Introduction to Image, Video and Speech Processing

Ben Goldsworthy
33576556
Computer Science Innovation MSci

March 14, 2015

Contents

1 Part I - Image & Video Processing

1.1 Introduction

This first half of the report presents the results of reading and writing images. Whilst the tasks are intended for users of MATLAB, circumstances require me to use the functionally identical free software[1] alternative GNU Octave.[2]

1.2 Lab Session 1

The first lab session revolved around interpretation and manipulation of images.

Below is the initial image of a dog:

Loading the image into GNU Octave requires the `imread()` function, which reads the image into a 3-dimensional array (x, y, c), where x is the x-index, y the y-index and c the RGB colour layer. We can save the result of the function `rgb2gray()` on `Dog.jpg` using the function `imwrite()`, which takes the arguments of the image variable and the new filename. This produces the following:

Increasing and decreasing the brightness of an image requires adding or subtracting a constant value to all pixel values. For example, the result of an increase in brightness of 100 points:

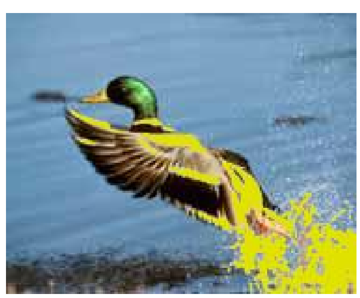
[1]https://www.gnu.org/philosophy/free-sw.html
[2]https://gnu.org/software/octave/

And a decrease in brightness of 100 points:

The function `flipLtRt()`[3] takes an image and produces a horizontally-mirrored image. It does so by creating a new image, where each value is initialised to 0, then populating the image with the pixel information of the original in reverse (i.e. for an image n columns wide, column 1 is filled with column n, column 2 with column $n-1$, etc.). The result is the following image:

The final portion of this first lab session is the script `yellowDuck.m`,[4] which places the image `duckMallardDrake.jpg` in the variable `im` and creates a new matrix of the same dimensions, initialised to 0s. It then places the values of each index of `im` in the new image, but if the pixel in question is more than 180 in all three colour channels, the value is replaced by $(255, 255, 0)$, which makes it yellow. Below is the original `duckMallardDrake.jpg`:

And here is the result of `yellowDuck`, saved to an image:



[3]Full analysis of code in Appendix A.1

[4]Full analysis of code in Appendix A.2

This illustrates a major difficulty with image processing, i.e. the lack of semantic difference between a white pixel of duck feather and white pixel of splashing water to the computer means both are treated equally.

1.3 Lab Session 2

The second lab session revolved around interpreting the image histogram.

Below is the initial image of a woman:[5]

The image was then converted into both grayscale (using `rgb2gray()`)...

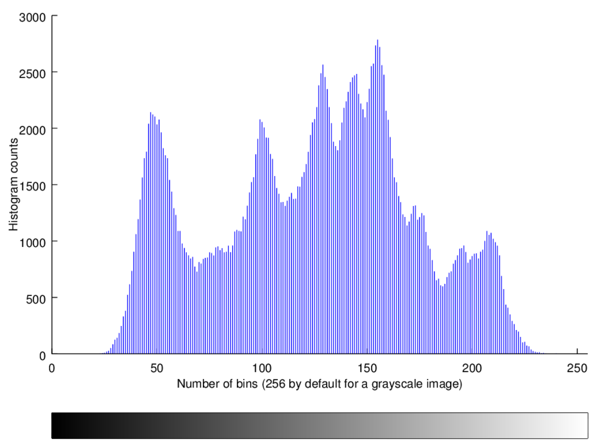
...and HSV (using `rgb2hsv()`) versions

Then, using the `im2bw()` function, the image was binarised using three threshold values (0.3, 0.6 and 0.9):

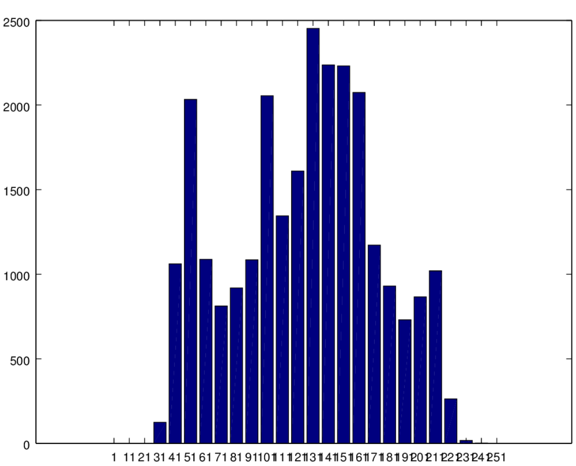
[5]The Lenna test picture, a ubiquitous part of computing history, is from a 1972 *Playboy* centrefold featuring Swedish Lenna Sjööblom; computing is weird.

As you can see, the value of 0.9 all but obliterated the image with black, much as 0.1 would have done with white. This is because the function works by dividing the 256 shades of grey into either black or white, based on the threshold value. With a value of 0.9, roughly 90 % of the image is deemed to be black.

