Multimedia Storage and Retrieval

Tutorial and further notes

Jonathan Edwards

Introduction

This document represents what I consider to be roughly *nine* weeks worth of tutorial material. The aim of it is to guide you through some of (what I consider!) the more interesting aspects of multimedia storage and retrieval. I hope that you will find at least some parts of the material interesting and memorable, and most of all enjoyable. The material at first may appear a little disjoint, this is because I've tried to cherry pick the best bits of this subject!

The tutorial reflects my approach to learning, which is to *do* first and ask questions *later*. I fundamentally believe you learn more by "tinkering", than you ever could do with a more structured approach (see the following article for a discussion on this topic `http://www.world-of-dawkins.com/Dawkins/Work/Articles/2002-07-06sanderson.shtml`). To help facilitate this, I have written example implementations of in the mathematical programming language **R** (see the **R** website `www.r-project.org`). The reason I have chosen this language is because there is a clear transposition between the ideas discussed and their implementation. Additionally, the language has minimal peripheral distractions (like **CLASSPATHS**!) hence things tend to work pretty much straight out of the box. It *is not* my aim that you will become a proficient **R** programmer, this skill will be of little use to you, I am just using it as it's the simplest way of getting the ideas across [1] It's freeware so by all means download a copy for home use. [2]A few ground rule for the tutorials: work through the tutorial and answer the questions *before* you move on, be sure to write these answers down so that I can check what you have and haven't done. Text in `Typewriter font` should be typed into **R**. All the material works (I have tested it!), so please make sure you have typed accurately. **Note** I have used (+)(+) to indicate a continuation in the **R** code (don't type this in !).

I have almost exclusively used **two** textbooks, and a handful of references to prepare this document. The one I am most keen on is David Mackay's outstanding text MacKay (2003) (30 quid) available for **FREE!** from `www.inference.phy.cam.ac.uk/mackay/itprnn/book.html` This book is a *must buy* (even though you can get it for free - it's cheaper to buy the book than to print it out!). For me the book is an absolute revelation, and should herald a renaissance in the field *Cybernetics* (data compression, AI, the interesting bits of stats (are there any you ask!)). The only problem with the book is that it is *hard*. For me it is a *life* goal to read the whole book! I will be getting you to perform a few exercises from this book so you might want a hard copy. The other book is "Fundamentals of Multimedia" Li and Drew (2003) (35 quid), this book is also necessary for the course, however is mount Snowdon compared to Mackay's K2. This text has more detail on the "Multimedia" aspects, but is less inspirational.

The Information Content of Data

There are two types of compression, *lossy* and *lossless*. It is easy to believe that lossy compression might work; you throw bits of your data away since you don't need them. However, lossless coding at first appears to be a *free lunch* situation, how can you make something smaller and it still *say* the same thing? The trick is to realise that there is redundancy already contained within the the data. This can be lost *without loosing any of the information* (see Figure 1)

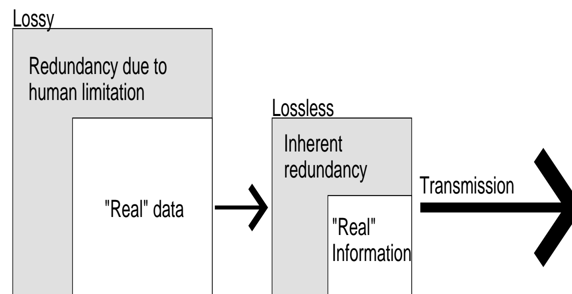


Figure 1: An schematic view of lossy and lossless compression (Transmission also means storage).

One point to note is that standard textbooks exaggerate this fact by presenting cases where the their discussed compression algorithms work with extreme efficiency rather than the more limiting cases (you will examine purported limiting cases in Section 1). For extreme compression things need to be lossy!

Q.1 Think about redundency from a multimedia perspective (images, sound, video), where is the redundancy?

[1] I also considered Java (not well liked by MM students),C (too low level for the work we are doing), Python (Nearly did the unit in this problems with graphing) and Perl (again hard to do graphs). **R** came out tops really because I want to present a lot of information visually.

[2] There is a small issue to do with add-on packages, this can be easily solved by installing *all* the extra packages (to do this go to `packages` menu then `install from cran` (make sure you are on the INTERNET) select all in the subsequent list and click **OK**.

Q.2 List 5 aspects of computing that are affected by compression technology. Isn't it fair to say that it lies at the heart of all modern applications.

