In the remainder of this course, we concentrate mostly on image and audio files. These files have the special property that the alphabet is a set of integers, representing a range of measurement levels of some physical quantity. Because the alphabet represents physical (intensity) quantities, it will make sense to perform operations such as +,-,*,/ on these quantitites. (Performing such operations on an alphabet such as ASCII made no sense.)

Digital Audio

A raw digital audio file represents a sequence of measurements of air pressure as a function of time and at a point in 3-D space, namely the position of a microphone.

Average atmospheric air pressure is called “1 atmosphere”. Audio files represent the time-varying *difference* between the air pressure at a point in space and the constant atmospheric pressure. That is, an audio file represents time-varying deviations of air pressure about some mean value.

High quality audio uses 16 bits – i.e. 2^{16} levels – to partition this range of pressures into equal units. This represents about $\pm 2^{15}$ levels above and below atmospheric pressure.

Raw high quality audio is typically sampled at rate of 44,000 pressure measurements per second, thus 88,000 bytes per second. This sampling rate is determined by the sensitivity of the human ear. The ear is not sensitive to variations that are faster than this. (You can think of the ear drum as a mechanical device. Naively you can think of this limit on sensitivity as a mechanical limit on how fast the ear drum can vibrate to “follow” the air pressure variation).

Just a few minutes of raw audio would require many megabytes. We would like to compress.

Digital images and Video

A typical digital camera takes images with 4 million pixels (4 MP), for example, it takes images of size 2048×2048 with 3 bytes per picture element (“pixel”) - one byte for red, one for green, one for blue. (We will not say anything more about color.)

A video is just a sequence of images. A typical video is 30 frames per second. You can calculate for yourself that a few minutes of raw video at such an image size would quickly fill up a few GB (a gigabyte is one billion bytes).

Differential coding

Audio files are compressible because the pressure often changes slowly from one time sample to the next. Similarly, image files are compressible because the intensity values often vary slowly from one pixel to the next, and one frame to the next (in case of video). It follows that a good way to compress audio or image files is to encode differences from one value to the next.

Consider a 1-D signal. This could be either the audio signal, or it could be a horizontal line of pixels in an image. Let $X_1, X_2, \ldots, X_n$ be the sequence of measurements. We are saying that X_j and X_{j+1} tend to have nearly the same value for all j.

One strategy for encoding such a file would be to encode,

$$X_1,\ X_2 - X_1,\ X_3 - X_2,\ \ldots,\ X_{j+1} - X_j,\ \ldots, X_n - X_{n-1}\ .$$

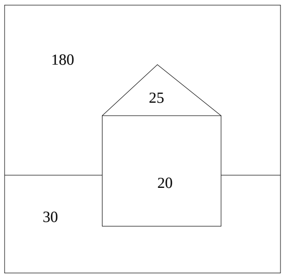
Since the differences $X_{j+1} - X_j$ tend to be relatively small, we can use fewer bits *on average* by using shorter codewords for values that are near 0 and longer codewords for values that are much different than 0.

Observe that the possible range of differences is greater than the range of raw measured values. A raw audio signal uses 16 bits for each value of X_j, and these represent the "pressure level" $\{-2^{15}, \ldots, -1, 0, 1, 2, \ldots, 2^{15}-1\}$. However, the differences $X_{j+1}-X_j$ can come from twice as large a range. The largest value of of $X_{j+1} - X_j$, is $2^{16} - 1$, and the smallest possible value of $X_{j+1} - X_j$, is $-(2^{16} - 1)$. Thus, the differences $X_{j+1} - X_j$ can take values in the range $[-(2^{16} - 1), 2^{16} - 1]$ which is an alphabet of size $2^{17} - 1$.

Of course, if one knows the value of X_j, then there are really only 2^{16} possible values of $X_{j+1}-X_j$, not 2^{17} possible values. The reason is simply that there are 2^{16} possible values of X_{j+1}. However, if we have to define a single code for the differences between two values, then the alphabet is of size $2^{17} - 1$.

Differential coding of 2-D images

The cartoon drawing below is an oversimplification of real images but it illustrates the basic concept. Many images have regions in which the intensities are roughly constant. These regions are separated by well-defined boundaries (sometimes called "edges"). The intensities differences across boundaries can be large, and so when we take the difference of two intensities on opposite sides of the boundary we can get a large number. Thus, for most image positions, differences of intensities give near zero values, but occasionally the difference is large.



