Digital Encodings for Information Representation

Prof. C.M. Wyss Prof. Y.M. Wu

In this part of the course, we will learn about the encodings which make information representation possible. The problem is as follows. We use mainly use English (or another natural language), mathematics, pictures, and sound to communicate and represent the world around us. How can these languages be usefully represented in a computer?

English uses an alphabet of 26 characters, which can be upper or lower case (for a total of 52 characters). In addition to the letters of our alphabet, there are also ten digits (0 through 9) we use to represent numbers – this brings the total up to at least 62. In written English, there are also various punctuation marks, bringing the total of symbols in the working alphabet still higher. Furthermore, what can we use as the "alphabet" for picture and sound data? The answer is not immediately obvious, but it probably involves many new symbols.

On the other hand, a computer is based on electronic circuits. The basic state of the physical components of a computer is inherrently *binary* in that it can hold one of only two values (usually these are modeled by the digits "0" and "1"). Thus the inherrent alphabet of the computer has only two symbols. This is nowhere near as many as we use! It may surprise you to know, however, that using this simple two-valued alphabet, we can nonetheless represent *all* the same information that (written) English can represent. We can also approximate numbers (including real numbers), pictures, and sound data usefully.

This is the subject of the current topic: digital encodings for information representation. Some issues to think about as you progress through this topic:

1 Text Encodings

One of the basic functions of a computer is to store text data (written language). This is accomplished with a *text encoding* which translates the English alphabet into binary. Of course, computers are also great calculators. Thus, there are other encodings for numbers, the most common of which we will see later.

It happens that the numeric digits 0 through 9 can appear in both text and as elements of numeric calculations. In programming languages, this is essentially the distinction between the string "1" and the number 1. Don't let this confuse you simply because we use the same symbol for the letter "1" and the number 1. This distinction arises because of the two different ways we *use* this symbol in communication. This dichotomy is reflected in the fact that we have separate encodings for "1" (the letter) and 1 (the number).

Definition 1.1 (The Encoding Process)
Suppose we have a source alphabet Σ_s consisting of n characters, each denoted a_i (for $1 \leq i \leq n$). Every word, w, in the source language is the concatenation of some of these characters: $w = a_{i_1} \cdots a_{i_k}$. Given an encoding of the source alphabet, ε, we produce the encoding of the word w by concatenating the encodings of the symbols in w: $\varepsilon(w) = \varepsilon(a_{i_1}) \cdots \varepsilon(a_{i_k})$.

Since we only have two "letters" to work with inside a computer, we'll have to use *multiple copies* of these letters to represent the English symbols.

Given a source alphabet, there are multiple ways to encode it. In the following sections, we will introduce a few of them, starting from *fixed width encoding*.

1.1 Fixed-Width Encodings

The simplest (and most effective) encoding for the English alphabet is a *fixed-width encoding*. This means that each alphabet symbol is encoded using a string of 0s and 1s that is the same length as all the others. Let's see how this works.

Example 1.1
We can represent the English letters 0, 1, 2, 3 as binary strings of length two:

$$0 \mapsto 00$$
$$1 \mapsto 01$$
$$2 \mapsto 10$$
$$3 \mapsto 11$$

Hopefully, this encoding looks familiar from your knowledge of how counting in binary works.

Example 1.2
We can play the same trick with the other English symbols; for example, to represent A, B, C, D, E, F, G, H we could use binary strings of length three:

$$A \mapsto 000$$
$$B \mapsto 001$$
$$C \mapsto 010$$
$$D \mapsto 011$$
$$E \mapsto 100$$
$$F \mapsto 101$$
$$G \mapsto 110$$
$$H \mapsto 111$$

To really convince yourself this works, you need to understand what this encoding is doing. We see "ADG" written on a page in English, and according to our encoding (above), this will translate to 9 bits in a computer: 000011110. Thus, the encoding for strings (commonly called *words*) in the input language involves encoding each of the symbols in the string separately.

In modern computers, similar encoding schemes are used to represent the symbols of natural languages. It is important to note that the reason these encoding schemes work is because they are *standardized*. Someone came up with the original encoding, and then everyone else agrees to use it. This is why an "A" on one computer is an "A" on every computer — this is why the encoding succeeds in representing real text.

We will now look at some of the standard encodings in use today.

1.1.1 ASCII

The original text encoding is known as ASCII (pronounced "Ask-ee"). When someone asks for "plain text", that usually means they want ASCII. Every computer can handle the ASCII encoding, and accordingly display the letters, numbers, and punctuation symbols of English text.

Each ASCII symbol is 7 bits long (for this reason, standard ASCII is sometimes called "7-bit ASCII"). For example, the ASCII code for "A" is 1000001, and the ASCII code 0111011 represents a semi-colon (";"). Some extensions to ASCII have been proposed, but we won't worry about them here.

1.1.2 Unicode

Computers were largely invented in English-speaking countries (Britain and America). However, there's a whole world out there, and they're digital now too. This raises a problem with the venerable old ASCII encoding. How are we to represent the alphabets of other languages now? There's no Greek letter "θ" in ASCII, nor is there a German "umlaut" decoration (as in "ö" or "ä").

To bring text encodings into the global village, a new standard emerged: *Unicode*. Unicode uses at least 16 bits (instead of 7 or 8) and thus can store over two hundred and fifty times as many symbols as ASCII. Unicode provides encodings for most the world's languages including Greek, Japanese, Arabic, English, and anything else you can think of. Find out about the richness of Unicode online at `http://www.unicode.org`.

1.2 Variable-Width Encodings

So far, all the encodings we've seen have been *fixed width*: each symbol in the start language has been encoded by exactly k bits in the target language (binary). Such an encoding is very easy to *decode*: simply replace every k bits by the appropriate source letter.

However, the ease of decoding comes at a price in that many extra symbols are used when they don't really need to be. For example, using the fixed-width encoding as shown in Example 1.2, the encoding of word 'AAAABBBB' is '000000000000001001001001', which is 24 bits long. This raises the following question:

> *Is there an encoding scheme that uses fewer symbols (on average) than a fixed-width encoding, but can still be reliably decoded?*

The answer is yes. Let's consider the same source alphabet as seen in Example 1.2.

Example 1.3
Now consider the following encoding:

$$
\begin{aligned}
A &\mapsto 1 \\
B &\mapsto 01 \\
C &\mapsto 001 \\
D &\mapsto 0001 \\
E &\mapsto 00001 \\
F &\mapsto 000001 \\
G &\mapsto 0000001 \\
H &\mapsto 00000001
\end{aligned}
$$

This encoding looks funny, but let's see what happens for the word $AAAABBBB$. Now the encoding is '111101010101', which is only 12 bits long, half as long as the encoding using the encoding scheme in Example 1.2.

Intuitively, if we use fewer bits to encode frequently used symbols and more bits for the less-frequently used ones, the average length of a word would be shorter than that using fixed-width encoding. However, being able to encode words are not the only goal; all the encodings we use must be easily *decoded*, which is what makes the task of finding better encodings challenging.

Example 1.4
Consider the following encoding.

$$A \mapsto 01$$
$$B \mapsto 0101$$

We receive the (encoded) string 01010101. What was the original message? According to our encoding, the original message could have been any of $AAAA$, AAB, BAA, ABA, or BB. If "$AAAA$" means "attack the enemy now" and BB means "we've reached a truce, call off the attack", we are facing a serious problem.

To avoid this problem, we introduce the idea of a *prefix code*.

Definition 1.2 (Prefix Code)
In a prefix code, no symbol's encoding is a prefix of any other symbol's encoding.