Q.3 Answer this important question before you move on?

- **What would computing/life be like without compression?**

Measuring information content

One thing that we need to establish right from the outset is that there is a fundamental limit to the amount of compression that can occur. Before we do this however we first need to understand the key concepts of *information content* and *entropy* (why do we have to use physics words for everything?). An excellent illustrative example (stolen from Mackay (pages 70-72) is a simplified version of the `battleships` game. Everyone knows this game - just keep guessing until you hit the right square ("you sunk my battleship!"). In this simplified version of the game *only one* battleship exists, and it is only one square big. Assuming an 8x8 grid, the probabilities involved in this game are initially:

$$P(hit) = \frac{1}{64} \tag{1}$$

$$P(miss) = \frac{63}{64} \tag{2}$$

at $t = 0$ the guesser knows nothing, hence they have 0 bits of information about the battleships board (you'll understand why "bits" are the unit of information measure shortly). The game continues with guesses - each guess contributes some information to the guesser. At some point, $(0 < t < 64)$ a `hit` will be registered. How much information does the guesser know at this point?

Since every square on the board is a `miss` square, other than the `hit` square, once the guesser has sunk the battleship they know something about all 64 positions on the board. The guesser knows 64 Yes/No's which (remembering our Computer Systems Fundamentals) can be encoded in 6 bits of data. Interestingly, it doesn't matter when the guess occurs, once it has occurred everything else is know hence the total information content is 6 bits!!!

Q.1 In teams of two, have a game of battleship!

Q.2 What are your search strategies?

Q.3 Have another game of battleship using David Mackay's examples *Battleship* and *Minesweeper* `http://www.inference.phy.cam.ac.uk/mackay/itila/softwareI.html`

Q.4 (A hard question to do properly, but worth thinking about) Perform the Minesweeper game with a ship of size three. What are the probabilities? Why is this harder to calculate?

This is formalised by the *Shannon information content*

$$h(x = a_i) \equiv \log_2 \frac{1}{p_i} \tag{3}$$

where a_i is the particular event (ie. the occurrence of a the letter `B`) and p_i is the associated probability. A further necessary function is the *entropy*, which is the weighted Shannon information content

$$H(X) = \sum_i p_i \log_2 \frac{1}{p_i} \tag{4}$$

these are sometimes written as:

$$h(x = a_i) = -\log_2 p_i \tag{5}$$

$$H(X) = -\sum_i p_i \log_2 p_i \tag{6}$$

see notes on log function in appendix 1.

Here's the code for entropy in the **R** language:

```
h <- function(pa)
{
    log2(1/pa)
}
H <- function(pa)
{
    sum(pa*log2(1/pa))
}
```

Q.1 Input the above code and sketch $h(x)$ (think what the x axis will be, then plug some values in a calculator) *(hint: p is in the range?)*

Q.2 Using the R code below calculate entropy for pairs of probabilities that sum to one using an interval of 0.1. Where is the maximum? What does this mean?

Q.3 With this in mind tackle the *weighing problem* in (Mackay page 66). Note that this method is a general solution to many of life's difficult problems!!

```
binom <-matrix(c(seq(0.1,0.9,
(+) by=0.1),seq(0.9,0.1,by=-0.1)),9,2)
binomH <- apply(binom,1,H)
plot(binomH,type="h")
```

Now we have established the *information content* we will assert the following fairly understandable statement.

> *It is not possible to compress data losslessly below it's entropy.*

That seems fairly reasonable as entropy is synonymous with information content, the proof however is some quite detailed maths (which again is explained in Mackay (pages 78-84).

Things look bad for lossless compression, what do we do? Remember compression is not about changing the information content, but the codes that represent the information. If we can in someway decrease the size of the most probable characters (and accept the fact that the least probable will increase in size). **It is possible!** Lets examine, *the* classic lossless scheme Huffman coding (Mackay covers this in some detail in his Chapter 5).

Here's the general scheme.

1. Generate a frequency table.
   (a) Take the **two** least probable symbols and combine into a single code.
   (b) Repeat.

