Third Year Design Project SF2: 2020

Image Processing

1 Introduction

This project introduces you to some of the essential design tradeoffs which must be made during the design of image data compression systems. The main purpose of such systems is to compress as far as possible the size of the data file required to store an image (typically a real-world scene) while still preserving the quality of the decompressed image at an acceptable level. You will be introduced to techniques which to some extent reflect the compression inherent in the JPEG, JPEG2000 and JPEG-XR standards[1].

An image compression system normally comprises three main processes:

- An input filtering (or transformation) process, which compacts most of the energy of the image data into a relatively small number of filter output samples;
- A quantisation process, which represents these samples to some desired accuracy;
- A lossless entropy coding process, which codes the quantised samples into the minimum number of bits that still allows the samples to be recovered to their quantised accuracy in the decompressor.

[1] JPEG (Joint Photographic Experts Group) is the image compression standard from 1992 still commonly used today. JPEG2000 and JPEG-XR are modern versions which are gradually becoming more widespread.

The project introduces you to each of these processes in turn and allows you to make a number of inter-related design decisions. New concepts are introduced as the project progresses, rather than introducing too much theoretical material at the beginning. If you are unsure why you are doing any particular task, then please ask a demonstrator.

At the end of the project all groups will use their final design solutions to compress a small set of images to given file sizes, and the quality of the reconstructed images will be assessed in a competition to select the best design.

The final report of the project will be expected to summarise briefly all the investigations and pay particular attention to the design process as the solution evolves.

2 Project Organization

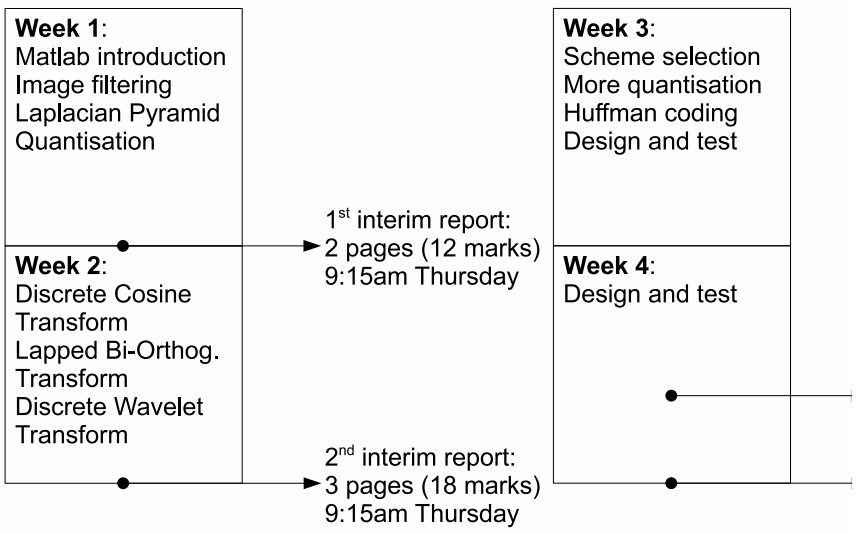


Figure 1: Overview of project

Important dates and deadlines are:

10am Thursday 7 May 2020 (Easter, week 3)	Project begins.
9:15am Thursday 14 May 2020 (Easter, week 4)	Hand in first interim report.
9:15am Thursday 21 May 2020 (Easter, week 5)	Hand in second interim report.
10am Monday 1 June 2020 (Easter, week 6)	Competition.
4pm Thursday 4 June 2020 (Easter, week 7)	Hand in final project report.

The rules below would have been those we adhered to if we were present in CUED:

****

During the project period, **8 hours per week** are **timetabled** when the lab is reserved for project use (9.00-11.00 Thursday, 14.00-18.00 Thursday and 11.00-13.00 Monday). Demonstrators will be available to give introductory talks, guidance and help during the morning sessions. Students are expected to attend **all** sessions — there is a **penalty of 1 mark per hour** or part thereof for missing the **morning** sessions.

****

Given the current circumstances, we will have online demonstrator sessions from 10:00-12:00 on Monday and Thursday mornings (BST) throughout the project period. Some extra sessions will also be organised. No penalties will be given for 'missing' a session, but an 'attendance record' should be kept.

Students will need to spend some additional **12 hours per week** per project **working on their own** (including report writing).

It is good practice to use a laboratory notebook to record *all* day-to-day activities, as a sketch book for conceptual design work, to record calculations etc.

This project requires 3 reports to be submitted, i.e. 2 interim reports and a final report. The maximum lengths for each report are shown below. There is a [possible] **penalty of 3 marks per day**, or part thereof, for late submission of **interim** reports. **Final reports must be handed in by 4pm Thursday 4 June 2020.**

First Interim Report (12 marks): $\leq$ **2 pages**, plus **2 pages** for appendices.

Second Interim Report (18 marks): $\leq$ **3 pages**, plus **3 pages** for appendices.

Final Report (50 marks): $\leq$ **9 pages**, plus **3 pages** for appendices.

In preparing reports, students are required to **adhere to the page limits**, however you may well want to include some pictures in appendices, which have an additional page limit. In any case, it is important to ensure that the pictures **as printed** actually demonstrate the features you are trying to convey in the report. It is often better to use zoomed-up cropped regions.

All reports are to be written individually. However, the compression schemes will be designed within groups of two, with a single entry per group in the final competition. Some groups also like to work closely throughout the project

Further details are provided elsewhere in this handout and in the document 'Third Year Project Guide'. However, note that the 'Third Year Project Guide' only suggests *typical* requirements: it is *this* document, the SF2 project handout, which you need to adhere to.

3 Key Design Stages

Students should carry out the following key design stages in order to achieve their final design solutions:

1. Familiarisation with Matlab and its image processing functions.
2. Experiments with 1-D and 2-D digital FIR filters to see their effects on images, and introduction to special techniques such as symmetric extension for dealing efficiently with finite data sets.
3. Development of the Laplacian Pyramid as an introduction to energy compaction. Different decimation and interpolation filters will be investigated, together with ways to allocate quantiser step-sizes to each pyramid level.

   The first interim report will be presented at this stage.
4. Development of the Discrete Cosine Transform (DCT) as a method of energy compaction. This is a standard block-based transform method and different transform sizes will be investigated.
5. Development of the Lapped Bi-orthogonal Transform (LBT) as an extension to the DCT. Different transform sizes and degrees of bi-orthogonality will be investigated.
6. Development of the Discrete Wavelet Transform (DWT), based on a tree of simple subband filters (LeGall filters). Different depths of tree will be investigated, together with ways to allocate quantiser step-sizes to each DWT level.

   The second interim report will be presented at this stage.
7. Selection of a small number (2 to 4) of preferred options for energy compaction for testing with the remainder of the system.
8. Experiments with quantisers. Different step sizes and widths of central clipping region will be investigated.
9. Development of combined run-length / Huffman coding for converting the quantised samples to a compact serial bit stream, loosely based on the JPEG standards.
10. Optimisation and test of a final design solution, based on all the information gathered above, and assessed on a set of test images.

    The final report will be presented at this stage.

You will find at many stages that a large number of permutations of the various design parameters are possible. The art or skill of good design is to investigate *only* the more promising options at each stage so that a near-optimal design is obtained without excessive experimental effort. A key aim of this project is to develop these skills. If in doubt, *please ask for help*!

4 Familiarisation with Matlab

Whether or not you are already familiar with Matlab, help can be found on the department's computing pages:

`http://www-h.eng.cam.ac.uk/help/tpl/programs/matlab.html`

The CUED guide *'Getting Started with Matlab'* is a good starting point.

The following instructions are for Windows. If you have a Mac or run Linux, you may need to adapt the following instructions slightly.

Use a web browser to visit the SF2 Moodle site where you will find a link to download the matlab files needed for this lab. Store these in any directory in your filespace.

Start up matlab and move to this directory.

Load the provided 'lighthouse' image using: **load lighthouse**

Now investigate the functions to display monochrome images as follows. Use **whos** (at the command prompt) or the workspace viewer to confirm that the 256×256 image is in **X** and there are two colour maps, **map** and **map2**.

Set the palette for 256 uniformly spaced grey levels using: **colormap(map)**
(note the American spelling!)

Display the image and set the axes for square pixels using: **image(X), axis image**

Change the palette to enhance the contrast using: **colormap(map2)**

map2 was generated for this image by a process known as histogram equalisation, which distorts **map** so that a histogram of the equalised mapping of the image intensities would be approximately flat over the full range of grey-scale from black to white (0 to 255).

You may also try pseudo colour palettes, with any of the following arguments for **colormap**: **hot, cool, bone, copper, pink, prism, jet, hsv,** etc.

Finally return to the correct palette **map**.

As an alternative to using the built-in Matlab function **image**, you may prefer to use a higher level function **draw** provided, which automatically generates the largest image of the correct aspect ratio which will fit in the figure window, and will also calculate an appropriate greyscale palette. Have a look at the file **draw.m** by typing **edit draw.m** to check that you understand what it does.