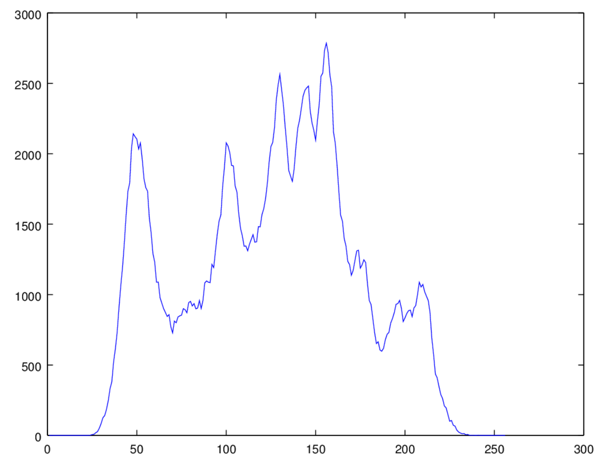
The program offers various ways of plotting an image histogram; one way of showing the greyscale histogram is the `imhist()` function:



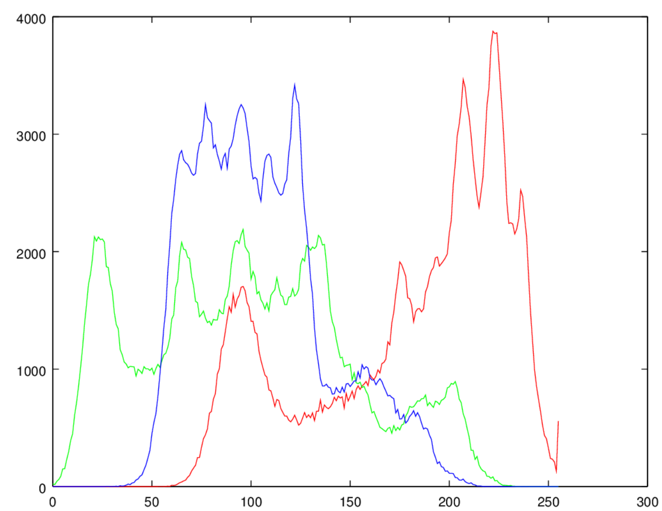
Another is the `bar()` function:



And another is the `plot()` function:



Following this, a histogram was plotted for the R, G and B channels of the original RGB image:

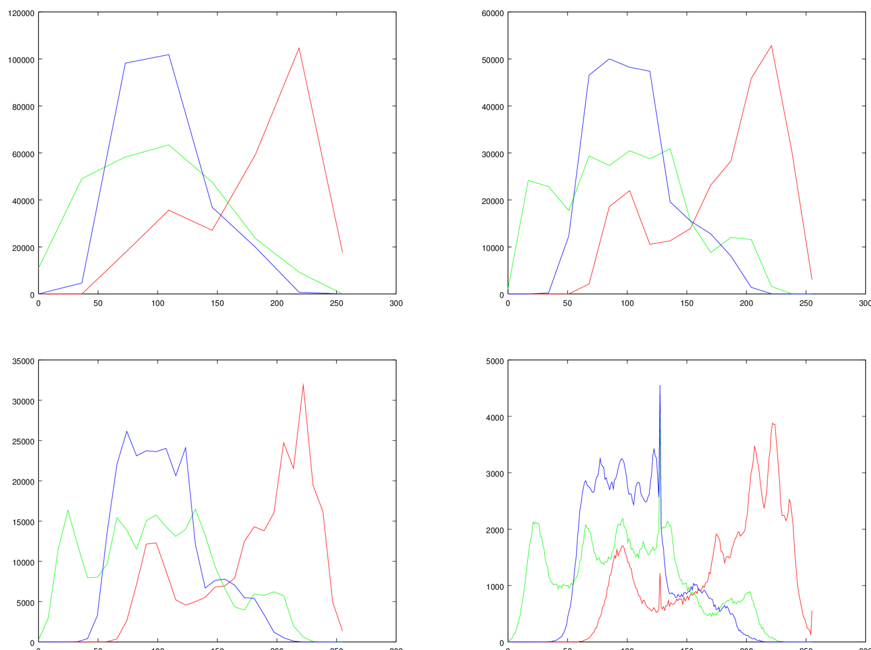


Choosing a threshold of 128 as a result of viewing the histogram, the following binarisation was produced:

These two examples show binarisations with thresholds of 100 and 150, respectively:

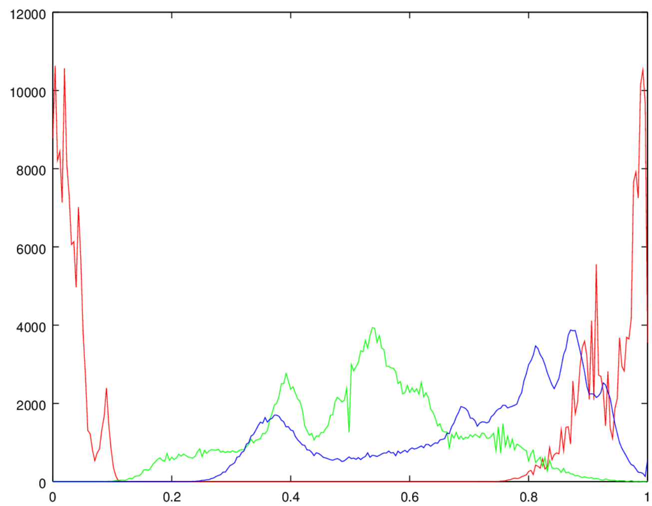
As before, varying the threshold affects the legibility of the image and, in more subjective terms, the mood.