Since two codes are combine this is best visualised using a binary tree, lets look at another example now we know the algorithm, encoding the text "the_cat_sat_on_the_mat"

First we need to calculate the frequencies. These are:

symbol	*freq*	$p(x)$	$h(x)$
t	4	0.2	2.322
h	2	0.1	3.322
e	2	0.1	3.322
_	5	0.2	2.322
c	1	0.05	4.322
a	3	0.15	2.737
s	1	0.05	4.322
o	1	0.05	4.322
n	1	0.05	4.322
m	1	0.05	4.322

Using our bottom up algorithm the tree generated is thus

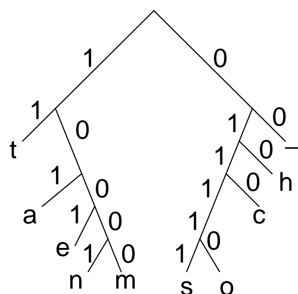


Figure 2: Huffman Tree for `the_cat_sat_on_the_mat`

Q.1 What is $H(X)$?

Q.2 A further metric that can be used is the expected bit length $L(C,X) = \sum_{i=1}^{I} p_i l_i$, where l_i is bit length of the code symbol i (I is the number of elements in X). What is the L(C,X)?

It is worth noting that for this ensemble Huffman does not produce and optimal codebook ($H(X) \leq L(C,X) < H(X)+1$, Mackay (page 101)) - Huffman coding is touted as the optimal encoding - it isn't - in fact for short sentence with low probability variance they produce a suboptimal code - for bigger more predicable documents this is outweighed and compression is achieved.

Arithmetic Coding and LZW

So far we have examined symbols rather than streams (although the coding scheme above is normally applied to blocks as this produces more efficient codes). A stream has the advantage that once you've seen a portion of it then you can have a better guess at the rest of it. This is known as a *predictive* model. The best current example of this *Arithmetic coding*. The basic premise is to generate a table which directly relates the probably to the cost (in bits of coding). This is easily achieved by establishing a range table that splits the interval $[0,1)$ [3] in proportion to the symbol probabilities. Let's look at the Arithmetic Coding algorithm:

1. Initialise an interval $[0,1)$.
2. For each symbol on the stream:
   a Subdivide the current interval proportionate to the *estimated* probability that a symbol will be the next symbol in the file.
   b Select the subinterval corresponding to the symbol that *actually* occurs.
3. output enough bits to fully distinguish this subinterval.

The crux parts of this method are in *italics* - *estimate* and *actual*. If the estimated probability model is *prefect* then optimal compression will be achieved!. Think about it this way - if you can always guess the next letter then your probability model is correct and hence you *always* generate the correct information (entropy) from your probability model (what a beautiful idea!). Lets look at the example `bye_bye_bo`:

Symbol	Range	Low	High
b	3	0	0.3
y	2	0.3	0.5
e	2	0.5	0.7
_	2	0.7	0.9
o	1	0.9	1.0
total	10		

[3] a set including 0 but not including 1

New Character	Low value	High Value
b	0.0	0.3
y	0.09	0.15
e	0.12	0.132
_	0.1284	0.1308
b	0.1284	0.12912
y	0.128616	0.12876
e	0.128688	0.1287168
_	0.1287082	0.1287139
b	0.1287082	0.1287099
o	0.1287097	0.1287099
output	0.1287097	

the output is a unique identifier for the interval in this case 1287097. Predictably you get nothing for free the two major problems with arithmetic coding are:

- Speed - up until the nineties this algorithm was too slow for common usage
- Estimating the probability - the "black hole" problem - unless you have a very sophisticated model this will be always be wrong! Nowadays there are specific models for text (ppmz) and images (DJVU) (check out this website `http://any2djvu.djvuzone.org/ulinit.php?submit.x=71&submit.y=26&submit=submit`)

Lempel Ziv Algorithm

Although arithmetic compression is currently the state of the art, *Lempel Ziv* is probably the most used. This simple algorithm again uses the stream as a way of eradicating redundancy. The theory behind is easily expressed

> 'you've seen this combination somewhere before!'

The stream is first parsed into a dictionary of unique substrings. This is then processed by encoding later substrings with **a bit plus an earlier substring**. This is best explained using an example here's `bye_bye_baby_baby_bye_bye`

source substring	$s(n)$	$s(n)_2$	(pointer,bit)
λ	0	0000	
b	1	0001	(,b)
y	2	0010	(,y)
e	3	0011	(,e)
_	4	0100	(,_)
by	5	0101	(1,y)
e_	6	0110	(3,_)
ba	7	0111	(1,a)
by_	8	1000	(5,_)
bab	9	1001	(7,b)
y_	10	1010	(2,_)
bye	11	1011	(5,e)
_b	12	1100	(4,b)
ye	13	1101	(2,e)

note: λ is the zeroth element. The stream also requires a unique terminal string (at the cost of a few bits).