Prefix codes prevent the type of ambiguity that occurred in example 1.4. From now on, we will only consider prefix codes. It can be shown that restricting our attention to prefix codes is fine since there is always a prefix code with the same goodness as a particular non-prefix code. Thus, we effectively won't lose any good encodings even though we're only going to create prefix codes.

1.3 Description Length

It turns out the question about how to design an efficient encoding is closely related to work in a branch of Engineering Science known as *Information Theory*. This field studies encodings from a similarly parsimonious viewpoint. By investigating the mathematical properties of quantities such as the *average description length* of an encoding, Information Theory can tell us if our encoding uses strings that are as minimal as they could be. Such a question was particularly important during the late 1940s, with the emergence of refined communications technology at the start of the cold war. If an operative is in a hostile environment, governments want to use as few symbols as possible to communicate important information.

In addition to this quandary, Information Theory addresses the problem of *bandwidth* in the Information Age. Although we have vast interconnected networks of high-speed conveyors at the heart of the Internet, there's inevitably yet more e-traffic that would ideally be supported. Using an encoding of minimal average description length means that our data is optimally compressed for transfer. This is the basis of today's compression algorithms, such as those employed by Windows (WinZip) and the GIF image format (LKZ compression). The father of Information Theory, Claude Shannon, may not have envisioned such a use of his original theories, but they are nonetheless at the heart of research on encoding schemes still today.

The point of this section is to introduce you to two of the most famous contributions of Information theory: Shannon's formula for *minimal average description length* (or *entropy*) and *Huffman encodings*. We'll start with trying to measure the description length of an encoding (with the goal that shorter description lengths are better).

The *description length* of an encoding is the ration of the length of output to input strings. This idea is based on viewing an encoding as a "black box" that accepts input strings written in a source alphabet and outputs equivalent strings encoded in the target alphabet.

Definition 1.3 (Description Length for a Word)
As a formula, for any input word, w, we can compute the description length of an encoding ε with respect to w as:

$$\mathrm{DL}_{\varepsilon}(w) = \frac{|\varepsilon(w)|}{|w|}.$$

Here, the function $|\cdot|$ gives the length of a string (in symbols).

Better encodings are ones with a shorter description length. Such encodings save bandwidth over more verbose ones.

Example 1.5
Consider the encoding of A through H in Example 1.2 (page 3). For the word $w = AAAABBBB$, the description length of this encoding ε_1 (with respect to w) is

$$\mathrm{DL}_{\varepsilon_1}(w) = \frac{3 \cdot 8}{8} = 3.$$

Consider the encoding of A through H in Example 1.3 (page 4). For the same word w, the description length of this encoding ε_2 (with respect to w) is

$$\mathrm{DL}_{\varepsilon_2}(w) = \frac{4 \cdot 1 + 4 \cdot 2}{8} = 1.5.$$

Thus, the encoding of the word w is half as long in ε_2 as in ε_1.

The examples above show a crucial insight into preserving bandwidth. If we know in advance that our input alphabet has 2 symbols, we simply use 1 bit to encode them. But what about the case where there are 8 symbols, but only 2 of them actually occur with any frequency to speak of? We can't use just 1 bit to encode these symbols, because what do we do when the (rare) case of one of the other symbols arises? On the other hand, it doesn't make sense to use longer encodings for the frequent symbols, since this clogs up our network unnecessarily.

The solution is to use shorter encodings for more frequent symbols and longer encodings for less frequent ones. In this way, we can approximate the best case where the infrequent symbols never occur and we don't waste any bandwidth waiting for them.

1.3.1 Average Description Length

Definition 1.3 gives us a way to calculate the description length of an encoding for any particular input word. In general, we are interested in the *average description length* for all possible input words.

Definition 1.4 (Average Description Length for Finite Languages)
Given a source language, L_s, which has exactly K words, w_1 through w_K, and an encoding ε for the symbols of the language, the average description length for L_s is:

$$\begin{aligned}\mathrm{ADL}_{\varepsilon}(L_s) &= \frac{\sum_{j=1}^{K} \mathrm{DL}_{\varepsilon}(w_j)}{K} \\ &= \frac{1}{K} \cdot \sum_{j=1}^{K} \frac{|\varepsilon(w_j)|}{|w_j|}.\end{aligned}$$

The problem with this formula is that, in general, the source language will have an infinite number of words. In this case, we need to consider the *average frequency* of each source symbol in the language.

To see how this helps us, note that we can break the formula in defintion 1.4 down as follows. Say our input alphabet has exactly N symbols: $c_1, \ldots, c_N$. Each word in the input language is composed of some combination of the input symbols. We can capture this using the notation $P(c_i|w_j)$, which is the probability that the letter c_i appears in the word w_j of the language (this will always be either 0 or 1).

Thus we can rewrite definition 1.4 using the fact that encodings of words are simply the concatenation of encodings of symbols. We get:

$$\begin{aligned}\mathrm{ADL}_\varepsilon(L_s) &= \frac{1}{K}\cdot\sum_{j=1}^{K}\frac{|\varepsilon(w_j)|}{|w_j|}\\ &= \frac{1}{K}\cdot\sum_{j=1}^{K}\sum_{i=1}^{N}\frac{|\varepsilon(c_i)|\cdot P(c_i|w_j)}{|w_j|}\\ &= \sum_{i=1}^{N}|\varepsilon(c_i)|\cdot\Big(\sum_{j=1}^{K}\frac{P(c_i|w_j)}{K\cdot|w_j|}\Big).\end{aligned}$$

It turns out the quantity in parentheses here is exactly $P(c_i|L_s)$, i.e. the probability that c_i appears with respect to the entire source language. Thus, we can re-write our formula for Average Description Length without reference to the number of words in the source language. The formula in definition 1.5 works for both finite and infinite source languages.

Definition 1.5 (Average Description Length)
Given a source language L_s, which has N symbols c_1 through c_N, each symbol occurs with probability $P(c_i|L_s)$, for any encoding ε, the *Average Description Length* of ε for L_s is:

$$\mathrm{ADL}_\varepsilon(L_s) = \sum_{i=1}^{N}|\varepsilon(c_i)|\cdot P(c_i|L_s).$$

1.3.2 Entropy

Given a particular input language (with associated probabilities of letters), we want to find an encoding ε that has *minimal* average description length (when compared to all other possible encodings).

Suppose our target language has M symbols in its alphabet. Claude Shannon, often called the "Father" of Information Theory, showed that the minimal value of the quantity in definition 1.5 occurs exactly when $|\varepsilon(c_i)| = -\log_M(P(c_i|L_s))$. Note that this means we should use shorter target strings to encode more frequent symbols. This minimal description length is called *Entropy*.

Definition 1.6 (Shannon's Entropy Measure)
Suppose L_s has N symbols c_1 through c_N and the target language L_t has M symbols. The minimal average description length, or *Entropy*, of any encoding of L_s into L_t is

$$E = -\sum_{i=1}^{N}P(c_i|L_s)\cdot\log_M(P(c_i|L_s)).$$

Example 1.6
Suppose that we have a class with several students, and we want to encode the information that each student has a certain hair color (in binary). Let's say we're working with four possible hair colors, having probabilities as follows:

$$\begin{aligned}\text{Brown} &\mapsto 1/2\\ \text{Black} &\mapsto 1/4\\ \text{Blonde} &\mapsto 3/16\\ \text{Red} &\mapsto 1/16\end{aligned}$$

In this case Shannon's formula gives us that the entropy for this situation is:

$$-\frac{1}{2}\cdot\log_2(\frac{1}{2}) - \frac{1}{4}\cdot\log_2(\frac{1}{4}) - \frac{3}{16}\cdot\log_2(\frac{3}{16}) - \frac{1}{16}\cdot\log_2(\frac{1}{16}) = 0.5 + 0.5 + 0.45 + 0.25$$
$$= 1.7$$

Note that if we use a fixed-width encoding we need 2 bits for each symbol, so the goodness is exactly 2. Let's consider what this means. If we want to transmit the haircolor of 100 students as a binary string, an optimal variable-width encoding will need 170 bits, whereas a simple fixed-width encoding will need 200 bits. Thus, we'll save 30 bits of bandwidth (15%) by using the variable-width encoding.