Another way of manipulating the histogram comes from changing the bin size, or the precision of the results; below are results with bin sizes of $8 \times 8 \times 8$, $16 \times 16 \times 16$, $32 \times 32 \times 32$ and $255 \times 255 \times 255$, respectively:



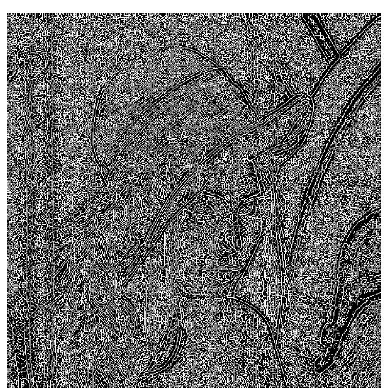
As one can see, the increased bin size translates to a more detailed line, but the smaller bin size can give a quicker notion of the broad distribution at a glance.

Below is a histogram of the HSV version of the image, produced the same way as the RGB one and coloured the same (i.e. the H channel is red, etc.):



Finally, edge processing algorithms were employed (via the `edge()` function); below are examples of Sobel and Prewitt edge detection, with the default threshold values used:

Sobel comes out on top for clarity, but only just; below is Sobel with a 0.1 treshold:



Not great; finally, here is Sobel with a 50 threshold:

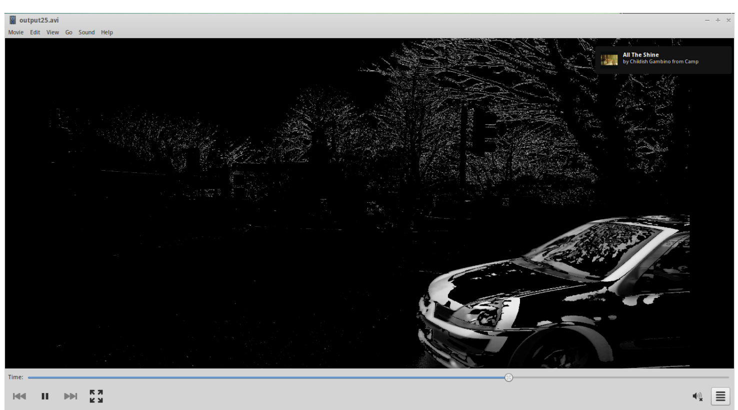
That'll do pig. That'll do.

1.4 Lab Session 3

The third lab session revolved around analysing video footage to detect elements.[6]

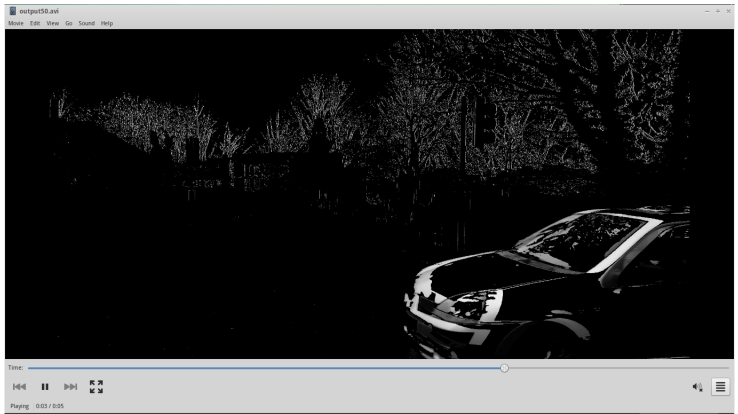
Below is a frame from the provided video `input.mp4`:[7]

Using code provided for the session,[8] a frame differencing algorithm was applied to the video with an initial threshold value of 25. Below is a frame from the resulting video:[9]

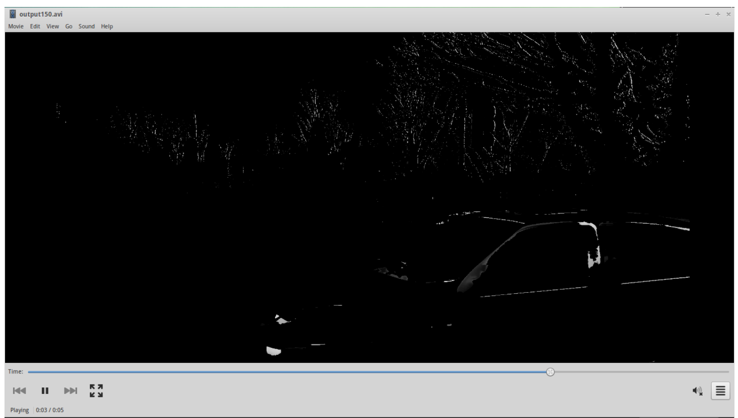


[6]For this session and this session only, MATLAB had to be used. This is due to the `VideoReader()` function not yet being implemented in GNU Octave (https://www.gnu.org/software/octave/missing.html)
[7]`input.mp4` available at http://ohwhatohjeez.co.uk/~files/input.mp4
[8]Full code in Appendix A.3
[9]Available at http://ohwhatohjeez.co.uk/~files/output25.avi

Two other threshold values were used; 50:[10]



and 150:[11]



As you can see, the car is perhaps best detected with a lower threshold; 25 gave the best results, whilst 150 barely separates the car from the background at all. However, a lower threshold than 25 would likely allow too much of the trees rustling in the background to distort the image. This portrays a weakness of as simple an algorithm as frame differencing.