Q. 1 Mackay offers a game to explain the functioning of an arithmetic coder (Mackay page 110). In a team of three play this game. Invent you own term, encode, and try to decode

- **Play game using your own intuition, do you get the right result?**
- **Play the game using the monogram and bi-gram distributions in appendix 1 (you should get the right answer!!)**

Q. 2 Encode the following (*by hand* use two of the three techniques above). Calculate $H(X)$ and $L(C,X)$

- `bbbbbbbbbjbbbbeddfd`
- `shouting larger larger larger`
- `they think it's all over, it is now`

Q. 3 Try Huffman using block lengths of 2 and 3.

Q. 4 Non-compression. Mackay (page 129-130) gives two examples which we are able to simulate. Using the code supplied in appendix to generate a pin-drop file and XAOS (`http://www.gnu.org/software/xaos/xaos.html`) to generate a fractal. By hook or by crook,manipulate the outcomes from the two processes (I used Irfanview) and generate a binary PBM file (raw one bit data) and a PNG (LZ compressed). Are Mackay's examples compressible?

Hash Coding

Hash functions are a way of generating a range limited address (a *hash*) from arbitrary sized integer, with the aim of using this address to reference the information. This seems a bit of an odd thing to do at first as it seems a little recursive, however when you think a little more about the idea of looking something up you rapidly realise that you don't want to be performing endless top to bottom list searches.

Q.1 Design an experiment to estimate the cost of adding and removing data from (you may wish to consult with Mackay's Chapter 12)

- **A raw list.**
- **A lookup table.**

There are various hash function, we will have a look at two different ones by comparing the collision probabilities using simulation. The most obvious hash function is *modulo* (normally $\mod n$). This is a straight forward range limiting function.

Here's an example of using the mod function to generate two weeks worth of days:

```
twoWeeks <- function(){
days <- c("monday","tuesday","wednesday",
(+) "thursday","friday","saturday","sunday");
for (i in 0:13)
  cat(i," ", days[(i%%7)+1],"
n");
}
```

Q. 1 How might I modify this code just loop through five days?

Q. 2 What would happen if I had done `7%%i`?

Now we have established that modulo is the basis for a hash function the next question is what is a suitable n (note this will fix the size of the eventual look-up table). Most books state that primes are best (**Q.4 why??**) and they should be not too close to a power of 2 (**Q.5 why??**).

Here's a function that returns a simple hash:

```
hash1 <- function(inVal,n)
{
  inVal%%n
}
```

We will test it on some random data, this is easily generated in **R** using the `sample` function, below is an example using $n = 43$. The `hist` function produces a histogram, that indicates spread of the index plus the number of collisions. An interesting experiment to perform is the number collisions compared to the fractional size of l/n where l is the probable number of items to store and n is the range of the hash function.

```
j <- sample(50000:60000,20,rep=T)
j.t <- table(hash1(j,43))
plot(j.t,type="h")
```

A more sophisticated hash function can be generated using a combination of flooring and multiplication Knuth (1968), this is called the *multiplication method*, here's the code:

```
hash2 <- function(inval,n)
{
  A <- (sqrt(5)-1)/2
  tempval <- inval*A
  floor(n * (tempval - floor(tempval)))
}
```

Repeat the above experiment with `hash2` - you should find that there is less collisions (I did! Figure 3)

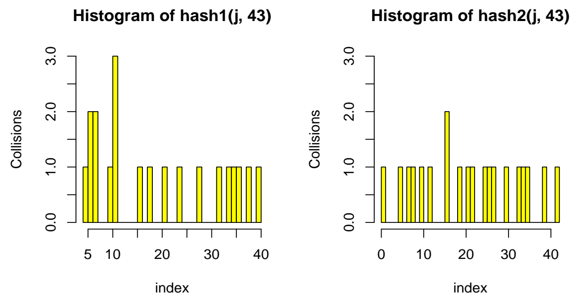


Figure 3: Collisions for the two hashing functions

Q. 1 So what? Explain what the above result means?

Q. 2 Repeat the above simulation varying the range of the sample, the size of the sample and the size of the index? What conclusions can be drawn?