1.4 Huffman Codes

Shannon's formula (definition 1.6) gives us a numeric measure for the average description length of an optimal encoding. But how do we find such an encoding? This problem plagued engineers for a while, until David Huffman devised a failproof way of generating optimal prefix codes in 1952. Today, such codes are simply known as *Huffman codes*.

To describe Huffman's algorithm, we need to know about a canonical data structure known as a *binary tree*. The definition of a binary tree is *recursive* (as with most of the important definitions in Computer Science). A binary tree is composed of one or more *nodes*. Inner nodes have two subtrees hanging off them (a *left* subtree and a *right* subtree). *Leaf* nodes are special in that they have no children. The *root* node is also special since it starts the tree. Binary trees can have various labelings, associated with either the nodes and/or the branches (connections between nodes). A typical binary tree is exhibited in figure 1.

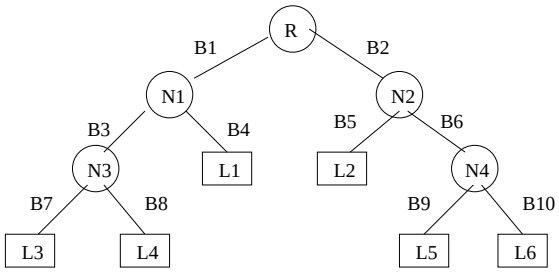


Figure 1: A binary tree. The root is labeled "R", inner nodes are labeled "N1" through "N4", leaves are labeled "L1" through "L6", and branches are labeled "B1" through "B10".

A *Binary Encoding Tree* is a binary tree where leaf nodes are labeled with the symbols of the source alphabet and branches are labeled with either "0" (left) or "1" (right). Each path from the root node to a leaf represents the encoding of the symbol contained in the leaf.

A *Huffman tree* is a binary encoding tree with even more special properties. In a Huffman tree, leaves are annotated with both input symbols and their associated probabilities. Inner nodes are labeled with the sum of probabilities of their left and right children. In addition, there is a "left-to-right" and "top-left-down" ordering. At each level k in the tree, the node labels go from lowest to highest as we go from left to right. Additionally, the leftmost node at level k is always greater than all nodes at level $k+1$. An example Huffman tree is shown in figure 3 (f). According to this tree, for example, the encoding for the letter "c" is 100.

Given a source language to encode into binary, we can produce a Huffman code as follows. We start with the leaves (the symbols of the source language paired with their probabilities) in a priority list. We take two of the symbols having the lowest probability (the two at the head of the priority list), and pair these leaves

under an inner node labeled with the sum of their probabilities. This new structure (a Huffman tree of height 1) is then put back in the priority list (in the appropriate position), replacing the two leaves that formed it. Next, we take the structures with the lowest probability again, and combine these similarly. *Note*: at this and successive stages, we may be combining one or more non-leaf structures. We continue this process until there is only a single Huffman tree left. This Huffman tree gives us our encoding. This algorithm is given in figure 2.

Input: alphabet symbols $A_1, ..., A_n$ with associated probabilities $P_1, ..., P_n$
Output: an optimal encoding for that alphabet
1. Initialize priority list L to the pairs $\{(P_{k_1}, A_{k_1}), ..., (P_{k_n}, A_{k_n})\}$.
2. WHILE (L has more than one item)
3. remove the first two items (with minimal probabilities) from L
4. combine these two items into a Huffman tree
5. place the new Huffman tree to the appropriate position in L, according to the probability of the root node of the new Huffman tree.
6. END WHILE;
7. RETURN the Huffman tree (the single item left in L.

Figure 2: Algorithm for obtaining a Huffman code.

Example 1.7
Figure 3 shows the steps involved in generating a Huffman code for the alphabet A through F with associated probabilities given in the table below.

	A	B	C	D	E	F
probability	0.45	0.13	0.12	0.16	0.09	0.05

The Huffman code of the language is:

$$A \mapsto 0$$
$$B \mapsto 101$$
$$C \mapsto 100$$
$$D \mapsto 111$$
$$E \mapsto 1101$$
$$F \mapsto 1100$$

1.5 Glossary

Character A symbol (letter) of a language.

Character Data Also known as *text*. Data made up of symbols in a natural language (such as letters of English).

Encoding A mapping from the words in a source language to words in a target language. The mapping consists of concatenating encodings of the source letters appropriately.

Encoding Tree A representation of an encoding as a tree data structure. The encoding of each letter is given by the symbols labeling the path from the root of the tree to the leaf containing that letter.

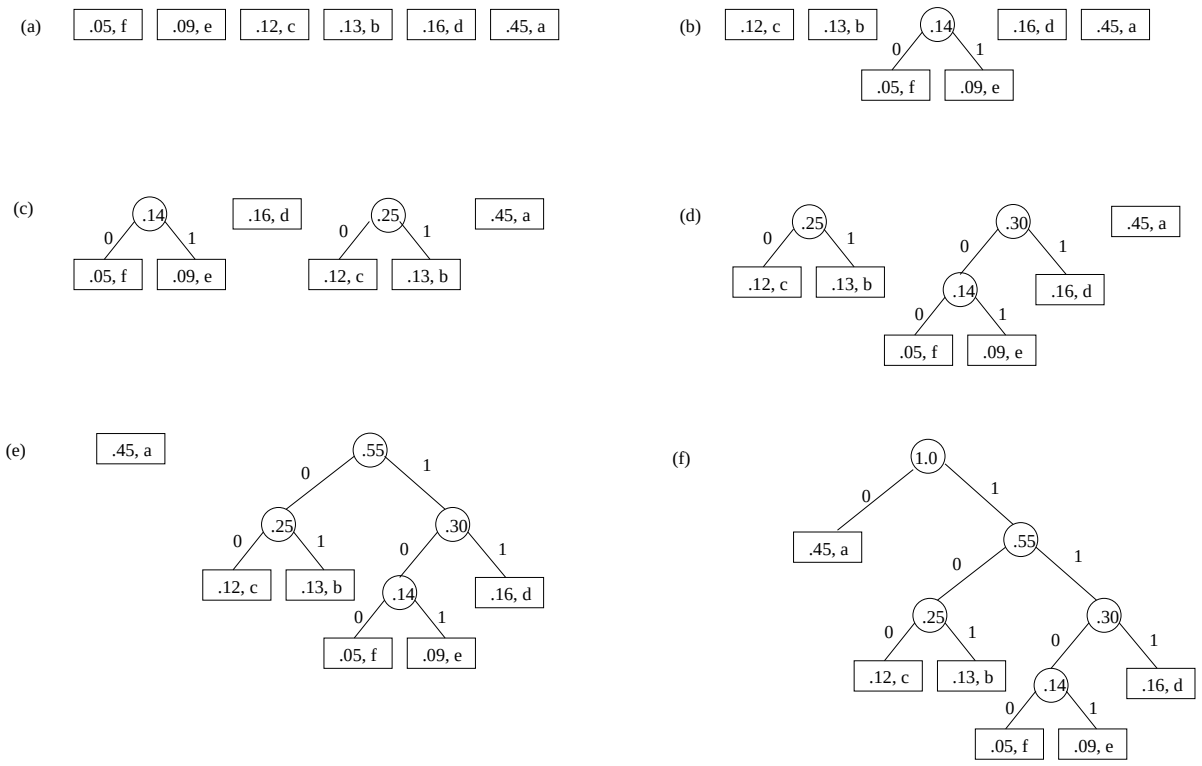


Figure 3: Example of Huffman encoding. This example appears in [2].

