2 Signals in the Digital Domain

Multimedia - Department of Computing, Imperial College
Professor GZ Yang ! **http://www.doc.ic.ac.uk/~gzy**

Sampling Theorem

In multimedia, there are two different sources of information that we are mainly dealing with. The first of such sources is what we find around us in the natural world with our eyes and ears. These sources are normally first converted into an electronic representation in which the signal amplitude is an analog of, for example, the volume or brightness. For digital processing, such as compression, these analog signals must be subsequently converted into the digital domain through sampling and quantization. This conversion should be performed in such a way that visible or audible disturbances that would alter the information content carried by the analog video or audio signal are avoided.

The second source of signals occurs when data are generated or substantially modified in the digital domain. Examples include synthesized sound and video, computer graphics, or virtual reality presentations. For efficient data processing and communication, synthesized sources are commonly limited, prior to compression, to levels that do not exceed the reproduction vehicle. For example, synthesized sounds may have frequencies outside of human hearing range, while computer-generated graphics may have sharp transitions or highly saturated colours that would not survive television transmission and display.

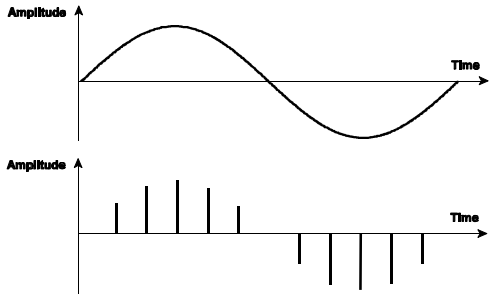


Figure 1

Analog signals in general are continuous in time and value. The amount of information contained in an analog signal is infinite from the information theory point of view. Digitised signals, however, exist only for certain points in time are represented by discrete amplitude values only. This reduction of information is the key that makes digital processing so useful, and is in fact one of the very first steps of signal compression. To convert an analog signal into a digital signal, the analog signal is usually sampled at equal time intervals, and the amplitude of each sample is quantized and assigned to a digital code word. The digital signal is thus a sequence with a constant bit rate derived from the equidistant sampling process of binary coded numbers of equal length (Figure 1). The sampling theorem dictates that to represent an analogue signal truthfully, one needs to sample the signal at a frequency that is at least twice of the maximum frequency of the original signal. That

is

$$f_{sampling} > 2 f_{\max}$$

Figure 2 demonstrate the sampling of a sinusoidal analogue waveform and its digital representation when sampled at different sampling frequencies. The recovery of the digital signal when sampled at 4f and 2f is unique. We hit a problem when we sample the original signal at 1.5f, there can be more than one analogue waveform that can pass through all the sampling points! This is called signal aliasing. In all digital

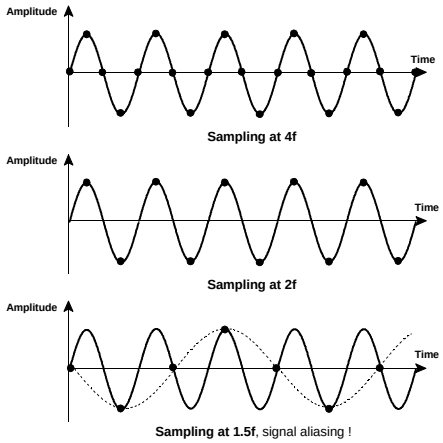


Figure 2

processing systems, it is important to determine the maximum frequency range that you are dealing with, and we normally employ an ***alogue*** anti-alias filter to throw away all signals that have frequencies beyond $f_{sampling}/2$. It is then followed by digital processing steps required, and finally convert them back as analogue signals (Figure 3**)**. The conversion back to analogue signal is done by a digital -to-analog converter (DAC), which delivers one value from the range of possible output levels at each clock cycle, giving a continuous signal in time. The only unwanted effect is that with amplitude changes between successive samples, there is no smooth transition between the two amplitude values but rather a step, which may lead to distortions. Therefore, the analog output signal from the DAC must be passed through a regeneration filter. This is a low-pass filter which, if designed properly, has no influence on the information carried by the signal up to the critical frequency.