MD5 for tamper detection

We talked about MD5 in the lecture - this is *the* state of the art hashing algorithm. An interesting use of hashing is to check that a file is what it says it is. This has implications for communications (to check that a uncorrupted file has been fully downloaded) as well as tamper detection and cryptography. As an example, lets take a bitmapped image (you can use any. A shield bitmap is available on from the module repository). Get the MD5 `.EXE` from the module repository and download it to the directory the image is in. Type the following:

```
md5 foo.bmp
```

where `foo.bmp` is the name of your bitmap. Note the response - **how many bits does this number represent?** Next read the image into paintbrush (or a suitable image editing package) and add **ONE BLACK PIXEL**. Save this as `foo2.bmp` and repeat the above `md5` process.

Q.1 Are the hashes the same ?

Q.2 Are they close ?

Q.3 What would the probability of a false collision be (assuming the has function is uniform and 128 bits)?

Here's some further fun! I wanted to generate two image files (of the same format) that look identical but contain different data. To do this I embedded some extra information in one of the files using a steganographic [4] tool. The tool I have used is called `Blindside` (`http://www.cs.bath.ac.uk/`

[4] steganography is the art of hiding information within another message. It is used to conceal information in a similar way to ciphering, however with a cipher you know there's a hidden message. For further information visit `http://en.wikipedia.org/wiki/Steganography`

`~jpc/blindside/index.htm` and was written by a **final** year undergraduate at Bath university (John Collomosse). If you don't believe me run the message retrieval process and find the hidden message (the `Blindside` zip is available in the Module repository).

First have a visual inspection of the the two images (**foo** and **sneaky**) do they look any different to you? Using the above MD5 process check whether the two files are identical. Do they look any different to MD5?.

The Discrete Cosine Transform

Fourier (and the Discrete Cosine) transform are amongst the most important tools for the signal processing (sound,images and video all being forms of signals). The DCT is thus:

$$F(j) = \sum_{k=0}^{n-1} cos\frac{\pi}{n} j(k - 1/2) f(k) \quad (7)$$

Make sense? The best way to think about it is to look at the code below - this is a programmatic interpretation of the above mathematical statement and should be easier to interpret. Basically we are multiplying the data by a series of cosine function - all with different periods. These are called the *basis*. The idea of basis functions is extremely important and has kept mathematicians, engineers, physicist and computer scientists occupied for the last 300 years (so don't worry too much if you don't get the concept immediately!)

```
dct <- function(inArray)
{
  n <- length(inArray)
  n1 <- n-1
  f <- rep(0,n)
  for( j in 0:n1){
    for (k in 0:n1){
      f[j+1] <- f[j+1] + cos(pi/n*j*(k+0.5))
(+)     * inArray[k+1]
    }
  }
  f
}
```

The best way to examine this further is to analyse some simple waves. First we need to generate a couple of waves (we will use the ones in the Fundamentals of Multimedia textbook to validate the code above). Here's the code:

```
plot(rep(100,32),type="l")
plot(dct(rep(100,32)),type="h")
plot(100*cos(seq(0,2*pi,l=16)),type="h")
plot(dct(100*cos(seq(0,2*pi,l=16))),type="h")
```

The results should just about agree with the textbook results to within a scaling (see Figure 4).

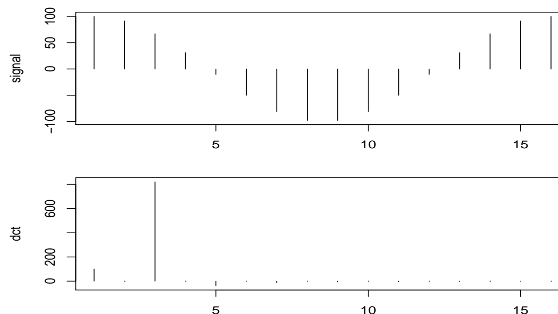


Figure 4: Signal and DCT from the book.