Fixed-Width Encoding An encoding where every source letter maps to a target word having the same length.

ASCII *American Standard Code for Information Interchange*. An 8-bit fixed-with encoding of character data into binary.

Unicode A standard for encoding international character data into binary. Some common flavors: UTF-8, UTF-16, UTF-16BE, UTF-32. For more information see:

`http://www.unicode.org/glossary/.`

Prefix Code An encoding where no source letter maps to any prefix of the encoding for any other letter.

(Average) Description Length The average length of the encoding of source letters, based on their occurrence probabilities.

Entropy Also known as *minimal average description length*. A lower bound for the average length of the encoding of source characters given a probability distribution for these source characters.

Huffman Code A prefix code having minimal average description length, generated by the procedure in algorithm 2.

Huffman Tree A tree where the labels along the path from the root to any leaf give the encoding for the source letter contained in that leaf.

2 Numeric Encodings

So far, we have seen how to encode text as binary for storage in a computer. But what happens if instead of storing the letter "1" we want to store the number "1" (so that, for example, we can perform numeric calculations)? Clearly ASCII won't work for this, as "1" plus "1" is not the text encoding for "2".

In some computer systems, integers are simply stored using their natural representation as binary (plus one bit for the sign). Thus knowledge of the binary and associated number systems is crucial for an understanding of computer arithmetic. For a refresher on binary arithmetic, work tutorial 4 ("Binary and Hexadecimal Arithmetic").

In the next sections, we'll see how numbers are represented within a computer. Numeric encodings are always *fixed width*. Their width depends on the particular computer and/or application, but will always be some multiple of Bytes.

2.1 Representing Integers

2.1.1 Signed and Unsigned Integers

The most obvious representation of integers would be to store their binary string in a computer. So, the decimal number 12 (for example), would be stored as 1100. Since numbers are stored using fixed width encodings, we may have to add leading zeros; if we use 8-bits, the decimal number 12 would correspond to 00001100. Commonly, 32 bits are used for numbers today.

This representation is called *Unsigned Integers*. It works for positive integers (the "unsigned" ones), but not for negative integers. The need to represent all integers (not just positive ones) led to competing representations; the most common ones are *Signed Integers* and *Twos Complement*. We will see the Twos Complement representation later in §2.1.2. Signed integers are a simple variation on the unsigned integer representation — the highest order bit is reserved as a *sign bit*. A value of 0 says the integer represented in the remaining bits is positive; a value of 1 says the integer represented in the remaining bits is negative.

However, even for this most "obvious" representation, things are not that simple. Figure 4 shows why. We read binary integers from left-to-right, and within a single Byte this is the way they are stored. But for multi-Byte widths, there are two possible representations: *big endian* and *little endian*. In a big endian representation, the least significant Byte of the number appears last (i.e. as the rightmost Byte). In a little endian representation, the least significant Byte appears first (i.e. as the leftmost Byte).

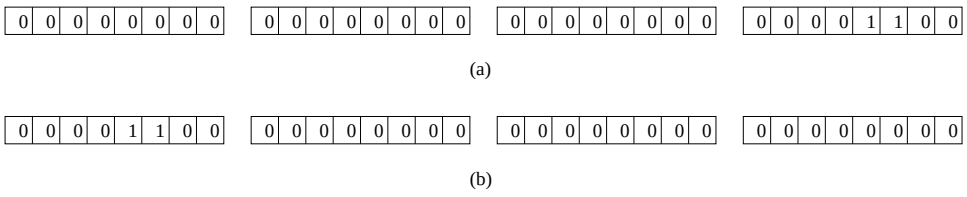


Figure 4: 32-bit (4 Byte) Representation of the decimal number 12. Part (a) shows the big endian byte ordering; part (b) shows the little endian byte ordering.

The distinction between signed and unsigned integers and big/little endian should further bring home an idea underlying this topic: *it is very important to know what encoding is used since different encodings may have very different meanings*. Thus, if you are told a binary string is ASCII, it has a very different meaning from if you are told it is a signed 8-bit integer. Computers are only as good at this distinction as their programming. For example, try opening a binary file (such as an executable program) as text in Notepad. You get gibberish!

2.1.2 Two's Complement Integers

If you think about it, multiplication by 2 in the ordinary binary encoding is very easy. This just requires a *left shift* operation: move all the bits to the left by one (and shift in a 0 at the right end). This is a nice property. Division by 2 is just as simple.

On the other hand, multiplication by -1 is rather complex. This is because using the highest order bit for a sign bit is very intuitive for us, but does not work very efficiently as a computer circuit. This means that subtraction is harder than it should be, and that negating numbers is also harder than it should be.

Example 2.1
The 8bSI representation of 12 is 00001100. The 8bSI representation of -12 is 10001100. A bit-wise addition operation on the two numbers gives us 00001100 + 10001100 = 10011000. The result, if we treated as a 8bSI representation, is -24, not 0, as we expected.

This is not an ideal state of affairs, since subtraction and negation are very common in computer operations. Thus, we would like our integer encoding to allow a more efficient negation operation.

The encoding for binary numbers that is most widely used today has this property. This encoding is called the *Two's Complement* encoding (2C for short). Positive integers represented in 2C have exactly the same format as for unsigned integers. The difference is in how negative integers are stored. In the 2C encoding, a number and its negation always add up to 2^{N+1} where N is the width of the encoding. This means that adding a number and its negation effectively "zeros out" the representation, which gives nice properties (see question **??** below).

Since it is important to have a value for the number of bits we are interested in, 2C is often called "N-bit" 2C, or Nb2C. So, for example, if we know the width we want to use is 8 bits, we will use the encoding 8b2C. Note that since the negative of a number is a key component for 2C, the encoding is always used for signed integers.

Definition 2.1
Given an Nb2C number $b_{N-1} \cdots b_1 b_0$, the value (in decimal) of the number represented is

$$-b_{N-1} \times 2^{N-1} + b_{N-2} \times 2^{N-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0.$$

Note that the leading negation in definition 2.1 ensures our 2C property that "a number and its negation always add up to 2^N".

Example 2.2
The value of the 4b2C number 1101 in decimal is

$$-1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = -8 + 4 + 1 = -3.$$

2.2 Representing Real Numbers

So much for integers. How about real numbers? In general, a real number may involve an infinite, non-repeating sequence of digits (such as π). Obviously we can't represent an infinite sequence of digits in a computer. Instead, what we actually represent are called *floating point* numbers. These are approximations to real numbers, and essentially function like the scientific notation used in representing real numbers.

A computer uses a different, binary version of scientific notation to represent floating point numbers. Given N working bits, they are divided into a sign bit, an unsigned *exponent*, E, with a bias of B and an unsigned *fraction*, F (figure 5). Inside a computer, the number represented in this way has the following value:

$$(+/-)(1 + F) \times 2^{E-B}.$$

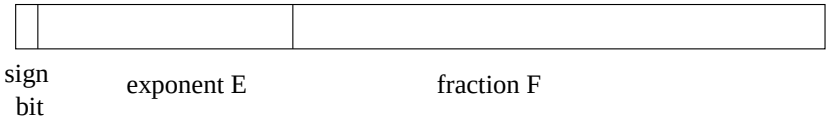


Figure 5: Floating Point representation.