5 Simple image filtering

An effective image lowpass filter, of odd length N, may be obtained by defining the impulse response $h(n)$ to be a sampled half-cosine pulse:

$$h(n) = G\cos\left(\frac{n\pi}{N+1}\right) \qquad \text{for } \frac{-(N-1)}{2} \le n \le \frac{N-1}{2}$$

where G is a gain factor, which, in order to give unity gain at zero frequency, should be calculated such that

$$\sum_{n=-(N-1)/2}^{(N-1)/2} h(n) = 1$$

(This may be done most easily by first calculating $h(n)$ with $G = 1$, summing all terms, and then dividing them all by the result.)

Take a look at the Matlab function (M-file) **halfcos.m** and check that it generates h for a given N.

Use the **conv** function in a **for** loop to convolve a 15-sample half-cosine with each row of the test image, Lighthouse. Observe the resulting image **Xf** and note the increased width and the gradual fade to black at the edges, caused by **conv** assuming the signal is zero outside the range of the input vectors.

Trim the filtered image **Xf** to its correct size and display it using:

draw(Xf(:,[1:256]+7))

Note that darkening of the sides is still visible, since the lowpass filter assumes that the intensity is zero outside the image.

Image trimming and convolution of *all* the image rows can also be achieved using the **conv2** function with the **'same'** argument:

Xf = conv2(1, h, X, 'same'); draw(Xf);

Symmetric extension is a technique to minimise edge effects when images of finite size are filtered. It assumes that the image is surrounded by a flat mirror along each edge so it extends into mirror-images (symmetric extensions) of itself in all directions over an infinite plane. If the filter impulse response is symmetrical about its mid point, then the filtered image will also be symmetrically extended in all directions with the same period as the original images. Hence it is only necessary to define the filtered image over the same area as the original image, for it to be defined over the whole infinite plane.

Let us consider a one-dimensional example for a 4-point input signal a, b, c, d. This may be symmetrically extended in one of two ways:

$$\ldots d, c, b, \underbrace{a, b, c, d,}_{\text{original}} c, b, a \ldots \qquad \text{or} \qquad \ldots d, c, b, a, \underbrace{a, b, c, d,}_{\text{original}} d, c, b, a \ldots$$

The left-hand method, where the end points are not repeated at each boundary, is most suitable when the signal is to be filtered by a filter of odd length. The other method is most suited to filters of even length.

In Matlab a matrix with symmetrically extended rows may be generated by extending just the indexing of its columns as in:

ind = [3 2 [1:n] n-1 n-2];
Xe = X(:,ind);

The rows of **Xe** are the rows of **X** extended by 2 samples at each end using the left-hand method above. The M-file **convse.m** make use of this to filter the rows of matrix **X** using the appropriate form of symmetric extension. The filtering is performed by accumulating shifted versions of **X** in **Xe**, each weighted by the appropriate element of **h**. Check that you understand how this function works.

Use **convse** to filter the rows of your image with the 15-tap half-cosine filter, noting the absence of edge effects.

Now filter the columns of the *row-filtered* image by use of the Matlab transpose operator **'**. Note that the function **conv2se** has also been provided, as a symmetrical-extension replacement for **conv2**.

Does it make any difference whether the rows or columns are filtered first? (You should test this accurately by measuring the maximum absolute pixel difference between the row-column and column-row filtered images. Beware of scientific notation, used by Matlab for very small numbers!)

This process of separate row and column filtering is known as *separable* 2-D filtering, and is much more efficient than the more general non-separable 2-D filtering.

It is possible to construct a 2-D *high-pass* filter by subtracting the 2-D low-pass result from the original. Note that your 2-D lowpass filter **h** *must* have a DC gain (sum of all filter coefficients) of unity for this to correctly produce a highpass filter. The highpass image **Y** now contains negative, as well as positive pixel values, so it is sensible to display the result using **draw(Y)** which automatically compensates for this.

Try generating both low-pass and high-pass versions of **X** using a range of different odd-length half-cosine filters. Comment on the relative effects of these filters.

One way to assess sets of filtered images like these is to contrast the *energy* content. In this context, the energy **E** of an image **X** is given by the sum of the squares of the individual pixel values:

E = sum(X(:).^2);

X(:) converts **X** into a vector, and **.^2** squares individual components.

What do you observe about the energy of the highpass images, compared with that of the lowpass images?

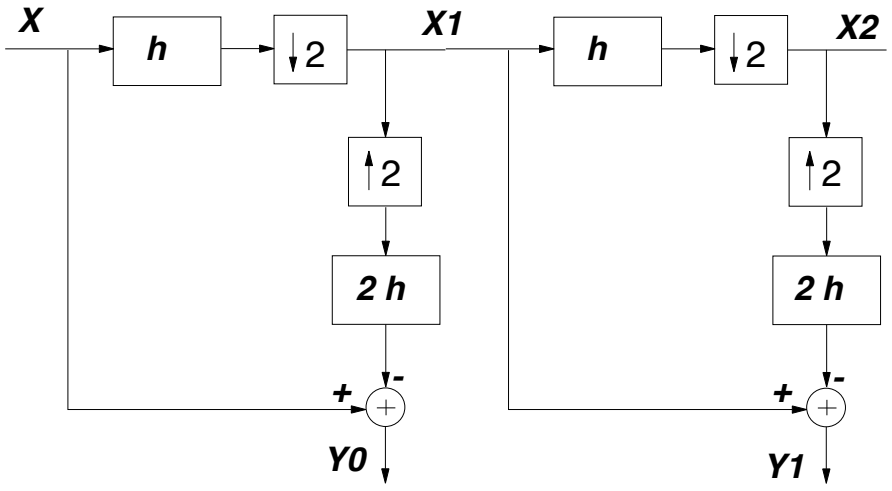


Figure 2: A typical laplacian pyramid with 2 levels: forward (analysis) part only

6 The Laplacian Pyramid

6.1 The basic concept

The Laplacian pyramid is an energy compaction technique, based on the observation:

> *For most real-world images, the high-frequency energy is much less than the low-frequency energy.*

Since the lowpass image is much lower bandwidth than the original image, it can be subsampled (*decimated*) 2:1 in both horizontal and vertical directions, without significant loss of information to give a quarter-size lowpass image. We have provided a row filter/decimation function, **rowdec.m**, to filter and decimate the rows of a matrix by 2:1. Use this twice (on the image and its transpose) to generate a quarter-size lowpass image **X1** of Lighthouse. For simplicity use a 3-tap (length 3) filter **h** with coefficients: $\frac{1}{4}$[1 2 1].

The image **X1** can be interpolated 2:1 in each direction to generate a lowpass image of the original size, which can then be subtracted from the original to yield a full-size highpass image. We have also provided a row interpolation/filter function, **rowint.m**, which doubles the length of each row of the image, by inserting zeros between alternate samples and then lowpass filtering the result with the filter **2h**. This is the same filter except for the DC gain of 2 which is needed to overcome the process of decimation. Apply this twice (as before) to generate a full-size lowpass image, and subtract this from **X** to give a highpass image **Y0**. Display **X1** and **Y0** to see these effects.

The highpass images will always have approximately zero mean (since any dc component is removed). Hence for the remainder of this project, we recommend that you also make all

lowpass images be approximately zero mean by subtracting 128 from them before you start any processing. This makes the 8-bit pixel values cover the range −128 to 127, instead of 0 to 255. The advantages of having a zero-mean input image are not very obvious here, but they will become clearer as the project progresses. If you use **draw** the images will still be displayed correctly.

Examine the files **rowdec.m** and **rowint.m** and check that you understand how they work. Decimation is achieved by selecting every second element of a vector using **y = x[1:2:N];**, and filtering is performed with symmetric extension as in **convse.m**, except that we do not bother to calculate the filter outputs that are to be discarded by the decimation process. Interpolation is achieved by loading every second element of a double-size vector using **x = zeros(N,1); x[1:2:N] = y;** . The intermediate samples of the interpolated vector x must be zero before the vector is passed through the interpolation lowpass filter.

If the small lowpass image **X1** and the full-size highpass image **Y0** are transmitted to a distant decoder, then the decoder can exactly reconstruct the original image by interpolating **X1** up to full size and adding in **Y0** (which represents the error between the original and the interpolated **X1**). We have achieved image compression if **X1** and **Y0** can be transmitted with fewer bits than **X**. Usually this will be the case because **Y0** contains so much less energy than **X**, and **X1** is only one quarter of the size of **X**. However we do start at a disadvantage because there are 25% more samples to code. Many of the **Y0** samples may be represented by zero, and we shall show later that runs of zeros may be coded with relatively few bits.

The quarter-size lowpass image **X1** may be further subsampled, using the same process as was applied to **X**, so that it may be transmitted as a one-sixteenth-size lowpass image **X2** and a quarter-size highpass image **Y1**. This usually achieves further data compression and may be repeated as many times as is desired (until, for typical images, no further compression is achieved). This leads to a pyramid of highpass images and a final tiny lowpass image. Usually three or four layers of the pyramid are sufficient to give maximum compression.