Next, we will switch analysis tools to using the the fast version of the Discrete Fourier Transform (FFT) as it more readily demonstrates the link between the time and frequency domain. the function which will be analysed is:

$$f(x) = cos(x) + cos(16x) + cos(50x) \quad (8)$$

```
sig <- seq(0,2*pi,l=256)
comp.sig <- cos(sig)+cos(16*sig)+cos(50*sig)
```

I have used this function as the individual parts can be easily illustrated. To generate this illustration plus the FFT the following lines of R are used:

```
sig <- seq(0,2*pi,l=256)
plot(cos(sig)+ cos(16*sig)
(+) +cos(50*sig) ,type="l")
lines(cos(sig)+cos(16*sig),col="blue")
lines(cos(sig),col="red")
plot(Re(fft(cos(sig)+cos(16*sig)
(+) +cos(50*sig))),type="h")
```

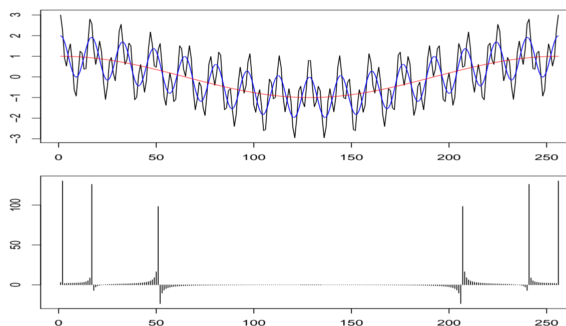


Figure 5: A complex wave and it's FFT

You can see that the three cosine components (indicated by different colour lines) have been identified by the FFT in positions 2,17 and 51 (don't worry about the mirrored impulses - this is not important in this instance). These correspond to the first, sixteen and fiftieth ac harmonic (the first value in the array is the dc offset which handles translation) The next thing to do is generate a complex wave form and quantise it used a very simple quantiser which simulates the emphasis on accuracy in low/mid frequency range over high frequency. A reasonable quantising function in the frequency domain would

be some kind of ascending function that emphasises the low frequency components. For JPEG (taking the diagonal of the 8x8 to remain in one dimension) the quantising function is:

$$F(\hat{u,v}) = round\left(\frac{F(u,v)}{Q(u,v)}\right) \quad (9)$$

where F is the image and Q is the quantisation level (see appendix 1 for quantisation levels)

In our example we will use a more radical quantising function - Get rid of everything but the first 16 ac components. Here's how we do this:

```
q2 <- c(rep(1,16),rep(0,224),rep(1,16))
Re(fft(cos(sig)+cos(16*sig)+cos(50*sig)
(+) + cos(100*sig)))*q2
sig.fft <- Re(fft(cos(sig)+cos(16*sig)
(+) +cos(50*sig)+ cos(100*sig)))*q2
sig2 <- fft(sig.fft,inverse=T)
plot(Re(sig2),type="l")
```

As you can see the only part of the signal left are the lower frequency components so the curve is a smoother representation.

Q.1 Using the above sampling rate `sig`, generate some more complex wave forms. View them using the plot function.

Q.2 Repeat the FFT plus quantisation steps as above (if you are clever you could use the JPEG values as shown in the appendix). Notice how the signal is still generally correct plus the little bit of high frequency at the end keeps some of the detail.

This is really the crux of what is elegant in JPEG - the next step are to bolt on a bit of lossless compression to further squeeze the data.

Wavelets

Wavelets == Fourier++

The aims of this tutorial are to establish *practically*:

- The nature of Wavelets.
- Briefly how they work.
- What they do to images?
- How they might be employed for compression?

We are going to do some preliminary Wavelet analysis using the **R** statistical package. Fortunately for you **R** has two Wavelet packages that provide all the tools necessary to perform Wavelet decomposition. Before you can do any analysis you need to load the necessary library (type `library(wavethresh)`this library contains all the function you will require for this particular tutorial.

A simple example

These are a couple of trivial examples to get the system up and running, they work on 1d signals.

Trivial Example 1: Wavelets on a random sequence

We just want any old number to start with so we'll generate a random series of digits to analyse. This is easily performed in **R** using the `Sample` function:

```
sample(1:20,64, replace=TRUE)
```

generates a set,

```
 [1] 20  9 13  4 20 17  7 20  7  3 12  6 16  5
 20  5 18 14  8 17 19  2 15  1 13
 [26] 17  9 12  8 14 18 20  3  7 20 19  3 11
 4 19 15 12  7 15 14 19 18  4 15 12
[51] 12  5 11 11  7 16 20  3 18  6  4 15 11  3
```

I bet your's are different!! To assign these to a variable (which I called `j`) use the $<-$ operator,

```
j <- sample(1:20,64, replace=TRUE)
j
```

To calculate the Wavelet co-efficients use the `wd()` function, different mother Wavelets are selected using the filter.number, in our case we will stick to the "Haar" Wavelet (filter number 1).

```
tw <- wd(t, filter.number=1)
tw
```