This representation is the most common in use today and is called *normalized floating point* (or simply *floating point*). We will denote this representation FP for short.

The parts of the FP encoding have the following meaning:

- The sign bit determines if the number is to be interpreted as positive (sign bit = 0) or negative (sign bit = 1).
- The fraction is stored as an unsigned binary string and represents a value between the real numbers 0 and 1. In other words, the fraction is the binary digits appearing *after* the decimal point in a binary expansion. The fact that the binary number has to be written so that the fraction is between 0 and 1 is why the representation is *normalized*. For help with representing fractions in binary, see tutorial 4 (*Binary and Hexadecimal Numbers*).
- The exponent is called a *biased* exponent, because its value is obtained in terms of the bias, B. The bias is a given quantity for a particular FP encoding (see below).

Note that the fraction F represents a number between 0 and 1. This number must be added to 1 to get the true value of the numeric part of the floating point number. The value 1+F is called the *significand*.

These decisions may seem somewhat arbitrary to you (for example, why have a bias at all?), but they are carefully chosen properties of modern FP encodings and designed to facilitate both scientific computation and everyday floating point arithmetic. The floating point standard was finalized in 1985 and is known as *IEEE-754-1985*. The standard has an interesting history, which can be found starting at reference [11].

As with 2C, FP relies on having N bits to work with. We will call the encoding for a particular N *NbFP*. In real computers, N is usually 32 (single precision) or 64 (double precision). We term these encodings 32bFP and 64bFP. The breakdown of bits for 32bFP and 64bFP is given in the following table. This is part of the IEEE floating point standard.

	32bFP	64bFP
bits in sign	1	1
bits in exponent	8	11
bits in fraction	23	52
bias	127	1023
Total bits	32	64

Example 2.3
The 32bFP encoding

1 10000001 01000000000000000000000

represents the number -5. This is because the value of the exponent as an unsigned integer is $1\times2^7+1\times2^0 = 129$. Thus the actual exponent is $129 - 127 = 2$. The value of the fraction as an unsigned integer is $0 \times 2^-1 + 1 \times 2^-2 = 1 \times 1/4 = 0.25$. Thus the value of the significand is 1.25. The sign bit (the first bit) indicates this is a negative number, so its true value is $-1.25 \times 2^2 = -5$. (Example from [5].)

3 Multimedia Encodings

Multimedia is a staple of the web today — pictures, sounds, streaming video and audio. But it wasn't always so. Each of these media takes significant space to store as bits. Over slow internet connections (*every* internet connection used to be "slow") any of these formats are impossible to wait for — some requiring years to download.

Improvement in both the speed of internet communication and the compression of multimedia has meant that the web is now alive with these formats. In this section, we'll introduce you to the basic concepts and measures involved in encoding multimedia as strings of bits.

3.1 Colors

First, we'll look at common color encodings. There are several of these in use today. All color encodings begin with the same basic precept: *colors are generated from a small number of primary colors*. We know this from playing with paints and lights in grade school, and computer monitors, TVs, printers, digital photographs, etc. are designed to produce only primary colors and intermix them in various ways. Thus the primary colors are the "input alphabet" for color encodings.

First, we'll learn some of the basic terminology involved.

Definition 3.1

1. A *pixel* (or *Picture Element*) is the smallest unit of color produced by a computer monitor. Thus, pixels have become the de facto unit of measure for computer images (and colors).

2. A *raster* is one horizontal row of pixels, representing one line of a computer monitor or image.

3. The range of colors that can be produced by mixing the primary colors available in the different intensities available is known as the *Color Space*.

Any color encoding must specify the intensities for each of the primary colors for each pixel in the encoded image. There are several schemes in use to encode intensities for a single pixel.

We'll begin with the simplest (and most common) color encoding: RGB (which stands for the primary colors Red-Green-Blue). RGB colors are used widely on the web and in most computer monitors, for example. When a web page is touted as "color-safe", that means it assumes the 24-bit RGB encoding discussed in the next section.

3.1.1 RGB

One of the most common color encoding schemes in use today is the *RGB* ("Red-Green-Blue") one. In grade school you may recall learning about "primary" colors, which can be mixed to form all the other colors. Red, green, and blue are such colors: all other colors can be seen as a combination of these three colors under the "additive" color mixing scheme. An example of this scheme is three colored lights focussed on a single spot. When all three lights are at zero intensity, we get black. When each light is turned up by itself, we get the three colors red, green, or blue. When the lights are turned up in combination, we can obtain all the other colors – including white (all lights full intensity). This is exactly the type of thing that happens with pixels on a computer monitor.

In the RGB encoding, this fact is encoded in a binary string which represents how much red, blue, or green intensity to have. One byte (8 bits) is used for each color, so this is a 24-bit (fixed) encoding. We usually write the encoding in hexadecimal, so we only have to write 6 digits (instead of 24 with binary, or 9 with decimal), although the encoding is, strictly speaking, a binary one. If a hexadecimal number appears

in HTML and/or CSS (see topic II), it is displayed with a "#" sign in front of it to signal that the following digits are to be interpreted as hexadecimal (as opposed to an ordinary decimal number). Thus the RGB color #FF0000 represents maximum red; #00FF00 represents maximum green, and #0000FF represents maximum blue.

The color #000000 displays as pitch black (the "absence" of any of our primary colors). On the other hand, the color #FFFFFF displays as pure white (the sum of all primary colors together). The colors in between adjust to this scale, so that a larger number for a primary color means "brighter" and a smaller number means "darker". Thus, #550000 is a darker red than #AA0000.

3.1.2 Other Color Spaces

RGB is not the only color space that is in use today.

CMY and CMYK Monitors naturally use an additive color scheme, and the encoding RGB reflects this. In contrast, computer printers naturally use a *subtractive* color scheme, where "no primary colors" is white, "all colors mixed" is black, and colors in between are obtained by subtracting some of the primary color intensities from black. On this scheme, the primary colors most useful are not red, green, and blue, but rather Cyan, Magenta, and Yellow. Thus this color space is known as *CMY*. As you can perhaps guess from the opposite representations of white and black (as opposed to RGB), a pixel that is represented by RGB values r, g, and b has corresponding C, M, and Y components $255 - b$, $255 - r$, and $255 - g$ (respectively).

In practise, mixing the three CMY colors produces a "muddy brown" instead of a clean black. Thus, printers will have a fourth color: pure black. This color space is then known as CMYK ("K" for black). For each CMY color, a certain amount of the primary colors is subtracted out and replaced by black. This noticably improves the quality of the colors that result on paper.

HSV The human vision system is not equipped to break colors down into their RGB or CMY components directly. We see colors in terms of the more qualitative measurements shade, brightness, and whether the color is pastel (lots of white) or vivid (not much white mixed in). These qualitative properties are quantified in the *HSV* or *Hue-Saturation-Value* color space.

Hue describes the shade of a color, which is basically where that color can be found on a circular representation of the rainbow. Saturation describes how "pastel" the color is, which is essentially how that color compares to white. Value describes the brightness of the color, which is a measure of how much light it reflects.

The relationship between RGB and HSV components are shown in figure 6. The outer cube represents the colors possible in a standard RGB encoding (note the three axes are labeled red, green, and blue). The origin of this cube (labeled 0) is black, and the opposite corner (labeled #FFFFFF) is white. Points in this cube can be described either with the usual Euclidean co-ordinates (which will be RGB values), or with polar co-ordinates on a circular plane whose center and angle of inclination are determined by its distance from the white and red vertices of the cube.