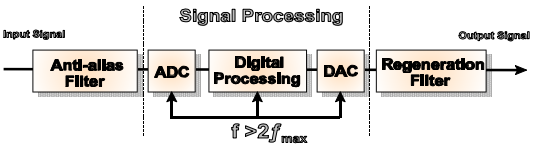


Figure 3

Simple Digital Processing for Data Compression

There is an intrinsic limitation to what we can see and what we can hear. An understanding of the characteristics of the destination is very useful for the design of a data-transmission system with compression. After all, the only information which must be carefully transmitted is what actually perceivable by the viewer/listener. Simple digital processing for data compression for colour images include colour re-

quantisation and the use of colour look-up tables. For example, most true colour images are stored in 24 bits/pixel, with 8 bites each for representing the red, green and blue components. By reducing this to 16 bits/pixel, such that we use 5 bits for each colour components, this will provide a 33% data reduction. Since in most colour images not all the colour space is occupied, one can also list out the most frequently used colours in a particular image and use a colour look-up table for storing the image data. For example, if we set the colour look-up table to have 256 entries, with each entry representing one colour mixture, we can achieve a 66% reduction in storage space. This is because we only need 8 bits for each pixel, plus 256x3 bytes for storing the look-up table. This is, in fact, how VGA cards were first designed when video memory at that time was still expensive.

DPCM and ADPCM

DPCM stands for Differential Pulse Code Modulation, which is a technique by exploiting spatio-temporal variations of the signal for achieving a higher compression ratio without jeopardizing the quality of the signal itself. Since most signals are smoothly varying in time or space, one can represent a signal by its initial amplitude plus inter-sampling differences. Figure 4 is an example audio signal where only 6 bits for representing signal variations between [-32, 31] are needed to encode the difference signals. This is due to the relatively low amplitude in inter-sampling differences. In this simplistic scheme, we do not expect the difference signal to exceed 6 bits, otherwise we will get slope overload (smearing of high contrast edges for images). This is because when the difference signal exceeds 6 bits, the residual error has to be sent by the subsequent data points. If the original signal is

2, 4, 5, 4, 16, 18, 17, 4, 3, 3, 3, 70, 72, 70, 70,70,70

The sequence we need to send becomes

2, 2, 1, -1, 12, 2, -1, -13, -1, 0, 0, 67, 2, -2, 0, 0, 0

Notice here that we have a strong signal variation between 3 and 70 which exceeds the encoding range, so in practice, the actual signal that can be sent by using only 6 bits becomes

2, 2, 1, -1, 12, 2, -1, -13, -1, 0, 0, 31, 31, 5, 0, 0,0

which results in a blurred edge. This problem can be partially resolved by using adaptive step size for representing the difference signal, such that large step sizes are used when sharp changes in signal amplitude are encountered. The overall

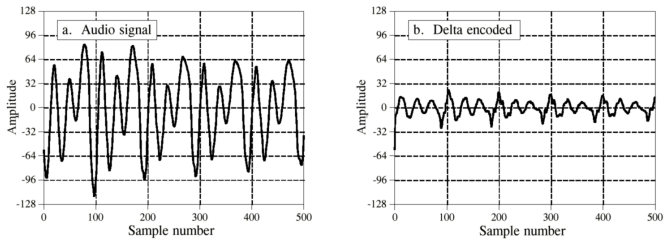


Figure 4

compression that can be achieved in practice by DPCM or ADPCM is about 2:1, and they are most commonly used for compressing audio signals.

With DPCM, it is possible to use prediction to further improve the compression ratio.