You will see a series of coefficients to retrieve the actual co-efficients use the rather clumsy `accessD` for downsized and `accessC` for co-efficients. Note that there is no way of choosing the number of levels directly.

```
accessD(tw,2)
accessD(tw,1)
accessD(tw,7)
accessD(tw,6)
accessD(tw,5)
```

A further tool for analysis is to plot the coefficients,`plot(tw)`

1. **Note what happens when you use the `plot` function.**

Trivial Example 2: Wavelets from a sequence

Just to show how one might work with sequenced data, try generating a Wavelet decomposition of the following sequence:

```
1:64
```

1. **Perform the same steps as above except at the start assign the above (`1:64`) instead of the random sample.**

Wavelets and Images

R can handle images (although it's hard to read your own in). An example image is the picture of John Lennon. To load this image you need to add the data to your environment.

```
data(lennon)
lennon
```

this should show the pixels of John Lennon!

Next you need to look at the image. **R** has a function to generate image from pixels, not surprisingly this is the `image` function . `image(lennon)` plots the image, the colours can be changed using the `col` switch `image(lennon, col=gray(255:1 /255))` gives a more reasonable colour mapping.

```
imwdL <- imwd(lennon)
c.gray <- gray(127:0/128)
plot(imwdL, col = c.gray)
plot(imwdL, col = c.gray, scaling = "none")
```

to change back to a Haar Wavelet use the `filter.number=1`.

```
imwdL <- imwd(lennon,filter.number=1)
plot(imwdL, col = c.gray, scaling =
(+) "none", co.type = "none")
```

1. **Perform the above analysis and examine the results.**
2. **Describe in your own words what the Wavelet decomposition has done.**

Further Questions

1. **The function `var` gives the variances of an array, what are the variances of each level of the pyramid?**
2. **What are the means?**
3. **Last week we looked at a very compressed JPEG-2000 image, given these experiments, why was it so bad?**
4. **Design an algorithm to extract the boundary from the John Lennon face.**
5. **The Burt and Adelson paper I have given you is a seminal work in the area of Multiresolution analysis, also it's a highly readable paper with fairly simple mathematics. In order to reinforce your work on Wavelets I would like you start producing a review document (no greater than 2 pages). In this document you will highlight:**
   - **a What problem the algorithm addresses?**
   - **b What approach to the problem does it take?**
   - **c How the algorithm works?**
   - **d What happens on a typical example?**
   - **e Does the algorithm fulfill it's aims?**

Java Multimedia Support

You NEED to be aware of the support JAVA offers for distributed multimedia (image,video,sound,networking and distributed computing).In groups of THREE pick one aspect and investigate the following:

- What functionality is available?
- What classes perform this functionality?
- How might these classes be used?

Report this to the rest of the class

IMPORTANT TUTORIAL Write an Exam!

Take the topic whose name has the nearest first letter to your name. Prepare an assessment question on this topic. The structure of this will be:

Question This should give the topic plus the marks awarded for the question. They must also consider the learning outcomes. It also should consider the issue of plagiarism (unless your going to have an exam!) and collusion.

Answer This should give the actual answer plus the marks to be awarded. There should be some indication of what to do for partial marks. Indications should be given for how long the question should take to answer.

Obviously, the question should be appropriate to final year level.

This information would be most useful if distributed in a P2P fashion

Logs

Logs to the base n of the number x is the power required of n to give x ($a = log_n(n^a)$). It is useful in many areas of maths, initially it was used to simplify large multiplications and additions when calculators were not available. We are mostly interested in log_2 ($log_2 \equiv lg$) - this can be thought of as a binary length function (check this out by calculating the log of some well known binary numbers 1,2,4,8,16,...). Logs have the following properties