YCrCb Television systems (and some digital image and video formats) presuppose a different color space, known as *YCrCb*. A linear transformation on the RGB components of a color yields a *luminance* value (the "Y" — this conveys brightness) and two *crominance* values. These convey the color hue (its distance from full red) and saturation (its distance from full white). This color scheme is similar to HSV. YCrCb is used in many image and video compression schemes (such as JPEG and MPEG) because it performs better under compression (than either RGB or HSV).

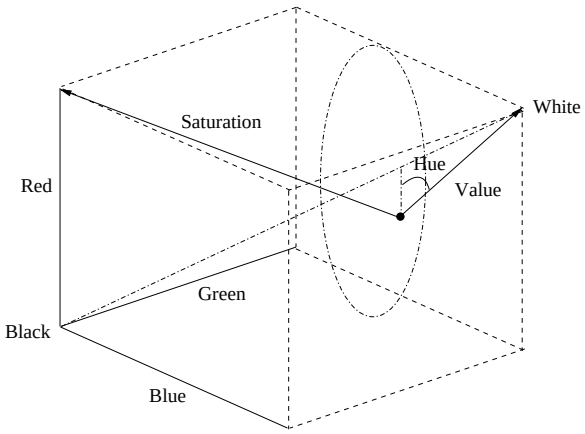


Figure 6: RGB color space represented as a cube and HSV coordinates for a sample point. Modified from [1].

3.2 Images

A *digital image* consists of specifications for creating an image on a computer monitor (or printer). Essentially, the encoding for an image must specify two things: (1) how large the image is (as a rectangle of pixels), and (2) what color each pixel should be. Thus, the simplest image format to imagine simply contains two numbers (the height and width of the image in pixels) and then a corresponding number of pixel specifications in a known color space.

Example 3.1
Suppose we want to represent images up to 256×256 pixels large. In this case, we can use two bytes of header information to store the image height (first) and then width (next byte). Note that the actual height (or width) will be 1 + the stored value. Colors can be represented as RGB specifications, each of which takes 3 bytes. Thus, an image that is 256×256 will have the following size (according to our simple encoding):

$$\begin{aligned} \text{ImageSize} &= 2 + (256 \cdot 256) \cdot 3 \text{ bytes} \\ &\simeq 192 \text{ Kb} \end{aligned}$$

As the example shows, this simple representation makes even quite small pictures intractably large. Thus, much effort has gone into *compressing* images, either by using a cleverer encoding scheme, or by using direct compression methods such as Huffman encoding.

3.2.1 Indexed Images

A simple observation can cut the size of images significantly. For any particular image, we probably won't need the entire color space, since that image is likely to use only a small subset of the color space. The RGB color space allows about 16 million colors, whereas a large-ish image (1280×1024) has only about 1 million pixels. In practice, an image will usually not have a different color for every pixel. This leads to the idea of *indexed* representations for colors in an image.

For an indexed image, every distinct color in the image is assigned a number (or index). The image file then has three components: (1) header information specifying how large the image is in pixels, (2) a *color map* specifying which colors correspond to indexes, and (3) the indexes for the colors for the pixels of the image.

Example 3.2
Suppose we want to encode images up to 256×256, and we know in advance that (at most) these images have a thousand distinct colors in them. In this case, we will have two bytes in the header specifying the width and height, and every one of the thousand colors can be represented using 10 bits. Thus, a maximally-large image will have a represented size of:

$$\begin{aligned} \text{ImageSize} &= 2 \text{ bytes for the header} \\ &\quad + 1024 \cdot (10 + 24) \text{ bits for the color map} \\ &\quad + 256 \cdot 256 \cdot 10 \text{ bits for the image} \\ &\simeq 84 \text{ KB}. \end{aligned}$$

Thus, using an indexed encoding scheme, we have saved ourselves over 3 orders of magnitude in the resulting file size.

3.2.2 Common Digital Image Specifications

BMP BMP ("BitMapped Pixels") is the graphics file format used by the Windows operating system. BMP images are also very common throughout the web. BMP files are uncompressed, and have essentially the same format as simple indexed images above, namely, a header, a color map, and then a list of colors corresponding to pixels. In BMP pictures, 4 or 8 bits may be used for each index. One optimization the BMP format does include, is known as *run-length encoding*. Each line of image data begins with a byte that is either #00 (signaling absolute mode — the next bytes list the pixel colors by index) or a non-zero number, in which case it lists the number of pixels in the current raster that are the same color (this color follows the opening byte). For example, a line such as "00 03 45 56 67" indicates that the next three pixels have indexed colors #45, #56, and #67. A line such as "03 45" on the other hand indicates that the next three pixels all have indexed color #45.

For more detail about the BMP specification, see [10].

GIF GIF (Graphics Interchange Format) is CompuServe's proprietary image file format. This format specifies the header, color map, and indexed image data. Since 1989, the GIF format has also specified the use of single-pass LZW compression, which is a technique in the same class as Huffman encoding (lossless text compression).

JPEG The JPEG (Joint Photographic Experts Group) standard was the first to emerge that specified *lossy compression* for image data. Unlike Huffman or LZW compression, JPEG compression is not fully reversible, since parts of the input will be "lost". Unfortunately, the original JPEG specification does not specify the input file format to the compression algorithm, but JFIF ("JPEG File Interchange Format") is the most commonly used and is a simple indexed scheme.

JPEG compression assumes a model of the human visual system, and chooses what features to "lose" based on our limitations. JPEG compression involves *discrete cosine transformations* on input data and then removes some features of the transformed data according to the underlying model of what most humans cannot easily detect. The quality of the result depends heavily on the input image (e.g. how "uniform it is, whether it has sharp edges, etc.) as well as on the desired size of the output file. The smaller the output file, the more features are dropped from the transformed data.

Other (newer) lossy compression schemes involve *wavelets* or *fractals*, but because of its early standardization, JPEG remains the lossy compression of choice.

A JPEG file may be Huffman encoded after it has been through the transformation, to further reduce file size.

Example 3.3
Let's compare the various standards for a typical digital picture. The input picture is 114×168 pixels. The original picture is JPEG (shown in figure 7 far left). This picture can be saved as BMP and GIF with no loss of quality (but an increase in file size). The original picture can be compressed using the JPEG algorithm. (Four examples are shown in figure 7. The percent indicates relative "quality" of the image, a value that arises within the underlying model for the compression.) The file sizes for all the images are shown in table 1.

Format	File Size (in KB)
BMP	95.1
GIF	31.1
JPEG (original)	35.1
JPEG (75%)	9.3
JPEG (50%)	6.0
JPEG (25%)	3.7
JPEG (10%)	1.8

Table 1: Sizes for various images (in KB).

3.3 Sound

Now let's consider how to represent sound digitally. Sound is a physical phenomenon, which is visualized as a *wave* (see figure 8 (a)). A sound wave has an *amplitude*, which basically determines its "energy" (volume or depth), and a *frequency*, which basically determines its pitch. A sound wave is continuous, and arises when the air surrounding us is compressed. This compression is picked up by our ears (or a listening device).

To capture a sound wave digitally, *sampling* is used. This means that measurements are taken of the wave's amplitude and frequency at discrete time intervals (figure 8 (b)). If these intervals are small enough, the shape of the wave can be re-created using only the sampled values. Note that if the sampling is not taken at appropriate intervals, the wave will be "warped" on re-creation, meaning the original sound will not be re-created (figure 8 (c)). Thus, the rate at which samples are taken impacts the resulting reproduction.

In fact, given a wave whose frequency is ν cycles per second (Hz), we must sample at least $2 \cdot \nu$ times per second to ensure we can reproduce this wave from our sampling. The frequency $2 \cdot \nu$ is called the *Nyquist* frequency.