[10] Available at http://ohwhatohjeez.co.uk/˜files/output50.avi

[11] Available at http://ohwhatohjeez.co.uk/˜files/output150.avi

2 Part II - Speech Processing

2.1 Introduction

This second half of the report presents the results of reading and manipulating speech signals. Again, GNU Octave was used.

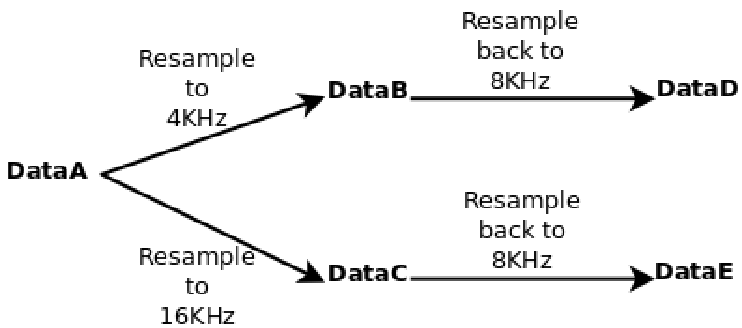
2.2 Lab Session 1

The first lab session revolved around basic audio manipulation.

Three `.wav` files were provided: `speech01.wav`,[12] `speech02.wav`[13] and `speech03.wav`.[14] Below are the durations and between-sample time intervals for each file:

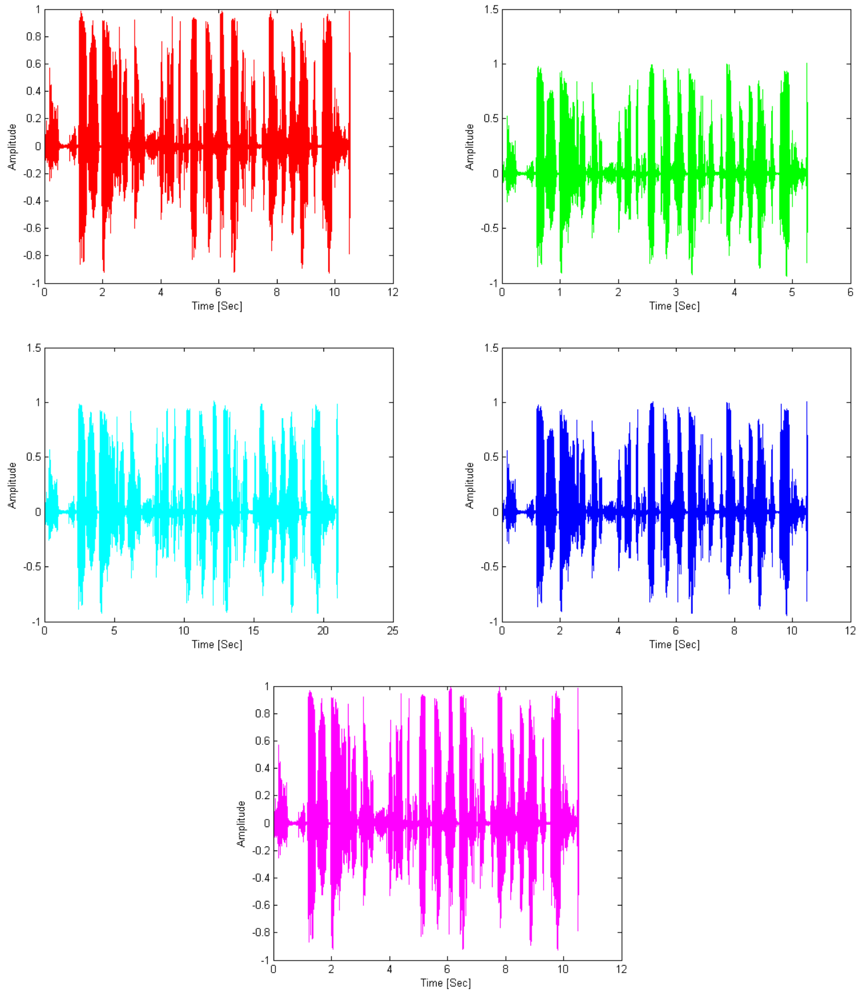
`speech01.wav`	10 s	0.02 s
`speech02.wav`	12 s	0.08 s
`speech03.wav`	15 s	0.06 s

For the rest of the session, I used the file `speech01.wav`, which contains 3.4 kHz bandwidth speech sampled at 8 kHz. Following the below process, I created five different versions of the file via various resampling:



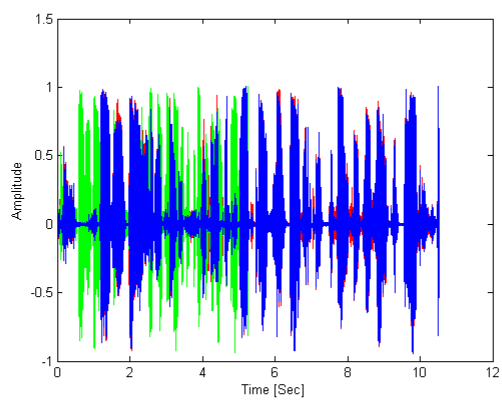
Below are the waveforms for DataA-E, coloured red, green,[15] cyan,[16] blue[17] and magenta,[18] respectively:

[12] Available at http://ohwhatohjeez.co.uk/˜files/DataA.wav
[13] Available at http://ohwhatohjeez.co.uk/˜files/speech02.wav
[14] Available at http://ohwhatohjeez.co.uk/˜files/speech03.wav
[15] Available at http://ohwhatohjeez.co.uk/˜files/DataB.wav
[16] Available at http://ohwhatohjeez.co.uk/˜files/DataC.wav
[17] Available at http://ohwhatohjeez.co.uk/˜files/DataD.wav
[18] Available at http://ohwhatohjeez.co.uk/˜files/DataE.wav

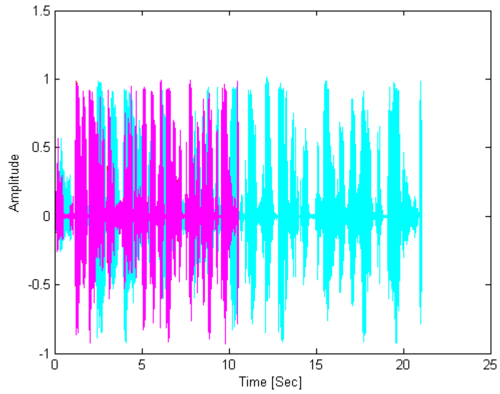


As you can see, DataB is half the duration of DataA, whilst DataC is twice it. This results in sped-up and slowed-down recordings.

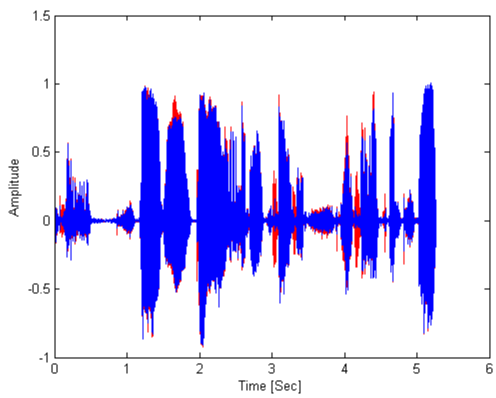
Below are the waveforms of the top path of the process (DataA–DataB–DataD), overlaid on each other:



And here are the overlaid waveforms of the bottom path (DataA–DataC–DataE):



Most visibly in the case of DataA and DataD in the top graph, even though they share a sampling rate and duration, there are differences introduced during the resampling process. This can be seen more clearly in the below graph, which plots half the length of both DataA and DataD:



However, the same general shape of wave is retained.

Listening to DataA, D and E, it's clear that whilst DataE is practically identical to DataA, DataD has suffered from the resampling process. This is because DataE came from DataC, which doubles the sampling rate of DataA, whilst DataD came from DataB, which halved it; as a result, information was lost. The Nyquist rate, which is at least twice the maximum frequency responses, is therefore demonstrated.

The signal to noise ratio (SNR) is discovered using the following formula:

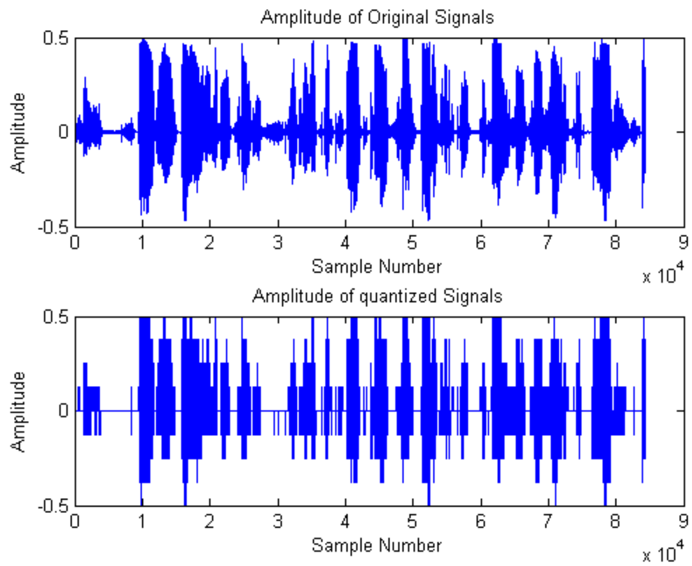
$$SNR_{dB} = 10\log_{10}\left[\left(\frac{A_{signal}}{A_{noise}}\right)^2\right]$$

This results in an SNR of 1:0.0875 for DataA and DataD, and !:0.0023 for DataA and DataE. This again relates to the Nyquist theorem, as DataD's construction from an audio track of half the sampling rate means far more noise is present than DataE, which its construction from one with twice the sampling rate (the Nysquist rate again).

Following this, quantisation was explored. The provided `QUAN_demo.m` function[19] handled the technical side of things. The initial quantisation value used was 3, which produced an audio track that crackled and was hard to hear, with the speaker's first couple words practically dropped in their entirety.[20] The effect was like listening on an old-timey wireless or similar device, albeit with perhaps a colder crackle to such a device's supposed warmer one.