In this case, m samples within a causal neighbourhood of the current sample are used to make a linear prediction (estimate) of its amplitude. More specifically,

$$\hat{x} = \sum_{i=0}^{m-1} a_i x_i$$

where x_m are the m samples prior to the current sample, and $á_i$ are the corresponding coefficients. To reduce the system complexity, the prediction is usually rounded to the nearest integer, although it may be preserved in floating point representations. It is also necessary to clip the prediction to range $[0,2^n-1]$ for an n-bit signal. The differential (error) signal, e_m, is constructed as the difference between the prediction and the actual value; *i.e.*,

$$e_m = x_m - \hat{x}_m$$

The differential signal typically has a greatly reduced variance compared to the original data and can be significantly compressed. The question now is to find the best predictor coefficients $á_i$. A widely used criterion is the minimization of the mean-squared prediction error. Under this criterion, the best linear estimate of x_m is the value that minimizes the expected value of the squared prediction error, *i.e.*,

$$s_e^2 = E - \left\{ \left(x_m - \sum_{i=0}^{m-1} a_i x_i \right)^2 \right\}$$

This is realized by making the prediction error orthogonal to all available data, and the m optimal coefficients can thus be found by solving the following set of linear equations:

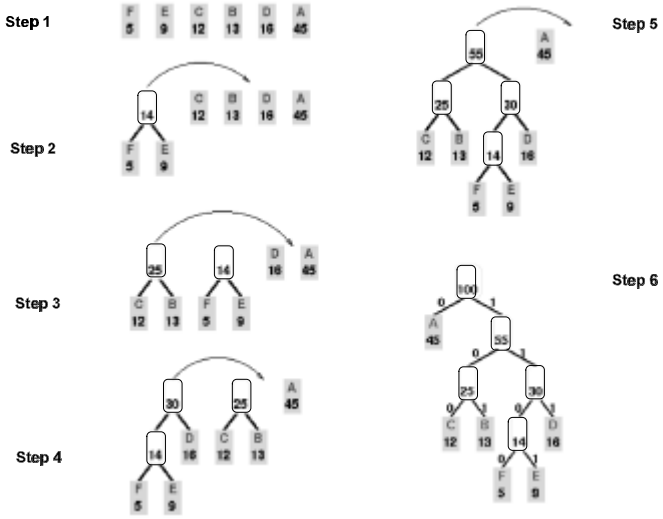
$$E\left\{ \left(x_m - \sum_{i=0}^{m-1} a_i x_i \right) x_k \right\} = 0$$

for k=0, ..., m-1.

Huffman Coding

Huffman encoding is a simple compression algorithm introduced by David Huffman in 1952. To understand Huffman encoding, it is best to use a simple example. Suppose we have a 32-character phrase: "traversing threaded binary trees" and wish to send the phrase using standard 8-bit ASCII codes, we would have to send 8*32= 256 bits. Could we reduce this number? Given that there are only fourteen different characters in the phrase, we could represent each with just four bits per character (since 2^4= 16, which is larger than 14). Then we would only have to send 128 bits. But could this be reduced further? Using the variable length codes in the Huffman algorithm we are about to describe, we can reduce the size by an additional ten percent, sending only 116 bits.

To form these codes, a binary tree structure is needed. This binary tree differs from standard binary trees by the fact that it is most easily constructed from the bottom to the top: from the leaves to the root, in other words. The procedure is as follows: first, list all the letters used, including the "space" character, along with the frequency with which they occur in the message. Consider each of these character/frequency pairs to be nodes. Pick the two nodes with the lowest frequency, and if there is a tie, pick randomly amongst those with equal frequencies. Make a new node out of these two, and make the two nodes its children. This new node is assigned the sum of the frequencies of its children. Continue the process of combining the two nodes of lowest frequency until only one node, the root, remains. The diagram below demonstrates all the steps involved in encoding a text block that consists of



The resulting code table: A-0, C-100, B -101, F-1100, E -1101, D -111

characters A-F, with respective occurrences vary from 45 to 5. Here is the pseudo code for Huffman encoding

```
Repeat the following steps until there is only one node left:
{
       1)  Find the two nodes with the lowest frequencies.
       2)  Create a parent node for these two nodes.
               Give this parent node a weight of the sum of the two nodes.
       3)  Remove the two nodes from the list, and add the parent node.
}
```

Run Length and LZW Coding

Run Length Encoding is a conceptually simple form of compression. RLE consists of the process of searching for repeated runs of a single symbol in an input stream, and replacing them by a single instance of the symbol and a run count. So for a string

17, 17, 17, 0, 0, 0, 0, 97, 25 ,25, 25, 25 ...