Write an M-file **py4enc.m** to generate a 4-layer pyramid, so that **X** is split into four highpass images, **Y0 Y1 Y2 Y3**, each a quarter of the size of its predecessor, plus a tiny lowpass image **X4**, which is a quarter of the size of **Y3**. These images may be displayed side-by-side using the function provided, **beside.m**:

draw(beside(Y0,beside(Y1,beside(Y2,beside(Y3,X4)))))

Get a demonstrator to check that your images look correct, and then write another M-file **py4dec.m** to decode **X4** and **Y3 Y2 Y1 Y0** into a set of lowpass images **Z3 Z2 Z1 Z0**. (**Z3** is obtained by interpolating **X4** and adding **Y3**, and then **Z2** is obtained from **Z3** and **Y2**, and so on.) If all is correct, **Z0** should be identical to **X**. You can check that your M-file is correct by calculating **max(abs(X(:) - Z0(:)))** and also displaying your pyramid of decoded images, **Z3** to **Z0**.

For further information on the Laplacian Pyramid see Burt and Adelson [IEEE Trans. on

Communications, 1983, vol 31, no 4, pp 532-540, "The Laplacian Pyramid as a compact image code"].

6.2 Quantisation and Coding Efficiency

To see whether data compression is possible using the above pyramid decomposition, we must calculate the approximate number of bits required to code the image pyramid. This may be done using the *entropy* of the quantised image data.

The entropy of a single data sample, which may randomly take one of Q possible quantised values such that each value $q(i)$ has a probability $p(i)$ of being in state i for $i = 1$ to Q, is given by:

$$\text{Entropy (bits/sample)} = \sum_{i=1}^{Q} p(i)\ \log_2(1/p(i)) = -\sum_{i=1}^{Q} p(i)\ \log_2(p(i))$$

The entropy represents the minimum average number of bits per sample needed to code samples with the given probability distribution $p(i)$, assuming that an ideal variable-length entropy code is used, and that the samples are uncorrelated with each other. Arithmetic codes can get arbitrarily close to this bit rate, and simpler Huffman codes can also get very close with many typical signals (you will be using Huffman codes later on). It is possible to code signals at bit rates less than the entropy, if the samples are correlated, but for simplicity we shall ignore this here.

To demonstrate the validity of the above formula, first consider a signal with 8 quantised values of equal probability $p(i) = \frac{1}{8}$ for all i. The entropy is then $8 \times \frac{1}{8} \times \log_2(8) = 3$ bits per sample, as expected.

Now consider a signal with only 3 values, with probabilities $p(1) = \frac{1}{2}$ and $p(2) = p(3) = \frac{1}{4}$. The entropy is then $\frac{1}{2}\log_2(2) + 2 \times \frac{1}{4}\log_2(4) = 1.5$ bits per sample. This is consistent with using a single bit '0' to represent state 1, and two bits, '10' and '11', to represent states 2 and 3.

The M-function **bpp(X)** has been written to calculate the entropy in bits per pixel of an image matrix **X**. First the function computes a histogram of **X** to determine the probabilities $p(i)$ and then it calculates the entropy, using the above formula.

In the Laplacian Pyramid, the total number of bits is obtained by multiplying each of the sub-image entropies by the number of pixels in each corresponding sub-image. However, in order to compress the data, we also need to quantise the images. We have also provided a function **quantise(X,step)** which will quantise **X** in steps centred on integer multiples of **step**. Hence **bpp(quantise(X,step))** will return the entropy of image **X** quantised in steps of **step**.

Calculate the entropies of images **X X1 Y0** and hence the total numbers of bits to encode **X**, or **X1** and **Y0**, when quantised to a step size of 17 (which gives 15 distinct grey levels if applied to a lowpass image with intensities from -127 to 127). Find the data compression for this simple one-stage pyramid, and then investigate the improvements from using more layers.

Since compressing an image will generally result in a reduction in quality, we also need a way to measure this quality reduction. It is actually quite hard to find a quality measure which matches individual perceptions of how an image has been changed due to compression, and for that reason it is important to always judge and comment on an image visually. However we also need a quantitative measure, and the most obvious is the rms error (standard deviation) between the input and compressed image (i.e. using **std(X(:) - Z(:))**), where **Z** is the compressed image.

Quantise the Laplacian Pyramid with a step size of 17, and reconstruct the output image from the decoding pyramid. Look at the visual features, and calculate the rms error (standard deviation) between the input image and the decoding pyramid output image. Repeat this for schemes with more layers in the pyramid.

Note that we call this error the rms error, but in fact we calculate the standard deviation, which only equals the true rms error if the mean error is zero. However the eye is very insensitive to small errors in the mean level of images, so the standard deviation (which ignores the mean) is a better measure of image quality.

Quantise the original image with the same step size (17) and note the visual features and rms error. Compare these to the results from the pyramid scheme above. Why are the rms errors larger in the pyramid scheme?

Comparisons of the number of bits with different coding strategies are only valid if they result in approximately the same image quantisation error. Write a function which will optimise the step size (resulting in a non-integer value) in the Laplacian scheme until the rms error is the same as for direct quantisation; you will find this optimisation useful for later investigations too.

Investigate what step size of the quantisers for the pyramid scheme you need, in order to get approximately the same error as for direct quantisation at a step size of 17.

In many of your results from now on you will need to express the performance of your algorithms in terms of *compression ratio*, which is normally defined as:

$$\text{Compression Ratio} = \frac{\text{Total bits for reference scheme}}{\text{Total bits for compressed scheme}}$$

Usually the *reference* scheme is the direct pixel quantisation method with its quantiser adjusted to give the same rms error as the scheme being evaluated (the *compressed* scheme). For good schemes we try to make the compression ratio as large as possible.

We now investigate the effect of using different step sizes for the different levels of the pyramid. There are many schemes for varying the step sizes between different levels: in this project we shall look at the *equal MSE* criterion. In the *equal MSE* scheme, step sizes

are chosen such that quantisers in each layer contribute equally to the Mean Squared Error of the reconstructed image. In general the step sizes will depend on the image signal being coded. However this can be achieved approximately by choosing a separate step size for each layer such that:

- A single impulse of that step size will give a filtered pulse in the reconstructed image which has the same energy, whichever layer of the decoder the impulse excites.

Impulse Response Measurement: Investigate the effect of a single impulse in a particular layer (eg. **Y0,Y1** etc.) as it appears in the reconstructed image **Z0**. This can be done by first generating a test pyramid image, which is zero everywhere. Then place an impulse (e.g. of amplitude 100) in the centre of one layer, reconstruct the entire pyramid to give **Z0**, then measure the total energy of **Z0**. If this is repeated for each layer, you will have measured how much energy a fixed size impulse at each layer contributes to the decoded image.

We actually want to arrange for the impulse sizes to vary and the energy to stay the same. Therefore the impulse sizes (and hence chosen quantisation steps) we require in each level will be inversely proportional to the square root of the energies measured above. The important result is the *ratio* of the step sizes between layers. If this ratio is maintained for any overall quantiser scaling, we will get an (approximately) *equal MSE* scheme.

Find new values for the data compression achievable when all schemes (i.e. constant step size or equal MSE, both with varying layer depth) produce the same rms error between the decoded image and the original image. Comment on the differences in compression, visual quality and rms error, and the optimum choice of layer depth in each case.

6.3 Changing the decimation / interpolation filter

It is also worth investigating whether a more complicated decimation / interpolation filter **h** can improve the compression. The z-transfer function of the filter used so far is:

$$h(z) = \left(\frac{1 + z^{-1}}{2}\right)^m$$

where $m = 2$. If m is increased to 4 (so the number of taps remains odd), the vector representation of **h** is $\frac{1}{16}[1\ 4\ 6\ 4\ 1]$. This filter has a lower cut-off frequency than when $m = 2$, so you should find that each interpolated lowpass image in the pyramid is a little more blurred, and there is a little more energy left in each highpass image.

Optimise the step sizes for the pyramid with this new filter and determine the new compression performance and visual features.

6.4 First Interim Report

Discuss and explain your results, gathered so far, in your first interim report. Try to answer the questions posed in the above text. Be brief where things are straightforward, but pay more attention to detail in areas where you think something interesting is happening. Write this as a standalone report, not as a series of bullet points answering the questions posed.

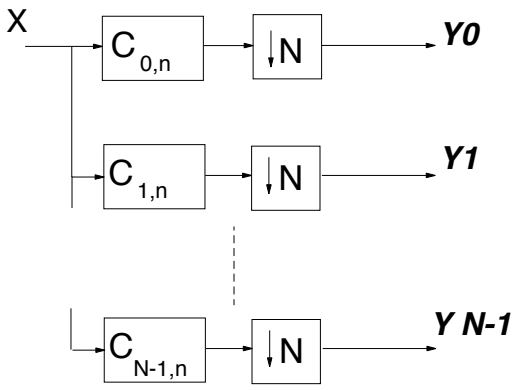


Figure 3: A DCT can be treated as an N channel filter bank where the coefficients of the filters are the basis functions.