Example 3.4
Suppose we can sample at a rate of 10^3 times per second. What is the frequency of the highest pitch wave we can capture at this sampling rate? In this case, the answer is $\frac{1}{2} \times 10^3 = 500$ Hz. The range of sound that is audible to the human ear is roughly 20 to 20,000 Hz. Thus, we can't capture nearly enough sounds if we sample at this low rate.

The simplest encoding for sound just includes one amplitude value taken at every sample, for the entire length of the recording (in seconds). The number of bits that are used to record the amplitude is important, since the ability to represent more amplitude values will give us more "tonal depth" to our sample and resulting reproduction (although sampling at higher rates can make up for lower bit usage).

Sampling may be done in *stereo* (two channels) or *mono* (a single channel). Each channel requires separate amplitude value for each sample.

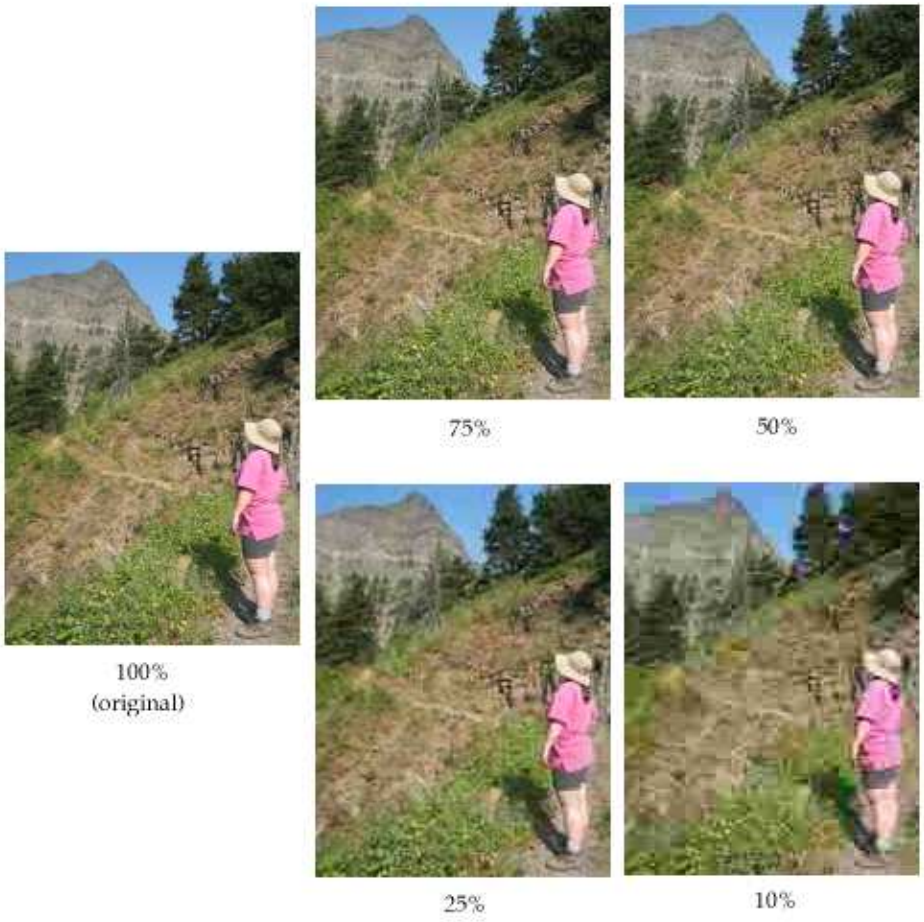


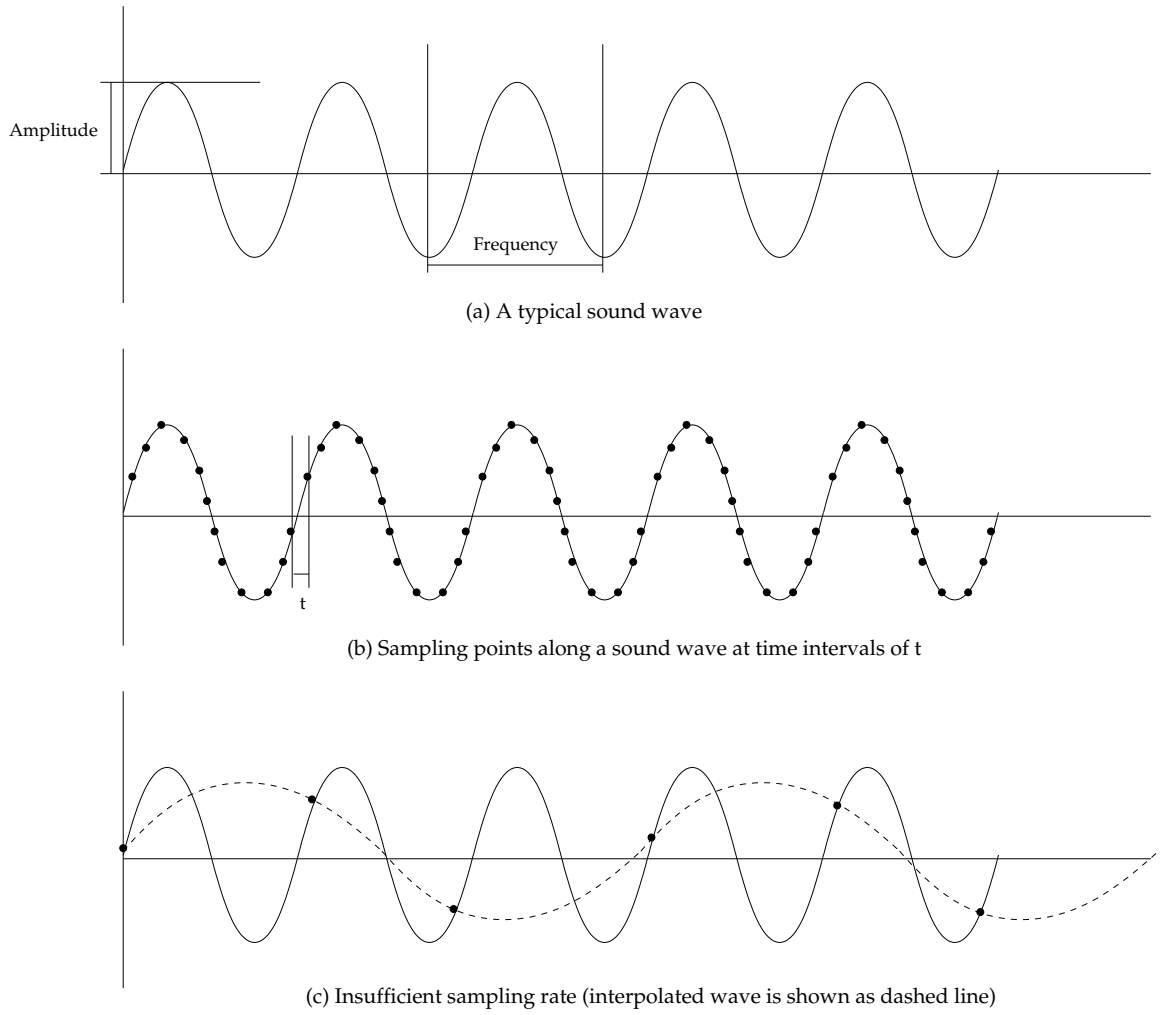
Figure 7: Examples of JPEG compression.