The corresponding run length encoding becomes

17, 3, 0, 3, 97, 1, 25, 4, ...

It is particularly useful for encoding black and white images, such as those from a fax page.

LZW compression is named after its developers Lempel, Ziv and Welch. The original Lempel Ziv approach to data compression was first published in in1977. Terry Welch's refinements to the algorithm were published in 1984. The algorithm is surprisingly simple. In a nutshell, LZW compression replaces strings of characters with single codes. It does not do any analysis of the incoming text. Instead, it just adds every new string of characters it sees to a table of strings. Compression occurs when a single code is output instead of a string of characters. The code that the LZW algorithm outputs can be of any arbitrary length, but it must have more bits in it than a single character. The first 256 codes (when using eight bit characters) are by default assigned to the standard character set. The remaining codes are assigned to strings as the algorithm proceeds. The sample program runs as shown with 12 bit

codes. This means codes 0-255 refer to individual bytes, while codes 256-4095 refer to substrings.

The LZW compression algorithm in its simplest form is shown in the following code block. A quick examination of the algorithm shows that LZW is always trying to output codes for strings that are already known. And each time a new code is output, a new string is added to the string table.

```
Routine LZW_COMPRESS
        STRING = get input character
        WHILE there are still input characters DO
            CHARACTER = get input character
            IF STRING+CHARACTER is in the string table then
                STRING = STRING+character
            ELSE
                output the code for STRING
                add STRING+CHARACTER to the string table
                STRING = CHARACTER
            END of IF
        END of WHILE
        output the code for STRING
```

The following table illustrates a sample string used to demonstrate the algorithm listed above. The input string is a short list of English words separated by the '/' character. Stepping through the start of the algorithm for this string, you can see that the first pass through the loop, a check is performed to see if the string "/W" is in the table. Since it isn't, the code for '/' is output, and the string "/W" is added to the table. Since we have 256 characters already defined for codes 0-255, the first string definition can be assigned to code 256. After the third letter, 'E', has been read in, the

Input String = /WED/WE/WEE/WEB/WET			
Character Input	**C**ode Output	**N**ew code value	**New String**
/W	/	256	/W
E	W	257	WE
D	E	258	ED
/	D	259	D/
WE	256	260	/WE
/	E	261	E/
WEE	260	262	/WEE
/W	261	263	E/W
EB	257	264	WEB
/	B	265	B/
WET	260	266	/WET
EOF	T		

second string code, "WE" is added to the table, and the code for letter 'W' is output. This continues until in the second word, the characters '/' and 'W' are read in, matching string number 256. In this case, the code 256 is output, and a three character string is added to the string table.

The process continues until the string is exhausted and all of the codes have been output. The sample output for the string is shown in the table listed on the right along with the resulting string table. As can be seen, the string table fills up rapidly, since

a new string is added to the table each time a code is output. In this highly redundant input, 5 code substitutions were output, along with 7 characters. If we were using 9 bit codes for output, the 19 character input string would be reduced to a 13.5 byte output string. Of course, this example was carefully chosen to demonstrate code substitution. In real world examples, compression usually doesn't begin until a sizable table has been built, usually after at least one hundred or so bytes have been read in.