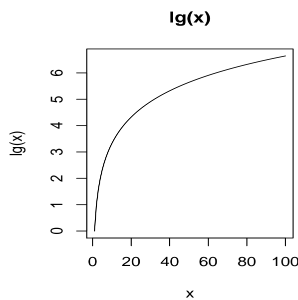
$$log_n 1 = 0 \tag{10}$$

$$log_n(n) = 1 \tag{11}$$

$$log_n(ab) = log_n a + log_n b \tag{12}$$

$$log_n(a/b) = log_n a - log_n b \tag{13}$$

$$log_n(1/a) = -log_n a \tag{14}$$



Generating incompressible images

The pin drop example can be simulated by the following **R** code

```
pins <- function(size=400,noPins=500,maxpinL=40){
#drops pins of length pinL
#on a surface (foo) either horizontally
#or vertically
#input: noPins
#input: maxpinL  pin length
#output
 pincushion <- matrix(1,size,size);
 pinpos <- round((size-(maxpinL+1))*runif(2*noPins));
 for (i in 1:noPins){
   pinL <- maxpinL*runif(1)
   if (0.5 < runif(1)){
     pincushion[pinpos[i],pinpos[i+1]:
(+)   (pinpos[i+1]+pinL)] <- 0;
   }
   else{
     pincushion[pinpos[i+1]:
(+)   (pinpos[i+1]+pinL),pinpos[i]] <- 0;
   }
 }
 pincushion;
}
```

to generate an image use the `image` command in **R** (we need the pixmap package to write out an image file)

```
library(pixmap)
z <- pixmapGrey(pins())
plot(z)
```

JPEG quantisation table in 1d

Here's a bit of code to generate a rough 1d view of the quantisation that takes place in JPEG.

```
jpeg.quant <- c(16,12,16,29,68,104,120,99)
plot(jpeg.quant,type="h")
```

the plot looks like this:

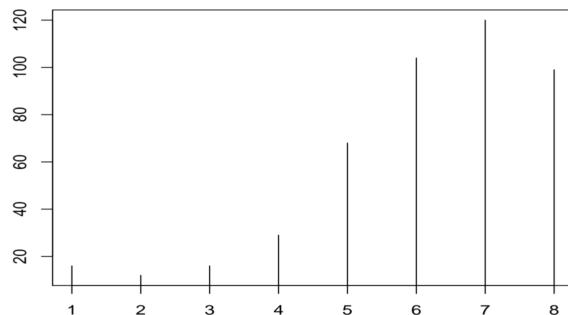


Figure 6: A rough 1d of the JPEG quantising vector. Notice how it's aim is to keep the low frequency signal and get rid of most of high frequency.

Mono and bi-gram alphabetic distributions

The distributions below were generated using the following piece of Perl code. They were generated on the module descriptor for this module,

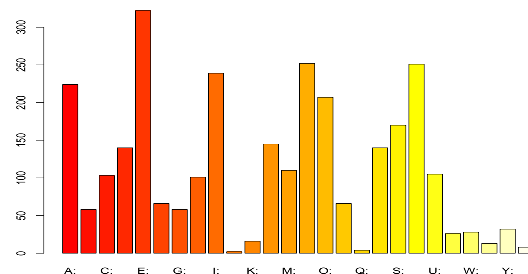


Figure 7: frequency of the letter in the alphabet in `unit.tex`.

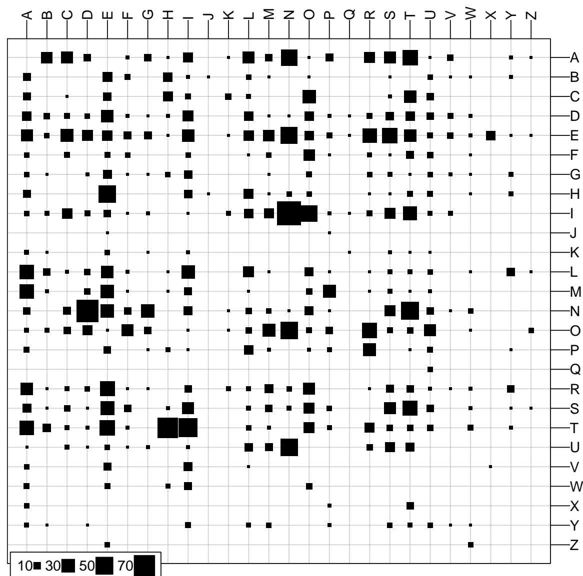


Figure 8: Conditional frequency of letters in the alphabet in `unit.tex`. Note this is read from the side and then down.

Jonathan Edwards
Dept. of Informatics,
University of Northumbria,
Newcastle upon tyne, UK
`{jonathan.edwards,paul.oman}@unn.ac.uk`

Bibliography

D. E. Knuth. *The Art of Computer Programming*. Four volumes. Addison-Wesley, 1968. Seven volumes planned (this is a cross-referenced set of BOOKs).

Z. Li and M. S. Drew. *Fundamentals of Multimedia*. Pearson, 2003.

D. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University press, 2003.