Example 3.5
Suppose we use a sampling rate of 50,000 Hz and use 16-bit numbers to represent amplitude. How large will the file be for a 5 minute song played in stereo? In this case, the answer is

$$\begin{aligned}\text{size in bytes / channel} &= 2\ \frac{\text{bytes}}{\text{sample}} \cdot 50000\ \frac{\text{samples}}{\text{second}} \cdot 60\ \frac{\text{seconds}}{\text{minute}} \cdot 5\ \text{minutes} \\ &= 30,000,000\ \frac{\text{bytes}}{\text{channel}} \\ &\simeq 30\ \frac{\text{MB}}{\text{channel}} \cdot 2\ \text{channels} \\ &= 60\ \text{MB total.}\end{aligned}$$

3.3.1 Streaming Audio

We often want to consider measurements of *streaming audio*: this is audio that "never ends". Thus, we do not include a time limit to the audio being sent, rather we must measure the (average) number of bits per second that the audio reproduction needs for all samples during that second. This simply means that we



(a) A typical sound wave

(b) Sampling points along a sound wave at time intervals of t

(c) Insufficient sampling rate (interpolated wave is shown as dashed line)

Figure 8: Sampling a typical sound wave.

think of size in terms of "bits per second" rather than "total bits".

Example 3.6
At a sampling rate of 50,000 times per second, using 16-bits per amplitude value, with two channels, what is the stream rate of the audio we produce? In this case, we get:

$$\begin{aligned}\text{bits per second} &= 50000\ \frac{\text{samples}}{\text{second} \cdot \text{channel}} \cdot 16\ \frac{\text{bits}}{\text{sample}} \cdot 2\ \text{channels} \\ &= 1600000 \frac{\text{bits}}{\text{second}} \\ &= 1.6\ \text{Mbps}.\end{aligned}$$

This is how fast your internet connection must be to receive this streaming audio signal. Table 2 shows some typical bandwidths for common internet connections. Would the set up you have at home be able to handle this stream rate?

Connection	Speed
Telephone modem	56 Kbps
ISDN	128 Kbps
DSL	1.5 – 8.4 Mbps
T1 line	1.5 Mbps
Cable modem	10 Mbps
Satellite	400 Kbps

Table 2: Common internet bandwidths.

The stream rate determines how much depth and variation we can reproduce; thus formats with a higher stream rate sound better than formats with a lower stream rate. The trade-off is that the higher stream rate requires more bandwidth to transmit.

3.3.2 Common Audio Formats

In this section, we'll briefly look at some common encodings in use.

WAVE The WAVE format is a Microsoft format for multimedia which specifies various header information and a data chunk. The data chunk encodes amplitudes at the sample rate indicated in the header. A detailed breakdown of WAVE files is available at:

`http://ccrma-www.stanford.edu/CCRMA/Courses/422/projects/WaveFormat/.`

The WAVE format is the embodiment of our simple encoding scheme for audio. The audio comes out uncompressed. CDs store music in a scheme very similar to WAVE.

Lossy Compression Schemes Better stream rates are achieved in modern audio encodings using various methods of *compression*. In theory, any compression technique could be used, and the effectiveness would be a tradeoff between the reduced stream rate and the need for more complex decoding (either hardware or software based). The most well-known format for audio in use today is MP3 (MPEG layer 3). MPEG is the standard for video developed by the Motion Picture Experts Group (more information is available at `http://www.mpeg.org`). The MPEG specification has several "layers", the third of which is used for audio signals; this specification has emerged quickly as a leader in audio encoding. The main appeal of MP3 is twofold: it achieves impressive compression of audio signals, and it has a very favorable licensing policy (unlike most other schemes).

In general, MPEG compression is based on a *perceptual coding scheme* (as JPEG was for images). This scheme involves a model of the human audio-visual system, and compression takes place by removing features of the signal that will (hopefully) be undetectable by the listener. For example, it is known that humans have difficulty distinguishing a softer tone that is played right near a significantly louder tone. MPEG-type compression simply removes these *masked* tones from the audio stream.

A distinct advantage of the MP3 coding scheme is that, while encoding a signal is relatively difficult, decoding an MP3 representation is quite simple. Since playback tends to be much more important on the web, MP3 is ideal in this case.

Table 3 gives some typical stream rates needed in MP3 to achieve some common audio qualities (adapted from [9]).

Quality	stream rate range (MP3)
CD	100+ Kbps
FM radio	50-100 Kbps
AM radio	20-50 Kbps

Table 3: Stream rates and their relative quality.

Other lossy compression schemes include: WMA (Windows proprietary compressed audio format) and OGG VORBIS (a relative newcomer). The OGG VORBIS is a completely open-source specification (unlike MP3), and can handle *variable compression rates* within a single encoding. Since some parts of an audio signal compress better than others, it is natural to want to compress these parts more. Find out more at `http://xiph.org`.

3.3.3 Sound Production (MIDI Format)

An interesting idea for encoding sounds that is completely orthogonal to the sampling methodology, is the idea of *digital sound production*. This idea is embodied in the MIDI format. A MIDI sound card is capable of producing sounds that mimic a wide range of instruments, including the human voice. To get an idea of the range of sounds MIDI cards can produce, see `http://void.vgmidi.com/midiinfo.html`. Under the section MIDI instruments, you'll find 24 piano-like sounds, 24 guitar-like sounds, synthetic sounds, various sound effects (heart beat, rain, telephone, ...), and many more. Basically a MIDI card puts all the instruments you can imagine at your disposal. The MIDI file format is then basically an encoding for writing musical scores where the score will be "played" by the MIDI hardware. Since MIDI sounds can be imitated by most sound cards, MIDI can in fact be played on almost any modern system, although it can sound dramatically different depending on the sophistication of the playback synthesizer. Since MIDI is essentially an encoding of the "notes" to be played by the instruments, MIDI files tend to be much smaller than sampled audio files.

On the web, you can find many resources for learning to compose MIDI scores, if you're interested.

3.4 Video

To encode video signals, the technologies for image and audio encoding are merged. Basically, a certain amount of the video signal is comprised of an audio signal (encoded as above), and the rest of the video signal involves transmitting *frames* (still-image snapshots) with a certain *frame rate*. If the frame rate is high enough, we perceive the frames as continuous video. Frame rates as low as 10 frames per second can seem continuous to human observers.

The most widely used compression scheme for video signals is that given by the Motion Picture Experts Group (MPEG); the principle for MPEG compression of video signals is basically the same as for MP3, namely, features in the signal that will not be noticed by a human viewer/listener are left out. These cannot be reproduced in the resulting video stream, so MPEG compression is *lossy*.

Whereas the bandwidth of audio signals is usually measured in Kbps (Kilobits per second), the bandwidth of video signals is measured in Mbps (Megabits per second), even when compressed. It is only the advent of widespread broadband internet connections that have enabled web use of video to emerge.

References

[1] Carey Bunks. *Grokking the GIMP: Advanced Techniques for Working with Digital Images*. New Riders Publishing, 2000.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT press, 1986.

[3] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991.

[4] Stephen Haag, Maeve Cummings, and Alan I. Rea, Jr. *Computing Concepts*. 1st edition. McGraw Hill, 2002.

[5] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[6] Thomas D. Schneider. "Information Theory Primer". `http://www.lecb.ncifcrf.gov/~toms/paper/primer/`

[7] Andrew S. Tanenbaum. *Structured Computer Organization*. 3rd edition. Prentice Hall, 1990.

[8] `http://www.asciitable.com`

[9] `http://ekei.com/audio`

[10] `http://www.dcs.ed.ac.uk/home/mxr/gfx/2d/BMP.txt`

[11] `http://grouper.ieee.org/groups/754/`

[12] `http://www.cl.cam.ac.uk/~mgk25/unicode.html`

[13] `http://www.unicode.org`

[14] `http://www.w3schools.com`