7 The Discrete Cosine Transform (DCT)

The DCT is a method of performing energy compaction that is rather different from the pyramid method. It operates on non-overlapping blocks of pixels (typically 8×8 pixels in size) by a reversible linear transform process, such that each block of pixels is replaced by a block of the same number of transform coefficients. If all the transform coefficients for a given block are transmitted unaltered to the decoder, then the original block of pixels can be exactly recovered by the inverse transform process.

In practise the transform coefficients are quantised before transmission, and if energy compaction has occurred, then fewer bits will be needed to send the coefficients than the original pixels. A key advantage of transform-based methods is that there is no expansion of the number of samples (the transformed block is the same size as the original block of pixels), whereas the previous pyramid method expands the data by $1 + \frac{1}{4} + \frac{1}{16} + \ldots \approx 1.33$ times, which is not very desirable for data compression.

7.1 Definition of the DCT

The one-dimensional form of the DCT is closely related to the Discrete Fourier Transform (DFT). The 1-D N-point DCT[2] is defined as follows:

$$\begin{aligned} y(k) &= \sum_{n=0}^{N-1} C_{kn}\, x(n) \qquad \text{for } \; 0 \le k \le N-1 \\ \text{where } \; C_{0n} &= \sqrt{\tfrac{1}{N}} \\ \text{and } \; C_{kn} &= \sqrt{\tfrac{2}{N}}\, \cos\frac{k(n+\frac{1}{2})\pi}{N} \qquad \text{for } \; 1 \le k \le N-1 \end{aligned}$$

[2]This is actually the Type-II DCT, and the inverse is the Type-III DCT - other types have slightly different relative phases

The equivalent inverse DCT is:

$$x(n) = \sum_{k=0}^{N-1} C_{kn}\, y(k) \qquad \text{for } 0 \le n \le N-1$$

where C_{kn} is defined as above.

We see that the forward transform is equivalent to multiplication of the N-point column vector $[x(0) \ldots x(N-1)]'$ by an $N \times N$ matrix, containing C_{kn} at each location (k, n), to produce the N-point column vector $[y(0) \ldots y(N-1)]'$. Similarly the inverse transform is equivalent to multiplication of the y vector by the transpose of the C matrix to give the x vector. In Matlab notation these become:

$$\mathbf{y} = \mathbf{C} * \mathbf{x} \qquad \text{and} \qquad \mathbf{x} = \mathbf{C}' * \mathbf{y}$$

Note that **C** is an orthonormal matrix since its inverse is just its transpose (its rows are othogonal to each other and have unit energy).

The two-dimensional version of the DCT (as used for image compression) is a simple extension of the above 1-D DCT. For an $N \times N$ block of pixels, the N-point 1-D DCT is first applied to each column of the block to give N columns of coefficients. Then the same 1-D DCT is applied to the rows of these coefficients to give the 2-D transform coefficients.

In Matlab notation, if the input block of pixels is matrix **X**, the output block of 2-D transformed coefficients **Y** is given by:

$$\mathbf{Y} = (\mathbf{C} * (\mathbf{C} * \mathbf{X})')' \qquad \text{or more simply} \qquad \mathbf{Y} = \mathbf{C} * \mathbf{X} * \mathbf{C}'$$

where **C** is the 1-D transform matrix as above. Note that in the 2-D transform, it does not matter whether the rows or the columns are transformed first (because the transform is linear and separable).

7.2 Applying the DCT to images

Conceptually the 2-D DCT is applied to all non-overlapping $N \times N$ blocks of pixels in an image (we assume that the image dimensions are exact multiples of N). However in Matlab, it is simplest and most efficient to perform 1-D N-point DCTs on all the columns of the image first, and then repeat the operation on the transpose of the result to transform the rows.

First generate an 8-point 1-D Type-II DCT matrix **C8** using: **C8 = dct_ii(8);**

Take a look at the function **dct_ii** and list **C8** to check that it agrees with the definitions for C_{kn} given above. Plot the rows of **C8** using **plot(C8')**. When we calculate the 1-D transform of an 8-point block of data, each transform coefficient represents the component of the data that is correlated with the corresponding row of **C8**. Hence the first coefficient represents the dc component, the second one represents the approximate average slope,

and so on. The later coefficients represent progressively higher frequency components in the data.

The function **colxfm(X,C8)** will perform a 1-D transform on the columns of image **X** using **C8**. We can therefore perform a 2-D transform on **X** by using **colxfm** twice, once with transpose operators, as follows:

Y = colxfm(colxfm(X,C8)',C8)';

In **Y**, each 8×8 block of pixels has been replaced by an equivalent block of transform coefficients. The coefficient in the top left corner of each block represents the dc value of the block of pixels; coefficients along the top row represent increasing horizontal frequency components, and along the left column represent increasing vertical frequency components. Other coefficients represent various combinations of horizontal and vertical frequencies, in proportion to their horizontal and vertical distances from the top left corner.

If we try to display **Y** directly as an image, it is rather confusing because the different frequency components of each block are all present adjacent to each other. A much more meaningful image is created if we group all the coefficients of a given type together into a small sub-image, and display the result as an 8×8 block of sub-images, one for each coefficient type. The function **regroup(Y,N)** achieves this regrouping, where N is the size of the original transform blocks. You need to ensure that **X** has zero mean (by subtracting 128) before you start transforming it, otherwise the dc coefficient will be purely positive, whereas the others are symmetrically distributed about zero. Also, an $N \times N$ 2-D DCT introduces a gain factor of N in order to preserve constant total energy between the pixel and transform domains: we need to divide by N *when displaying* to get back to the expected range.

Hence we can display **Y** meaningfully using:

N = 8; draw(regroup(Y,N)/N)

In this image, you should see a small replica of the original in the top left corner (the dc coefficients), and other sub-images showing various edges from the original, representing progressively higher frequencies as you move towards the lower right corner.

What do you observe about the energies of the sub-images as frequencies increase?

Now check that you can recover the original image from **Y** by carrying out the inverse transform using:

Z = colxfm(colxfm(Y',C8')',C8');

Measure the maximum absolute error between **X** and **Z** to confirm this.

The DCT analyses each 8×8 block of image pixels into a linear combination of sixty-four 8×8 basis functions. The following will generate an image comprising these basis functions (the zeros separate the sub-images, and the (:) operator converts from a matrix to a vector):

bases = [zeros(1,8); C8'; zeros(1,8)];
draw(255*bases(:)*bases(:)');

Display this image and explain how it relates to the DCT coefficients.

7.3 Quantisation and Coding Efficiency

We are now going to look at the effects of quantising the DCT coefficients fairly coarsely and determine the entropies of the coefficient sub-images. At this stage we shall quantise all sub-images with the same step-size, since they all are the same size and have unit energy gain from the quantiser to the output image (due to the orthonormal transform matrices).

First quantise the transformed image **Y** using a step size of 17 to give **Yq**. Then regroup **Yq** to form sub-images of each coefficient type as before, to give **Yr**. These sub-images have different probability distributions and we can take advantage of this later in coding them efficiently. Hence we get a better estimate of the number of bits required to code **Yq** by looking at the entropies of each of the re-grouped sub-images separately.

Write a function **dctbpp(Yr, N)** to calculate the total number of bits from a re-grouped image **Yr**, by using **bpp(Ys)** on each sub-image **Ys** of **Yr**, then multiplying each result by the number of pixels in the sub-image, and summing to give the total number of bits.

Visualise **Yr** and comment on the distributions in each of the sub-images. Use the function **dctbpp(Yr, N)** that you have written to calculate the total number of bits, and compare it with just using **bpp(Yr)**, explaining your results.

Now reconstruct the output image **Z** from **Yq** and measure the rms error (standard deviation) between **X** and **Z**. Compare this with the error produced by quantising **X** with a step-size of 17 to give **Xq**.

As with the Laplacian Pyramid, we really need to contrast compression ratios and visual results on compressed images with the same rms error. Re-use your step optimisation code to calculate the (non-integer) step size required in this case for the same rms error as quantising **X** with a step-size of 17.

Calculate the compression ratio for this scheme compared to direct quantisation. Use **dctbpp** to calculate the number of bits needed. Contrast the visual appearance of the DCT-compressed image, the directly quantised image, and the original image.