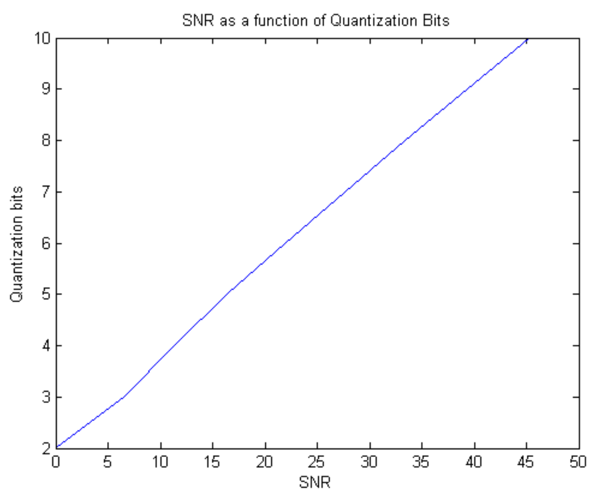
Below is a plot of both files' amplitudes:

[19] Full analysis of code in Appendix B.1
[20] Available at http://ohwhatohjeez.co.uk/~files/QUAN.wav



Obviously, the quantised file has had a large loss of information via the quantisation process.

Below is a plot of the SNR as a function of the number of quantisation bits R, using the aforementioned SNR formula:



The SNR increases along with the number of quantisation bits, i.e. more quantisation bits gives a clearer output. This becomes clear by listening to the outputs of quantisation bit values from 2-10,[21] with 2 almost illegible and 10 clear as day. Not having used a landline in years, I would hazard a guess that a quantisation value of 6[22] provides a level of quality most like the fixed public telephone network.

2.3 Lab Session 2

This lab session revolved around Linear Predictive Coding (LPC) based speech processing. Given two functions, `LPC_Analyze.m`[23] and `LPC_Synthesis.m`,[24] the `speech01.wav` file from the previous session was again used.

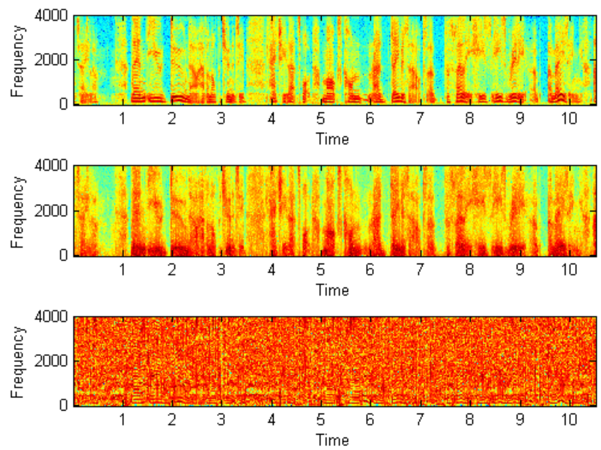
Initially, a noise-excited LPC synthesised signal was created without a residual error signal being generated.[25]

[21] Available at http://ohwhatohjeez.co.uk/~files/QUANx.wav, where x is the desired quantisation bits value
[22] Available at http://ohwhatohjeez.co.uk/~files/QUAN6.wav
[23] Full code in Appendix B.2
[24] Full code in Appendix B.3
[25] Available at http://ohwhatohjeez.co.uk/~files/neLPC.wav

This produces a version of the speech that is distorted, as though the speaker is breathing heavily into the microphone.

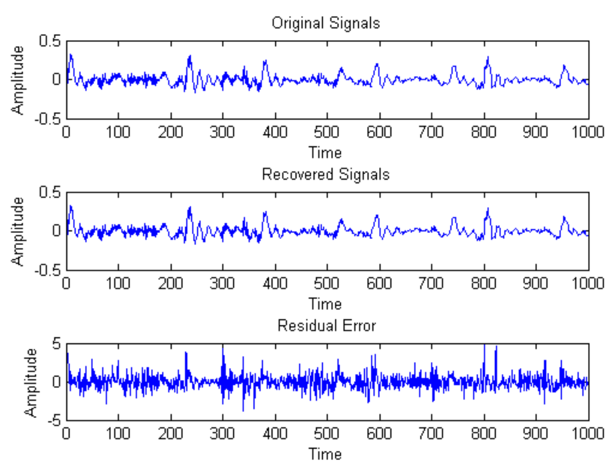
Afterwards, a residual-excited LPC synthesised signal was created, with a residual error signal.[26] This produces a signal that sounds the same as the original.

Below is a plot of the spectograms of the original speech, the noise-excited signal and the residual error signal:[27]



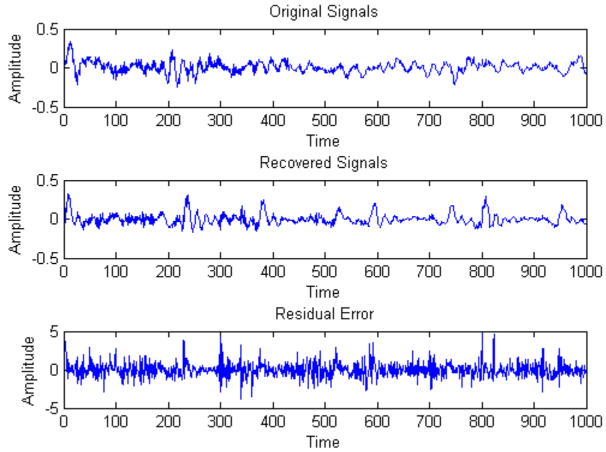
Finally, the original speech signal was synthesised at the output of the previous LPC synthesis filter, with an excitation input signal derived from the LPC analysis of a recording of a Bach piece.[28] The resultant file[29] recreates the speech using the sounds of the Bach piece. Impressively, it's still understandable.