If image intensities have 256 values (0 to 255), intensity differences can range from $\{-255, \ldots 255\}$. One way to encode intensity differences for 2D images is as follows. Let $X(i, j)$ be the pixel at row i and column j. We could define the difference image:

$$D(i,j) = \begin{cases} X(1,1), & i = 1 \text{ and } j = 1 \\ X(i,1) - X(i-1,1), & i > 1 \text{ and } j = 1 \\ X(i,j) - X(i,j-1), & j > 1 \end{cases}$$

We could compress the image quite well by encoding the difference image D, since differences tend to have values near 0.

The above method is quite convenient since raw images intensities are typically stored row-by-row in memory. (We can think of an image as a 2D array. The image is stored by row-by-row in the file. There is a natural block structure in the file, namely each row is one block.)

[ASIDE: The method is clearly reminiscent of fax compression. However, there are some crucial distinctions here. With faxes, each image bit had only value (white or black) whereas now we have 256 grey levels. It doesn't make sense here to use run-length coding, since run-length coding does not discriminate a low intensity difference (common) from a high intensity difference (uncommon).]

Transform coding

Many methods for coding image and audio files can be thought of as performing a linear transform on the raw data. For example, take the case of $n = 8$. The differential coding method above can be written as a transformation:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \\ Y_8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \end{bmatrix}$$

The encoder would encode the Y sequence, rather than the original X sequence.

The decoder needs to perform the inverse. Given the Y sequence, recover the original X sequence. The inverse transform happens to be:

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \\ Y_8 \end{bmatrix}$$

I am not proposing that the encoder and decoder construct such matrices. Rather, I am merely illustrating that there is a linear algebra interpretation that applies here. Later we will see why writing transforms as matrices is useful.

When n is very large, often we will partition it into blocks of a fixed size m , and encode each block. If we partition $X_1, X_2, \ldots, X_n$ into blocks of size m, we would encode blocks

$$(X_1, X_2, \ldots, X_m)(X_{m+1}, \ldots, X_{2m}), \ldots (X_{n-m+1} \ldots X_n)$$

To encode the blocks, we can treat each m-tuple as an vector of integers, perform a linear transform, and then encode the transformed values. The example of size $m = 8$ is the matrix above.

For $m = 2$, the transform is:

$$\begin{bmatrix} Y_{2j-1} \\ Y_{2j} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} X_{2j-1} \\ X_{2j} \end{bmatrix}$$

Let's do something a bit different. We transform:

$$\begin{bmatrix} Y_{2j-1} \\ Y_{2j} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} X_{2j-1} \\ X_{2j} \end{bmatrix}$$

This amounts to encoding a sum and a difference.

Notice that the transform is an orthogonal matrix. The rows are orthogonal vectors. This means that the inverse is just the transpose (times the factor 1/2). The inverse transform is:

$$\begin{bmatrix} X_{2j-1} \\ X_{2j} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} Y_{2j-1} \\ Y_{2j} \end{bmatrix}$$

This orthogonality property is nice. We will see more examples later.

Block coding of 2-D images

Suppose we partition our 2D image into square blocks of size 2×2. For each of the blocks (i, j), the pixels in the block are $X(2i-1, 2j-1)$, $X(2i-1, 2j)$, $X(2i, 2j-1)$, $X(2i, 2j)$. We can transform the intensities by:

$$\begin{bmatrix} Y(2i-1, 2j-1) \\ Y(2i-1, 2j) \\ Y(2i, 2j-1) \\ Y(2i, 2j) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} X(2i-1, 2j-1) \\ X(2i-1, 2j) \\ X(2i, 2j-1) \\ X(2i, 2j) \end{bmatrix}$$

The $Y(2i-1, 2j-1)$ values tend to be large since they are sums of intensities of four neighboring pixels. The $Y(2i-1, 2j)$, $Y(2i, 2j-1)$, and $Y(2i, 2j)$ values tend to be near zero, however, because they represent intensity differences of neighboring pixels.

For example,

$$\begin{bmatrix} 531 \\ 1 \\ 3 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 132 \\ 133 \\ 132 \\ 134 \end{bmatrix}$$

Also note that the transform matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 \end{bmatrix}$$

is an orthogonal matrix. Its inverse is just its transpose (divided by 4). Thus, given the Y values for each block, the decoder can easily compute what were the original X values by multiplying by the inverse matrix.

Finally, note that the range of each of the Y values is four times the range of each of the X values. The $Y(2i-1, 2j-1)$ value is always positive, whereas the other three Y values can be either positive or negative. For example, if the X values were eight bits each (2^8 possible values), then the $Y(2i-1, 2j-1)$ values would be anywhere from 0 to $4 * (2^8 - 1)$, whereas the other three Y values would be in the range $-2 * (2^8 - 1)$ to $2 * (2^8 - 1)$.