7.4 Alternative transform sizes

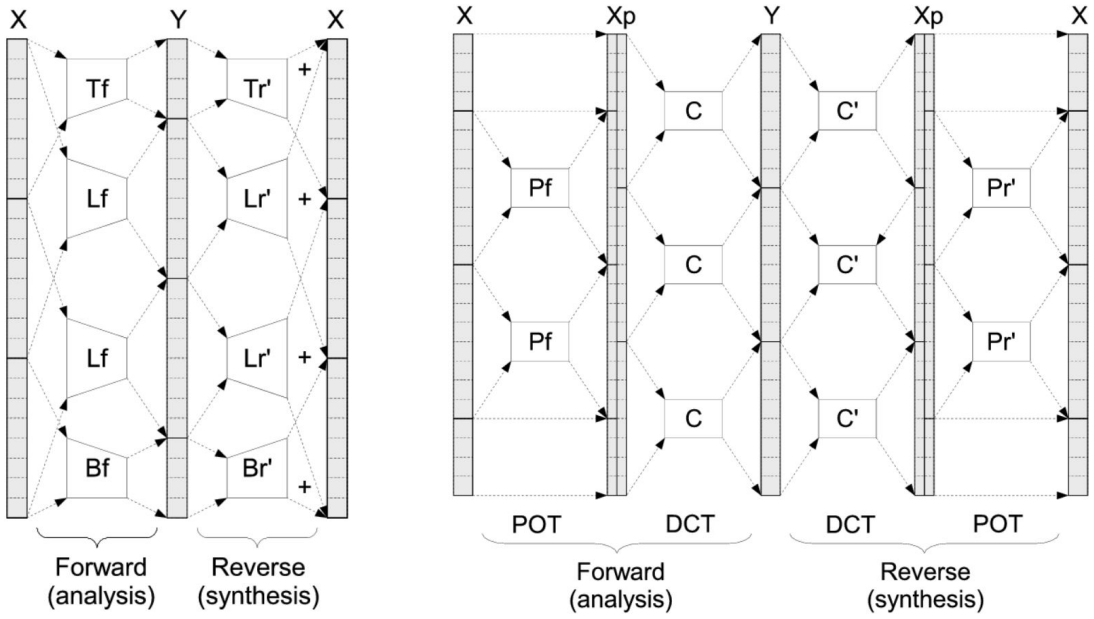
So far, we have concentrated on 8×8 DCTs using **C8** as the 1-D transform matrix. Now generate 4-point and 16-point transform matrices, **C4** and **C16** using **dct_ii**.

Repeat the main measurements from the previous section, so as to obtain estimates of the number of bits and compression ratios for 4×4 and 16×16 DCTs when the rms errors are equivalent to those in your previous tests. Also assess the relative subjective quality of the reconstructed images.

This analysis is in fact slightly biased because with larger transform sizes the function **dctbpp(Yr, N)** will use a greater number of smaller sub-images on which to calculate probability distributions. It may be better to use the same **N** in this function even when the actual transform changes; however whether this is more predictive of actual coding performance depends on what scanning method is used in the coding scheme.

What happens in the limit if you use **dctbpp(Yr, 256)** (i.e. the entropy is calculated independently for each pixel)? Why is this the case, and why isn't this a realistic result?

Can you draw any conclusions about the best choice of transform size for the Lighthouse image? Try to postulate what features in other images might make your conclusions different, and suggest why.



(a) Lapped Bi-orthogonal Transform (b) Photo Overlap Transform and a DCT

Figure 4: (a) An LBT transforms overlapping sections of $\mathbf{X}$ to create $\mathbf{Y}$. (b) In some cases this can be interpreted as pre-filtering with a POT, followed by a DCT.

8 The Lapped Bi-orthogonal Transform (LBT)

One of the difficulties with the DCT is that it processes each block separately and hence does not take advantage of any correlation between blocks. A possible solution to this is to use a *Lapped Bi-orthogonal Transform* (LBT). These transform overlapping blocks in $\mathbf{X}$ to generate smaller non-overlapping blocks in $\mathbf{Y}$. In the left-hand figure above, 16 values in $\mathbf{X}$ are used to generate each set of 8 values in $\mathbf{Y}$.

LBTs are quite complicated to derive and analyse: however one of the most popular forms[3] can also be represented as a pre-filtering operation before performing the DCT described in the previous section. In this case the pre-filtering (or post-filtering for the reverse operation) is sometimes known as a *Photo Overlap Transform* or POT. A POT followed by a DCT is then equivalent to a particular type of LBT.

The right-hand figure above demonstrates this. The POT is first performed on a section of data $\mathbf{X}$, *shifted by* $\mathbf{N/2}$, so that it runs across the block boundaries of the subsequent DCT. Ignoring this block shift for a moment, the forward operation for a 2D image $\mathbf{X}$ is:

$$\mathbf{Y} = \mathbf{C} * \mathbf{Pf} * \mathbf{X} * \mathbf{Pf}' * \mathbf{C}' \qquad \text{or in reverse} \qquad \mathbf{X} = \mathbf{Pr}' * \mathbf{C}' * \mathbf{Y} * \mathbf{C} * \mathbf{Pr}$$

[3]The type-II fast lapped (bi-)orthogonal transform, or LOT-II

8.1 Applying the LBT to images

The pre-filtering $\mathbf{Pf} * \mathbf{X} * \mathbf{Pf}'$ (with the correct block shift) is straightforward in Matlab:

```
t = [(1+N/2):(I-N/2)]; % N is the DCT size, I is the image size
Xp = X; % copy the non-transformed edges directly from X
Xp(t,:) = colxfm(Xp(t,:), Pf);
Xp(:,t) = colxfm(Xp(:,t)', Pf)';
```

This is followed by the DCT **C** as before. In the reverse operation, the inverse DCT **C'** is performed first, followed by **Pr'**:

```
Zp = Z; % copy the non-transformed edges directly from Z
Zp(:,t) = colxfm(Zp(:,t)', Pr')';
Zp(t,:) = colxfm(Zp(t,:), Pr');
```

We have provided a function **pot_ii(N, s)** which will generate a forward (pre-filtering, **Pf**) and reverse (post-filtering, **Pr**) matrix of size **N** with scaling factor **s**. Edit your code for performing DCT analysis so that it can pre-filter **X** with **Pf** before the forward DCT, then post-filter **Z** with **Pr** after the inverse DCT. Confirm that, without quantisation, this correctly recreates the original image, i.e. **Zp** = **X**. Use **[Pf Pr] = pot_ii(N)** with the default scaling value **s**.

8.2 Quantisation and coding efficiency

The scaling factor **s** determines the *degree of bi-orthogonality*. If $\mathbf{s} = 1$ then **Pf** is the same as **Pr**, otherwise $1 < \mathbf{s} < 2$ weights the relative contributions of **Pf** and **Pr** un-equally.

For an 8×8 DCT, try implementing an LBT with POT scaling factors varying from 1 to 2 ($\sqrt{2}$ is often a good choice). In each case find the quantisation step which makes the rms error match the directly quantised image. Note the compression ratios and find the scaling factor which maximises these. Also note the visual features in these images.

The POT can often improve both compression and block smoothing, since the pre-filter acts to reduce correlations between each DCT sub-block, whilst the inverse post-filter acts to remove the discontinuities between sub-blocks. This is rather different from the operation of the DCT. Investigate this by looking at the basis functions, as you did with the DCT:

```
bases = [zeros(1,8); Pf'; zeros(1,8)];
draw(255*bases(:)*bases(:)');
```

Look at both these bases and the pre-filtered image **Xp**, using different scaling factors **s**, and comment on the visual effect of varying these scaling factors. You may need to multiply **Xp** by up to 0.5 to display it better.

With this type of POT / DCT combination it is common to use smaller DCT block sizes but to code several blocks together. Hence a more accurate estimate of the number of bits

is found by always using 16×16 blocks, i.e. regroup **Yq** with the correct size **N** to give **Yr**, but then always use **dctbpp(Yr, 16)**.

Investigate the relative visual and compression performance of LBTs with 4×4, 8×8 and 16×16 blocks, using the scaling factor you have previously selected. As before, be careful to match the rms error with a directly quantised image.

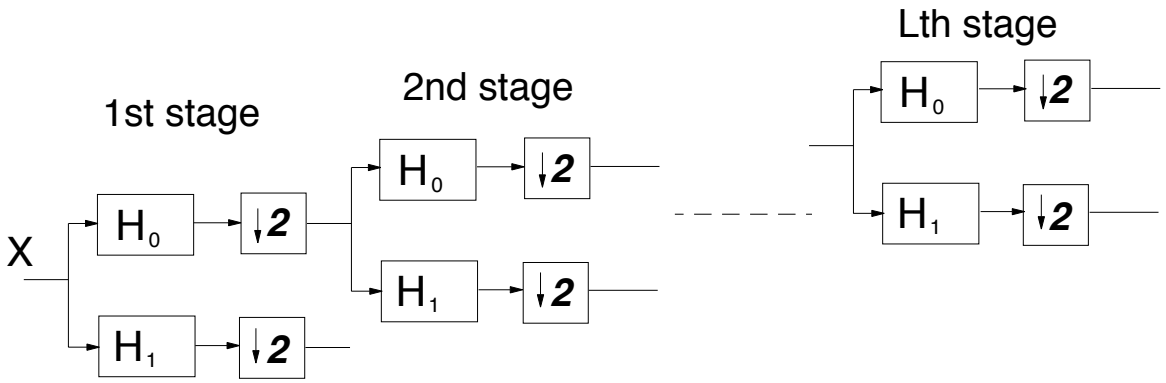


Figure 5: An L level binary discrete wavelet transform.

