless Huffman coding, using the function `huffman` and memoryless arithmetic coding, using the function `arithcode`. Neither of these functions perform the actual coding, they just estimate the number of bits needed.

The source coding is done on the quantized transform image. To be able to do efficient coding, we first need to estimate the distribution by calculating a histogram of the data we want to code. This can be done with the function `ihist`.

The simplest case is if we use a single code for all subbands.

Example:

```
>> p = ihist(im1ysq(:));
>> bits = huffman(p)
```

This returns the total number of bits required to code the quantized data. Note that the `huffman` function doesn't include the cost to code the huffman tree, and thus the real number of bits needed would be slightly higher.

Normally, we measure the coding efficiency in bits per pixel, which we can get by just dividing the total number of bits with the image size:

```
>> bpp = bits/(512*768)
```

The quantization will of course affect the bit rate: Coarser quantization gives a lower rate while finer quantization gives a higher rate. Change the quantization parameter and see how the rate is affected.

Since the distribution is typically different for different subbands, a more efficient

coding can be performed if we have separate huffman codes for each subband:

```
>> im1ysdq=sbdivide(im1ysq, 2);
>> bits=0;
>> for k=1:length(im1ysdq)
p = ihist(im1ysdq{k}(:));
bits = bits + huffman(p);
end
>> bpp = bits/(512*768)
```

Don't forget to change the second argument to `sbdivide` if you change the number of subband splits.

Change `huffman` to `arithcode` to try arithmetic coding.

6 Chrominance subsampling

Usually, the chrominance signals (Cb and Cr) can be subsampled before coding, without giving noticeable effects on the image quality. Subsampling can be done using the function `imresize`.

Example:

```
>> im1cb2 = imresize(im1cb, 0.5);
```

This will subsample the chrominance image with a factor 2 both horizontally and vertically. The subsampled image is then transformed, quantized and coded. In the decoder, the reconstructed chrominance image is upsampled before transformation back into the RGB colour space:

```
>> im1cbr = imresize(im1cb2r, 2);
```

assuming that the reconstructed subsampled image was called `im1cb2r`.

An alternative way to do subsampling in a subband coder is to just throw away the three highest frequency bands after transformation (set them to zero and don't code them).

7 Own experiments

Now that you know how to use all three parts (transform, quantizer, entropy coder) it is time for you to experiment freely.

Try coding both greyscale images (just the luminance component) and colour images.

NOTE: When comparing two different coders, you should compare them at the same rate and check to see which one gives the lowest distortion (alternatively, keep the distorsion constant and see which one gives the lowest rate). Ideally, the comparison should be done for a whole range of rates.

You should try to find answers to the following questions:

- How does the number of subband splits affect coding performance?
- What subband filter gives the best results? At least compare filter number 0 (Haar filter) and filter number 1 (9/7 tap filter).
- What entropy coding method gives the best results? Compare at several different bit rates.
- How does chrominance subsampling affect coding performance?
- Does the PSNR measure correspond well to visual quality? Compare images that have been coded to the same distortion using different methods.
- What is the lowest rate (in bits per pixel) that gives coded images that are indistinguishable from the original image at normal viewing distance?
- What is the lowest rate that gives an acceptable image quality?

A simple MATLAB function to get you started can be found in `sbcoder.m`.
Copy it to your directory and edit it to suit your needs.

8 Filter coefficients

These are the filter coefficients of the different synthesis filters.

Haar filter	
Low pass filter	High pass filter
0.707107	0.707107
0.707107	-0.707107

9/7 tap filter	
Low pass filter	High pass filter
	-0.037828
-0.064539	-0.023849
-0.040689	0.110624
0.418092	0.377402
0.788486	-0.852699
0.418092	0.377402
-0.040689	0.110624
-0.064539	-0.023849
	-0.037828

LeGall filter	
Low pass filter	High pass filter
	0.176777
0.353553	0.353553
0.707107	-1.060660
0.353553	0.353553
	0.176777

2/6 tap filter	
Low pass filter	High pass filter
-0.088388	
0.088388	
0.707107	0.707107
0.707107	-0.707107
0.088388	
-0.088388	

6/10 tap filter	
Low pass filter	High pass filter
0.018914	
0.006989	
-0.067237	0.129078
0.133389	0.047699
0.615051	-0.788486
0.615051	0.788486
0.133389	-0.047699
-0.067237	-0.129078
0.006989	
0.018914	

4 tap filter	
Low pass filter	High pass filter
0.176777	-0.176777
0.530330	-0.530330
0.530330	0.530330
0.176777	0.176777