For LZW decompression, one needs to be able to take the stream of codes output from the compression algorithm, and use them to exactly recreate the input stream. One reason for the efficiency of the LZW algorithm is that it does not need to pass the string table to the decompression code. The table can be built exactly as it was during

Input Codes: / W E D 256 E 260 261 257 B 260 T

Input	OLD_CODE	STRING Output	CHARACTER	New table Entry
/	/	/		
W	/	W	W	256 = /W
E	W	E	E	257 = WE
D	E	D	D	258 = ED
256	D	/W	/	259 = D/
E	256	E	E	260 = /WE
260	E	/WE	/	261 = E/
261	260	E/	E	262 = /WEE
257	261	WE	W	263 = E/W
B	257	B	B	264 = WEB
260	B	/WE	/	265 = B/
T	260	T	T	266 = /WET

compression, using the input stream as data. This is possible because the compression algorithm always outputs the STRING and CHARACTER components of a code before it uses it in the output stream. This means that the compressed data is not burdened with carrying a large string translation table.

```
Routine LZW_DECOMPRESS

           Read OLD_CODE
           output OLD_CODE
           WHILE there are still input characters DO
               Read NEW_CODE
               STRING = get translation of NEW_CODE
               output STRING
               CHARACTER = first character in STRING
               add OLD_CODE + CHARACTER to the translation table
               OLD_CODE = NEW_CODE
           END of WHILE
```

Just like the compression algorithm, it adds a new string to the string table each time it reads in a new code. All it needs to do in addition to that is translate each incoming code into a string and send it to the output. The following table shows the output of the algorithm given the input created by the above compression process. The

important thing to note is that the string table ends up looking exactly like the table built up during compression. The output string is identical to the input string from the compression algorithm. Note that the first 256 codes are already defined to translate to single character strings, just like in the compression code. Unfortunately, the nice simple decompression algorithm shown here is just a little *too* simple. There is a single exception case in the LZWcompression algorithm that causes some trouble to the decompression side. If there is a string consisting of a (STRING,CHARACTER) pair already defined in the table, and the input stream then sees a sequence of STRING, CHARACTER, STRING, CHARACTER, STRING, the compression algorithm will output a code before the decompressor gets a chance to define it.

A simple example will illustrate the point. Imagine the the string JOEYN is defined in the table as code 300. Later on, the sequence JOEYNJOEYNJOEY occurs in the table. The compression output will look like that shown below. When the decompression algorithm sees this input stream, it first decodes the code 300, and outputs the JOEYN string. After doing the output, it will add the definition for code 399 to the table, whatever that may be. It then reads the next input code, 400, and finds that it is not in the table. This is a problem, what do we do? Fortunately, this is the only case where the decompression algorithm will encounter an undefined code. Since it is in fact the only case, we can add an exception handler to the algorithm. The modified algorithm just looks for the special case of an undefined code, and handles it. In the example in Figure 5, the decompression routine sees a code of 400, which is undefined. Since it is undefined, it translates the value of OLD_CODE, which is code 300. It then adds the CHARACTER value, which is 'J', to the string. This results in the correct translation of code 400 to string "JOEYNJ".

```
Routine LZW_DECOMPRESS

Read OLD_CODE
output OLD_CODE
WHILE there are still input characters DO
    Read NEW_CODE
    IF NEW_CODE is not in the translation table THEN
        STRING = get translation of OLD_CODE
        STRING = STRING+CHARACTER
    ELSE
        STRING = get translation of NEW_CODE
    END of IF
    output STRING
    CHARACTER = first character in STRING
    add OLD_CODE + CHARACTER to the translation table
    OLD_CODE = NEW_CODE
END of WHILE
```

Input String: ...JOEYNJOEYNJOE

Character Input	New Code/String	Code Output
JOEYN	300 = JOEYN	288 (JOEY)
A	301 = NA	N
.	.	.
.	.	.
.	.	.
JOEYNJ	400 = JOEYNJ	300 (JOEYN)
JOEYNJO	401 = JOEYNJO	400 (??)

Sources

1. The scientist and Engineer's guide to digital signal processing - SW Smith, California Technical Publishing 1997.

2. LZW data compression by Mark Nelson, Dr Dobb's Journal, Oct, 1989.