9 The Discrete Wavelet Transform (DWT)

The final method of energy compaction that we shall investigate, is the discrete wavelet transform. In some ways this attempts to combine the best features of the Laplacian pyramid and the DCT:

- Like the pyramid, the DWT analyses the image at a range of different scales (levels) and employs symmetrical filters;
- Like the DCT, the DWT avoids any expansion in the number of coefficients.

Wavelet theory was evolved by mathematicians during the 1980's. As with the LBT, we shall not attempt to teach this theory here, just illustrate a relatively simple form of it.

Wavelets are short waveforms which are usually the impulse responses of filters. Wavelet transforms employ banks of bandpass filters, whose impulse responses are scaled versions of each other, in order to get pass-bands in different parts of the frequency spectrum. If the impulse response of a filter is scaled in time by a factor a, then the filter frequency response is scaled by the factor $1/a$. Typically $a = 2$ from one filter to the next, and each bandpass filter is designed to pass a 2:1 range of frequencies (one octave). We can split an image up using wavelets by a process known as a *binary wavelet tree*.

9.1 The binary wavelet tree

We start in 1-D with the simplest possible pair of filters, operating on just two input samples, x_n and x_{n-1}. The two filter outputs, u_n and v_n at time n are given by:

$$u_n = \tfrac{1}{2}(x_n + x_{n-1}) \qquad \text{and} \qquad v_n = \tfrac{1}{2}(x_n - x_{n-1})$$

The first filter averages adjacent samples, and so rejects the higher frequency components of x, while the second filter differences these samples, and so rejects the lower frequency

components. These filters are known as the *analysis* filter pair, $H_1(z) = \frac{1}{2}(1 + z^{-1})$ and $H_2(z) = \frac{1}{2}(1 - z^{-1})$. It is clear that we can recover the two input samples from the filter outputs using:

$$x_n = u_n + v_n \qquad \text{and} \qquad x_{n-1} = u_n - v_n$$

Next it is important to note that we need only retain the samples of u_n and v_n at even values of n in order to be able to recover all the original samples of x. Hence u and v may be decimated 2:1 and still allow perfect reconstruction of x. If x is a finite length vector (e.g. a row of image pixels), then u and v are each half as long as x, so the total number of samples is preserved by the transformation.

A wavelet binary tree may be constructed using these filters, by using an identical pair, H_1 and H_2, to filter the decimated lowpass signal u_{2n}, to give a pair of outputs, uu_{2n} and uv_{2n}, representing the lower and upper halves of the first low band. These may again be decimated 2:1 and still permit perfect reconstruction of u. This process may be continued as often as desired: each time splitting the lowest band in two, and decimating the sample rate of the filter outputs by 2:1. At each stage the bandwidth of the two lowest filters is halved, and their impulse responses are doubled in length. The total number of output samples remains constant, however many stages are used.

For example, if f_s is the input sample rate, a 3-stage binary tree will split the input signal bandwidth of 0 to $f_s/2$ into the following four bands:

$$0 \to f_s/16; \quad f_s/16 \to f_s/8; \quad f_s/8 \to f_s/4; \quad f_s/4 \to f_s/2.$$

The very simple filters, given above, do not generate a filter tree with good characteristics, since the wavelets turn out to be just a pair of square pulses. These generate *blocking* artefacts when used for image compression (in fact they are equivalent to the 2 point ($N = 2$) DCT). A better set of filters are the LeGall 5 and 3 tap pair, given by:

$$u_n = \tfrac{1}{8}(-x_{n+2} + 2x_{n+1} + 6x_n + 2x_{n-1} - x_{n-2}) \quad \text{and} \quad v_{n+1} = \tfrac{1}{4}(-x_{n+2} + 2x_{n+1} - x_n)$$

If u and v are decimated by 2 by choosing even n only, the lowband outputs u_n are centred on the even samples, and the highband outputs v_{n+1} are centred on the odd samples. This is very important to allow perfect reconstruction of x from u and v.

The equations for reconstruction may be obtained by solving the above to get:

$$\begin{aligned} x_n &= \tfrac{1}{2}(-v_{n+1} + 2u_n - v_{n-1}) \qquad \text{and} \\ x_{n+1} &= \tfrac{1}{2}(x_{n+2} + 4v_{n+1} + x_n) = \tfrac{1}{4}(-v_{n+3} + 2u_{n+2} + 6v_{n+1} + 2u_n - v_{n-1}) \end{aligned}$$

In general, most analysis filters will not yield such simple reconstruction solutions, and the design of suitable filters is a non-trivial topic that we shall not cover here.

9.2 Applying the DWT to images

As with the DCT, the 2-D DWT may be obtained by applying a 1-D transform to first the rows and then the columns of an image.

Start by loading the Lighthouse image and defining the two LeGall filters given above:

h1 = [-1 2 6 2 -1]/8;
h2 = [-1 2 -1]/4;

We can use the function **rowdec** from the pyramid work, to produce a decimated and lowpass filtered version of the rows of **X** (remembering to subtract 128 as before) using:

U = rowdec(X,h1);

To get the high-pass image **V**, it is important to align the decimated samples with the odd columns of **X** (assuming the first column is $n = 0$) whereas **U** is aligned with the even columns. To do this we use a slightly modified version of **rowdec.m**, called **rowdec2.m**.

V = rowdec2(X,h2);

Display [**U V**] to see the outputs of the first filter pair and comment on their relative energies (or standard deviations). Note that **U** and **V** are half the width of **X**, but that **U** is otherwise similar to **X**.

Now filter the columns of **U** and **V** using **rowdec / rowdec2** with the transpose operator:

UU = rowdec(U',h1)';
UV = rowdec2(U',h2)';
VU = rowdec(V',h1)';
VV = rowdec2(V',h2)';

Display [**UU VU; UV VV**], and comment on what sort of edges or features are selected by each filter. You may need to multiply the high-pass images by a factor $k > 1$ to display them clearly. Why is this?

We must now check that it is possible to recover the image from these sub-images, using reconstruction filters, **g1** and **g2**, and the functions, **rowint.m** and **rowint2.m** (which is modified in a similar way to **rowdec2** to allow correct alignment of the high-pass samples). To reconstruct **Ur** and **Vr** from **UU**, **UV**, **VU** and **VV** use:

g1=[1 2 1]/2;
g2=[-1 -2 6 -2 -1]/4;
Ur = rowint(UU',g1)' + rowint2(UV',g2)';
Vr = rowint(VU',g1)' + rowint2(VV',g2)';

Note the gain of 2 in the reconstruction filters, **g1** and **g2** (to compensate for losing half the samples in the decimation / interpolation processes). These filters are also not quite the same as those that might be inferred from the equations for x_n and x_{n+1} on the previous page. This is because **g1** defines how *only* the u samples contribute both to the even and odd samples of x, while **g2** defines how the v samples contribute.

Check that **Ur** and **Vr** are the same as **U** and **V**, and then reconstruct **Xr** from these:

Xr = rowint(Ur,g1) + rowint2(Vr,g2);

Check that **Xr** is the same as **X**.

The above operations are a bit tedious to repeat if we want to apply the DWT recursively to obtain several levels of filtering, so we have written a pair of functions, **dwt.m** and **idwt.m**, to perform the 2-D analysis and reconstruction operations. Examine these to see that they perform the same operations as above, except that the transformed sub-images are stored as parts of a single matrix, the same size as **X**, rather than as separate matrices. You can check their operation using:

Y = dwt(X); figure(1); draw(Y)
Xr = idwt(Y); figure(2); draw(Xr)

Y should be the same as the composite [**UU VU; UV VV**] image that you displayed earlier, and **Xr** should be the same as **X**.

Now implement a multilevel DWT by first applying **dwt** to **X** using:

m=256; Y=dwt(X); draw(Y)

and then iteratively apply **dwt** to the top left sub-image of **Y** by repeating:

m=m/2; t=1:m; Y(t,t)=dwt(Y(t,t)); draw(Y)

We now have the image split using a binary wavelet tree (stricly a quaternary tree in 2-D). Write similar iterative code to that given above, which can reconstruct the image from the final set of **Y** sub-images after a 4-level wavelet transform. Check that your reconstructed image is the same as **X**.

9.3 Quantisation and coding efficiency

First rewrite the sequences of operations required to perform n levels of DWT and inverse DWT as two separate M-files, **nlevdwt.m** and **nlevidwt.m**. **nlevdwt.m** should transform **X** into **Y**, and **nlevidwt.m** should inverse transform a quantised set of sub-images **Yq** into the reconstructed image **Z**. Check your functions by ensuring that **Z** is the same as **X** if **Yq = Y**.