Below are the waveforms of the original speech piece, the residual error signal dn the LPC synthesis recovered signals (the non-Bach one):



And below, with the noise-excited signal replacing the residual-excited one:

[26]Available at http://ohwhatohjeez.co.uk/~files/reLPC.wav
[27]Available at http://ohwhatohjeez.co.uk/~files/residual.wav; I suggest turning down your volume
[28]Available at http://ohwhatohjeez.co.uk/~files/bach16.wav
[29]Available at http://ohwhatohjeez.co.uk/~files/SynthwithMusic.wav



I then synthesised a noise-excited signal[30] and a residual-excited signal[31] from `speech02.wav`; the results were much the same.

[30] Available at http://ohwhatohjeez.co.uk/~files/rd.wav
[31] Available at http://ohwhatohjeez.co.uk/~files/rde.wav

A Appendix – Part I

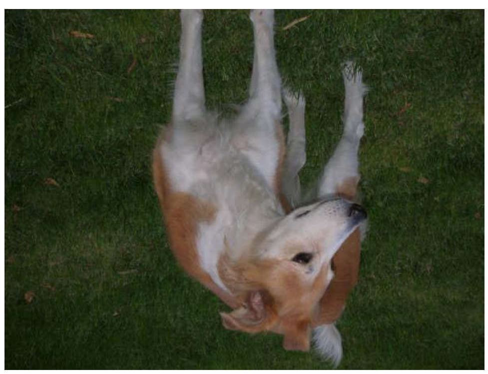
A.1 `flipLtRt`

```
1 function newIm = flipLtRt(im)
2 % newIm is impage im flipped from left to right
3
4 [nr,nc,np]= size(im);  % dimensions of im
5 newIm= zeros(nr,nc,np); % initialize newIm with zeros
6 newIm= uint8(newIm);   % Matlab uses unsigned 8-bit int for color values
7
8 for r= 1:nr
9    for c= 1:nc
10      for p= 1:np
11      newIm(r,c,p)= im(r,nc-c+1,p);
12     end
13    end
14 end
```

`flipLtRt` creates a new matrix with the same dimensions as `im`, initialised to 0s (lines 4–6). The row index is indicated by `r`, with `nr` storing the number of rows. `r= 1:nr` (line 8) therefore uses the `:` vector construction operator to run through every row from the first to the last. Nested within this `for` loop are two more loops, which run through the columns (`c` being the column index and `nc` the number of columns) and colour channel indices (`p` being the colour channel index and `np` the number of colour channels). So for a 3-dimensional matrix, the nested `for` loops start on the coordinate $(1, 1, 1)$ (MATLAB arrays start at 1), or the top-left pixel in the R colour channel. It then moves onto $(1, 1, 2)$, or the top-left pixel in the G colour channel. After $(1, 1, 3)$, it moves onto $(1, 2, 1)$, or the pixel to the right of the top-left pixel in the R colour channel. This repeats until it reaches the bottom-right pixel in the B colour channel.

For each pixel of `newIm`, the values inserted are those of the original `im`, but with the column (`c`) value taken from the opposite side of the image, as to flip an image horizontally, column c's value will be equal to that of column $nc - (c - 1)$ in the original image.

The function could be easily altered to flip vertically by swapping `im(r,nc-c+1,p)` for `im(nr-r+1,c,p)`. Below is the result of such a change:



A.2 `yellowDuck`

```
% Color the duck yellow!

im= imread('duckMallardDrake.jpg');
[nr,nc,np]= size(im);
newIm= zeros(nr,nc,np);
newIm= uint8(newIm);

for r= 1:nr
   for c= 1:nc
      if ( im(r,c,1)>180 && im(r,c,2)>180 && im(r,c,3)>180 )
         % white feather of the duck; now change it to yellow
```

```
                newIm(r,c,1)= 225;
                newIm(r,c,2)= 225;
                newIm(r,c,3)= 0;
            else % the rest of the picture; no change
             for p= 1:np
                newIm(r,c,p)= im(r,c,p);
                end
             end
        end
    end

    imshow(newIm)
```

This script, similar to the function in Appendix A.1, creates a new matrix of the same size as the image to modified, and also initialises it to 0s. Using a similar column-by-column, row-by-row loop as that function, `yellowDuck` then tests each pixel. If the pixel is higher than 180 in all three colour channels (which produces a light grey), the pixel in `newIm` is set to $(225, 225, 0)$, or yellow. Otherwise, the new pixel is given the same value as the old pixel.

A.3 Frame Differencing Algorithm

```
clear all

source = VideoReader(input. mp4 );

thresh = 25; % A parameter to vary

nFrames = source.NumberOfFrames; % Get the total number
for k=1:nFrames
   bg(k).cdata=read(source,k);
end

bgg=bg(1).cdata;

bg = source(1).cdata; % read in 1st frame as background frame
bg_bw = rgb2gray(bgg); % convert background to greyscale
% ----------------------- set frame size variables -----------------------
fr_size = size(bgg);
width = fr_size(2);
height = fr_size(1);
fg = zeros(height, width);
% --------------------- process frames -----------------------------------
for i = 2:nFrames
   fr = bg(i).cdata; % read in frame
   fr_bw = rgb2gray(fr); % convert frame to grayscale
   fr_diff = abs(double(fr_bw) - double(bg_bw));

   for j=1:width  % if fr_diff > thresh pixel in foreground
      for k=1:height
         if ((fr_diff(k,j) > thresh))
            fg(k,j) = fr_bw(k,j);
         else
            fg(k,j) = 0;
         end
      end
   end

   bg_bw = fr_bw;

   figure(1),subplot(3,1,1), imshow(fr)
   subplot(3,1,2), imshow(fr_bw)
   subplot(3,1,3), imshow(uint8(fg))
```

```
    M(i-1) = im2frame(uint8(fg),gray); % put frames into movie
end

movie2avi(M,'frame_difference_output', 'fps', 30); % save movie as avi
```