Now design an M-file, **quantdwt.m**, which will quantise the sub-images of **Y** to give **Yq** and calculate their entropy. The sub-images at each level **i** of the DWT should be quantised according to a $3 \times (n+1)$ matrix **dwtstep(k,i)** of step-sizes, where $\mathbf{k} = \{1, 2, 3\}$ corresponds to each of the three high-pass images at level **i**, and the final low-pass image is quantised with **dwtstep(1,n+1)**. This matrix will be populated either with the same number in all elements (for equal-step-size quantisation) or a range of different numbers (for equal-MSE quantisation). The entropies for each sub-image should be stored in a similar $3 \times (n+1)$ matrix **dwtentk,i)**.

Using these M-files, for a given number of levels n (typically between 3 and 5), you should generate **Y**, quantise it to give **Yq** and reconstruct **Z** from **Yq**.

All of our experiments thus far have been performed on only one image. At this stage it is worth starting to experiment with the additional **Bridge** image, as well as Lighthouse. Bridge contains a lot more fine detail and may not lead to the same conclusions regarding

performance.

Investigate the performance of both an equal-step-size and an equal-MSE scheme (follow a similar procedure as you used for the Laplacian Pyramid to find the appropriate step-size ratios). Hence determine how many levels of DWT are reasonably optimal for the Lighthouse and Bridge images. Also evaluate the subjective quality of your reconstructed images, and comment on how this depends on n and on the way that step-sizes are assigned to the different levels. Once again, for each image choose quantisation steps such that you match the rms error to that for direct quantisation with a step-size of 17.

9.4 Second Interim Report

This report should include the new results from the DCT, LBT and DWT energy compaction methods in a format that will allow them to be compared with each other and contrasted to the Laplacian pyramid work in your first report. Again try to answer questions raised in the text, and also include discussion of any topics that have led to unexpected results or have proved particularly interesting.

10 Selection of preferred energy compaction options

The remainder of this project will concentrate on developing the rest of an image compression system, based on a few of the filtering / transformation schemes studied so far.

Since the subsequent processes are non-linear, we cannot expect to be able to choose precisely the right front-end at this stage, so we adopt the pragmatic approach of picking about three good candidates and trust that one of these will lead to a near-optimum solution in the end. Remember that up to this point we have only been using entropy to give us an *estimate* of the number of bits required, the accuracy of which is affected by subsequent stages.

At this stage it is worth trying your schemes with all three test images, (**Lighthouse**, **Bridge**, and **Flamingo**). You will find **Bridge** more difficult to compress than the other two. You may also want to introduce other images of your own.

Write M-files to implement each of your chosen schemes, so that you do not have to remember long sequences of commands each time you run them. You can easily edit the M-files to introduce different options later. Using plenty of comments in these files will help when you want to change them.

11 Centre-clipped linear quantisers

The quantisers that you have used so far have all been uniform quantisers (i.e. all steps have been the same size). However the probability distributions of the intensities of the bandpass sub-images from the energy compaction front-ends are usually highly peaked at zero. The amount of data compression depends heavily on the proportion of data samples which are quantised to zero; if this approaches unity then high compression is achieved.

Hence it is often found desirable to make the quantiser non-linear so that more samples tend to be quantised to zero. A simple way to achieve this is to widen the step-size of the 'zero' step. In a uniform quantiser, the 'zero' step is normally centred on zero, with rises to the next level at $\pm$ half of the step-size on each side of zero. **quantise** allows a third argument **rise1** to be specified, which is the point at which the first rise occurs on each side of the zero step. A value of **rise1** = **step/2** is the default, but **rise1** = $\{0.5, 1, 1.5\} \times$**step** are worth investigating. To show what effect these have, try:

x=[-100:100];
y=quantise(x,20,rise1); plot(x,y), grid

A wider zero step means that more samples will be coded as zero and so the entropy of the data will be reduced. The use of a wide zero step is beneficial if it results in a better entropy vs. error tradeoff than a uniform quantiser.

For each of your preferred front-end schemes, investigate the effects of varying the first rise of the quantiser. To do this, you could plot how the quantising error varies as a function of the number of bits for a few different ratios of **rise1** to step-size, and hence find the ratio which gives the best compression for a given rms error.

Most current image compression standards use quantisers with a double-width centre step (**rise1 = step**). Do not spend too much time on this as the compression gains are likely to be quite small.

Discuss whether your results indicate that **rise1 = step** is a reasonable compromise if all quantisers are to be similar.

A final strategy which you can consider is to completely suppress some sub-images or DCT coefficients. This is equivalent to increasing **rise1** to a very large value for these components. In the sub-images / coefficients which represent only the highest horizontal and vertical frequency components combined, the effects of suppression can be almost unnoticable and yet a useful saving in number of bits can be achieved.

Investigate any additional gains which can be achieved with suppression of some sub-images / coefficients.

12 Combined run-length / Huffman coding methods

Up to this point, we have been using entropy as a measure of the number of bits for the compressed image. Now we attempt to produce a vector of compressed image data which accurately represents the compression that can be achieved in practise.

Huffman codes are relatively efficient at coding data with non-uniform probability distributions, provided that the probability of any single event does not exceed 50%. However, when an image is transformed by any of the energy compaction methods considered so far, a high proportion of the quantised coefficients are zero, so this event usually does have a probability much greater than 50%. In fact it is only when this *is* a high probability event that high compression can be achieved! Therefore new ways of using Huffman codes have been developed to deal with this situation as efficiently as possible.

12.1 Baseline JPEG coding techniques

The standard Huffman coding solution, used by the baseline JPEG specification and most other image compression standards, is to code each non-zero coefficient combined with the number of zero coefficients which preceed it as a single event.

For example the sequence of coefficients:

$$3, 0, 0, -2, 0, 0, 0, 1, 0, -3, -1, 0, 0, 0, 0, 0, 1, \ldots$$

would be coded as the following 6 events:

0 zeros, 3
2 zeros, -2
3 zeros, 1
1 zero, -3
0 zeros, -1
4 zeros, 1

Each event has a certain probability (usually well below 50%) and can be coded efficiently with a standard Huffman code. As formulated above, the number of combinations of amplitude and run-length can be very large, leading to a highly complex code. JPEG limits this complexity to only 162 combinations by restricting the maximum run-length to 15 zeros and by coding only the base-2 logarithm of the amplitude in the Huffman code, rounded up to integers from 1 to 10. The sign bit and the remaining amplitude bits are then appended to the Huffman code word. 16 run lengths (0 to 15) and 10 log amplitudes (1 to 10) give 160 of the code words. The other two codewords are the end-of-block word (EOB), signifying no more non-zero coefficients in the current block, and the run-of-16 word (ZRL), which may be used repetitively ahead of another codeword for runs of 16 or more zeros.

JPEG is based on 8×8 DCT transformations of the image, and the data from each 8×8 block of DCT coefficients is coded as a block of Huffman codewords. First the dc

coefficient (top left corner) is coded. There is little penalty in using a fixed-length binary code for this, although JPEG uses differential and Huffman coding for slightly improved performance. Then the remaining 63 ac coefficients are arranged into a linear vector, by scanning the 8×8 block in a zig-zag manner corresponding to progressively increasing frequencies (see the JPEG standard, section 3, fig 5). This places the larger low-frequency coefficients close together near the start of the vector (with short run lengths) and the smaller high-frequency coefficients spread out towards the end of the vector (with long run lengths). The end-of-block word efficiently terminates the coding of each block after the last non-zero coefficient.

For further details of the JPEG techniques, referred to above, see the JPEG standard, sections 3.3 and 3.6 and appendices A.3, C, F.1.1, F.1.2, F.2.1, F.2.2, K.1, K.2, and K.3. Note that for this project we ignore the higher layers of the JPEG specification, and do not align code segments with byte boundaries or use two-byte marker codes to identify different data segments. JPEG also permits arithmetic codes to be used instead of Huffman codes, but these are more complicated so we recommend that you should use the latter.

12.2 Matlab implementation of Huffman coding

Performing bit manipulations, such as required for Huffman coding, is not very easy with Matlab. We have provided a number of M-files and M-functions to do this and suggest you use these where possible to save time. The files are fully commented and you should feel free to modify them (*and if so,* rename them) if you wish.

jpegenc.m Function to perform simplified JPEG encoding of an image **X** into a matrix of variable length codewords **vlc**.

jpegdec.m Function to perform simplified JPEG decoding of a codeword matrix **vlc** into an image **Z**.

quant1.m Function to quantise a matrix into integers representing the quantiser step numbers, which is the form necessary to allow Huffman coding.

quant2.m Function to reconstruct a matrix from integers. Together with **quant1.m** this is equivalent to **quantise.m**.

runampl.m Function to convert a vector of coefficients **a** into a matrix of run-length, log-amplitude and signed-remainder values **rsa**.

huffenc.m Function to convert a run/amplitude matrix **rsa** into a matrix of variable-length codewords **vlc**.

huffdflt.m Function to generate the specification lists, **bits** and **huffval**, for the default JPEG Huffman code tables for AC luminance or AC chrominance coefficients (JPEG specification, appendix K.3.3.2).

huffdes.m Function to design the specification lists, **bits** and **huffval**, for optimised JPEG Huffman code tables using a histogram of codeword usage **huffhist**.

huffgen.m Function to generate the Huffman code tables, **huffcode** and **ehuf**, from **bits** and **huffval**.

In order to allow relatively fast decoding in Matlab, we have cheated a little in the format of the coded data. Each variable-length codeword is stored as an integer element of the required word length in the first column of a 2-column matrix **vlc** and the length of the codeword in bits is stored next to it in the second column. We do not bother to pack this data into a serial bit stream since it is awkward and time consuming to unpack in Matlab. The length of the bit-stream if it were packed can easily be obtained from **sum(vlc(:,2))**.

To perform the simplified JPEG encoding, based on the 8×8 DCT, load the image in **X** and type: **vlc = jpegenc(X-128, qstep)**

This produces variable-length coded data in **vlc**, using quantisation step sizes of qstep. To decode **vlc**, type: **Z = jpegdec(vlc, qstep)**

These M-files are given to you as examples of how to achieve a complete compression system. They have other options and outputs, and in general you will need to modify them and rename them to perform your own algorithms.

In **jpegenc**, there are two ways to specify the Huffman tables: either the default JPEG AC luminance or chrominance tables may be used; or custom tables may be designed, based on statistics in the histogram vector **huffhist**. To generate a valid histogram for **huffdes**, coding must be performed at least once using **huffdflt** instead, so **jpegenc** is written such that the default tables are used first and then, if required, the code is redesigned using custom tables. Note that if it is planned to use **huffdes** to generate an optimised Huffman code for each new image to be coded, then the specification tables **bits** and **huffval** must be sent with the compressed image, which costs (16 + 162) bytes = 1424 bits. You should consider whether or not this is a sensible strategy.

12.3 Going beyond JPEG and the DCT

If you have chosen the DCT as one of your energy compaction methods then it is fairly straightforward to follow the JPEG guidelines for coding the coefficients. However if you have chosen one of the other methods then a modified scanning strategy is required.

It has already been mentioned that the LBT (which is at the heart of the JPEG-XR standard) is often coded several sub-blocks at a time. We can make a smaller LBT (4×4 is the default) look like a 16×16 DCT by using the **regroup(Yq, 4)** function within each 16×16 block of **Yq**. The functions **jpegenc** and **jpegdec** have already been written to do this if the **M** argument (which specifies the *coding* block size) is larger than the **N** argument (which specifies the DCT block size).

The DWT (which is the basis of the JPEG2000 standard) can also be re-arranged to make

it look similar to a DCT. For instance, a 3-level DWT could be re-arranged into an 8×8 block B using coefficients from the same square spatial area:

4 values from level 3: $B3 = [UU_3\ VU_3;\ UV_3,\ VV_3]$
3 surrounding 2×2 blocks from level 2: $B2 = [B3\ VU_2;\ UV_2\ VV_2]$
3 surrounding 4×4 blocks from level 1: $B = [B2\ VU_1;\ UV_1\ VV_1]$

It is not possible to achieve this sort of grouping using the simple **regroup** function, so we have provided a more complicated function **dwtgroup(X,n)** which converts an n-level DWT sub-image set into blocks of size $N \times N$ (where $N = 2^n$) with the above type of grouping. Try this function on some small regular matrices (e.g. **cumsum(ones(16,16))**) to see how it works. Note that **dwtgroup(X,-n)** reverses this grouping.

With these modified scanning strategies, the JPEG run-length / log amplitude coding can then be used for each vector in the same way as for the DCT coefficients. However, these scanning strategies are not optimal, and do not represent those outlined in the JPEG2000 and JPEG-XR standards.

You should write versions of **jpegenc** and **jpegdec** for your chosen compression strategies and check the following:

1. The rms error (standard deviation) between the decoded and original images should be the same as for the equivalent quantisation strategies that were tested in the previous section on centre-clipped linear quantisers. No extra errors should be introduced by the scanning or Huffman encode / decode operations.

2. The number of bits required to code an image should be comparable with the value predicted from the entropy of the quantised coefficients (i.e. within about 20%). Note that it is possible to code with fewer bits than predicted by the entropy because the run-length coding can take advantage of clustering of non-zero coefficients, which is not taken account of in the first-order entropy calculations.

13 Optimisation and test of the final design solution

At this point you should test your complete candidate coding strategies on the set of three test images which are provided (**Lighthouse**, **Bridge**, and **Flamingo**) and any other similar images if you wish. The aim of the tests is to get the best possible decoded image quality using no more than 40,960 bits (5 kB) per image. A combination of rms error and subjective quality will be used to assess the decoded image quality. If in doubt, you should select schemes with the **best subjective quality**.

Some suggestions for possible improvements to these schemes which may be worth investigating (other than the coding suggestions already mentioned) follow:

- Improve the scanning strategy, or use different sized blocks, before Huffman coding.
- Use of frequency-dependent quantisation - either the standard JPEG quantisation tables for the DCT or LBT or use of a quantisation of **step**$\times k^n$ where n relates to the DWT frequency band and k is just less than 1.
- Use of a hierarchical DCT or LBT with 2 or 3 levels (i.e. repeating a DCT or LBT on all the dc coefficients from the previous level), as in the JPEG-XR standard. Note that to make use of any additional compression, you will have to code in larger groups than the DCT size, or code the additional levels separately.
- Finding an error measure which is more directly connected to visual appearance than the rms error.
- Developing a super-scheme which uses one of a possible set of sub-schemes depending on some property of each image.
- Improving the coding of dc coefficients.

You are not expected to pursue all of these suggestions, nor limited to only looking at these ideas. Pick one or two that you think are interesting and may be fruitful. One of the features of good design is to develop an idea of what is likely to be of benefit before spending too much time pursuing it.

Once you have a solution you are happy with, you should ensure that you have an M-file to run it as simply as possible. It should print out the number of bits to code each image and the rms error between the coded image and the original, and display the original and the coded image side by side. You will probably need iteratively to adjust a single quantiser scaling parameter in order to achieve a bit rate less than than 40,960 bits for each image in turn. When you are satisfied that you have achieved an optimal design (within the constraints of this project), enter your design in the **Design Competition**, the rules of which are given overleaf.

Make sure that you have tested your code well before the competition is due to start,

14 Final Report

Your final report should be written to be read in conjunction with the two interim reports produced earlier. It should briefly describe the design processes which took place during the investigation of centre-clipped quantisers and Huffman coding; and then it should include a reasonably full description of the tests and reasoning which led to the final design selection, and of the final assessment tests. As before, you should emphasise any unexpected or subtle aspects of the project and not dwell on more obvious aspects.

The final report should include a brief discussion of how the development of the final design was shared within your group, and what aspects of this process each individual contributed to.

To complete the reports, include an overall summary and a final conclusions section, to bind the three reports into a coherent piece of work.

Joan Lasenby
Easter 2020

Image Processing Competition Rules

Aims:

To compress the 2 images in **bridge.mat** and **flamingo.mat**, and one new image, to less than 40,960 bits each (5 kB), including any header information, and to obtain reconstructed images with the **best subjective quality**. In the event that a decision between 2 or more groups cannot be reached based on subjective quality, the images with the minimum mean squared error (using **std(X(:)-Z(:))**) will be selected as the winners.

Procedure:

The competition will start at 10:00 on Monday 1st June in the DPO, at which point the additional image will be provided. Judging will begin around 12:00, by which point all decoded images should be displayed. Each group will score all the (anonymised) entries – it will be a challenge to do this remotely, but we hope to find a way!

Each group should prepare separate encoder and decoder M-files for their chosen compression system. The encoder program should take an image that has been loaded into matrix **X** and generate a compressed VLC matrix **vlc** (format as described in section 12.2 of the project handout) and any other integer parameters which are needed by the decoder. These parameters and **vlc** should be stored in a .MAT file called **?????cmp.mat** where **?????** are the first 5 characters of the image name.

The workspace should then be cleared, before **?????cmp.mat** is reloaded and the decoder program is then run to reconstruct the output image in matrix **Z** and display it. The image should be saved as **?????dec.mat**.

The decoder program should automatically check the number of bits needed by the vlc matrix (by summing column 2 — **vlctest.m** is a routine that can do this) and should add an estimate of the number of bits needed for any header information and print these two quantities out for any image it is decoding. It should also calculate the rms error between the original and reconstructed images and print that.

All 3 decoded images should be displayed in figure windows using **draw.m** for evaluation.

Measures to prevent cheating:

The following measures will be taken to discourage cheating:

1. The best candidate systems may be tested in order to check for cheating after the main competition has been held. One or more additional images may be used to check that the systems are not unduly image specific.

2. The vlc matrix and any additional parameters may be examined using **vlctest.m** before they are used by the decoder to check that the bit budget has been calculated correctly (each entry in the first column of **vlc** will be limited so that it cannot lie outside the range 0 to $2^n - 1$, where n is the entry in the 2nd column).

Joan Lasenby, Easter 2020