B Appendix – Part II

B.1 QUAN_demo.m

```
1  function [snr, Y, Yq, Sr] = QUAN_demo(R,string)
2    % The OUTPUTS are:
3    % snr is the signal-to-noise ratio of quantization
4    % Y is the original speech/audio data in the file
5    % Yq is the quantized data
6    % Sr is the sample rate
7    % The INPUTS are:
8    % R is the number of bits to set L = 2^R quantization levels
9    % string is the input WAV filename
10   % Read the speech file data
11   [Y,Sr]= wavread(string);
12   % Calculate the number of Quantization Levels
13   L = 2^R;
14   % Normalize amplitudes to range [+0.5, -0.5]
15   Y = Y/max(Y)*0.5;
16   % Perform the quantization
17   Yq = round(Y*L)/L;
18   % Print a message to screen
19   fprintf('Quantized with R = %d bits = %d levels\n', R, L);
20   % Compute the SNR
21   snr = sum(abs(Y).^2)/sum(abs(Y-Yq).^2);
22   return;
```

This function takes a filename and a number of quantisation bits, then manipulates the waveform based on the latter by adjusting the amplitude range based on the input signal (line 15, which takes the audio data in `Y` and divides it by the maximum frequency present, halved).

B.2 LPC_Analyze.m

```
function [a,g,e] = LPC_Analyse(x,p,h,w)
  % [a,g,e] = LPC_Analyse(x,p,h,w) Fit LPC to short-time segments
  %    x is a stretch of signal. Using w point (2*h) windows every
  %    h points (128), fit order p LPC models. Return the successive
  %    all-pole coefficients as rows of a, the per-frame gains in g
  %    and the residual excitation in e.
  % 2001-02-25 dpwe@ee.columbia.edu

  if nargin < 2
    p = 12;
  end
  if nargin < 3
    h = 128;
  end
  if nargin < 4
    w = 2*h;
  end

  if (size(x,2) == 1)
    x = x'; % Convert X from column to row
  end

  npts = length(x);

  nhops = floor(npts/h);

  % Pad x with zeros so that we can extract complete w-length windows
  % from it
```

```
    x = [zeros(1,(w-h)/2),x,zeros(1,(w-h/2))];

    a = zeros(nhops, p+1);
    g = zeros(nhops, 1);
    e = zeros(1, npts);

    % Pre-emphasis
    pre = [1 -0.9];
    x = filter(pre,1,x);

    for hop = 1:nhops
      % Extract segment of signal
      xx = x((hop - 1)*h + [1:w]);
      % Apply hanning window
      wxx = xx .* hanning(w)';
      % Form autocorrelation (calculates *way* too many points)
      rxx = xcorr(wxx,wxx);
      % extract just the points we need (middle p+1 points)
      rxx = rxx(w+[0:p]);
      % Setup the normal equations
      R = toeplitz(rxx(1:p));
      % Solve for a (horribly inefficient to use full inv())
      an = inv(R)*rxx(2:(p+1))';
      % Calculate residual by filtering windowed xx
      %rs = filter([1 -an'],1,wxx);
      aa = [1 -an'];
      rs = filter(aa, 1, xx((w-h)/2 + [1:h]));
      G = sqrt(mean(rs.^2));
      % Save filter, gain and residual
      a(hop,:) = aa;
      g(hop) = G;
      %e((hop - 1)*h + [1:w]) = e((hop - 1)*h + [1:w]) + rs'/G;
      e((hop - 1)*h + [1:h]) = rs'/G;
    end
```

B.3 LPC_Synthesis.m

```
function d = LPC_Synthesis(a,g,e,h)
   % d = lpcsynth(a,g,e,h) Resynthesize from LPC representation
   %    Each row of a is an LPC fit to a h-point (non-overlapping)
   %    frame of data. g gives the overall gains for each frame and
   %    e is an excitation signal (if e is empty, white noise is used;
   %    if e is a scalar, a pulse train is used with that period).
   %    Return d as the resulting LPC resynthesis.
   % 2001-02-25 dpwe@ee.columbia.edu

   if nargin < 3
     e = [];
   end
   if nargin < 4
     h = 128;
   end

   [nhops,p] = size(a);

   npts = nhops*h;

   if length(e) == 0
     e = randn(1,npts);
   elseif length(e) == 1
     pd = e;
```

```
  e = sqrt(pd) * (rem(1:npts,pd) == 0);
else
  npts = length(e);
end

d = 0*e;

for hop = 1:nhops
  hbase = (hop-1)*h;

   %  oldbit = d(hbase + [1:h]);
  aa = a(hop,:);
  G = g(hop);
  newbit = G*filter(1, aa, e(hbase + [1:h]));

   %  d(hbase + [1:w]) = [oldbit, zeros(1,(w-h))] + (hanning(w)'.*newbit);
   d(hbase + [1:h]) = newbit;
end

% De-emphasis (must match pre-emphasis in lpcfit)
pre = [1 -0.9];
d = filter(1,pre,d);
```