Chapter 3

Compression

In Chapter 1 we examined the fundamental unit of information, the bit, and its various abstract representations: the Boolean bit, the circuit bit, the control bit, the classical bit, and the quantum bit. Our never-ending quest for improvement made us want representations of single bits that are smaller, faster, stronger, smarter, and safer.

In Chapter 2 we considered some of the issues surrounding the representation of complex objects by arrays of bits. The mapping between the objects to be represented (the symbols) and the array of bits used for this purpose is known as a code. We naturally want codes that are stronger and smaller, i.e., that lead to representations of objects that are both smaller and less susceptible to errors. In this chapter we will consider techniques of compression that can be used for generation of particularly efficient representations. In Chapter 4 we will look at techniques of avoiding errors.

In Chapter 2 we considered systems of the sort shown in Figure 3.1, in which symbols are encoded into bit strings, which are transported (in space and/or time) to a decoder, which then recreates the original symbols.

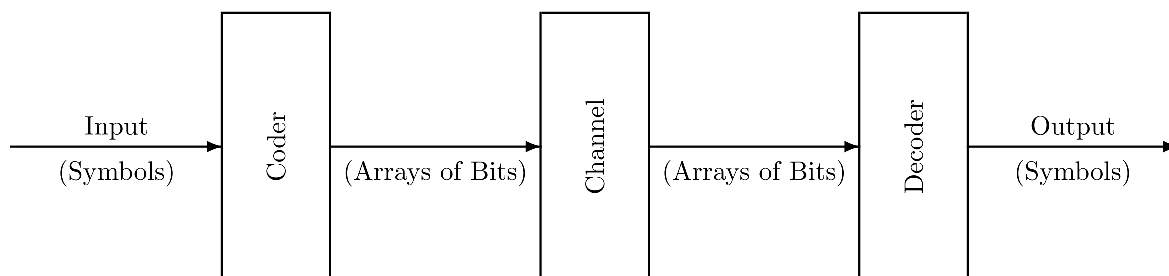


Figure 3.1: Generalized communication system

Typically the same code is used for a sequence of symbols, one after another. The role of data compression is to convert the string of bits representing a succession of symbols into a shorter string for more economical transmission, storage, or processing. The result is the system in Figure 3.2, with both a compressor and an expander. Ideally, the expander would exactly reverse the action of the compressor so that the coder and decoder could be unchanged.

On first thought, this approach might seem surprising. Why is there any reason to believe that the same

Author: Paul Penfield, Jr.
This document: http://www.mtl.mit.edu/Courses/6.050/2007/notes/chapter3.pdf
Version 1.4, February 12, 2007. Copyright © 2007 Massachusetts Institute of Technology

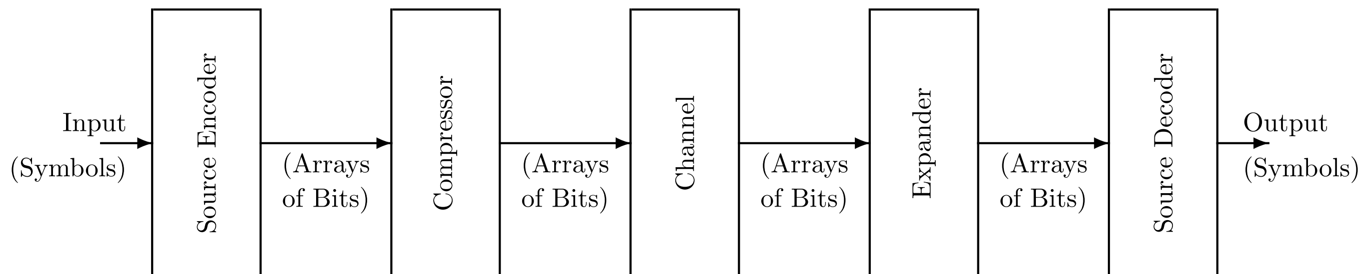


Figure 3.2: More elaborate communication system

information could be contained in a smaller number of bits? We will look at two types of compression, using different approaches:

- **Lossless** or **reversible** compression (which can only be done if the original code was inefficient, for example by having unused bit patterns, or by not taking advantage of the fact that some symbols are used more frequently than others)
- **Lossy** or **irreversible** compression, in which the original symbol, or its coded representation, cannot be reconstructed from the smaller version exactly, but instead the expander produces an approximation that is "good enough"

Six techniques are described below which are astonishingly effective in compressing data files. The first five are reversible, and the last one is irreversible. Each technique has some cases for which it is particularly well suited (the **best cases**) and others for which it is not well suited (the **worst cases**).

3.1 Variable-Length Encoding

In Chapter 2 Morse code was discussed as an example of a source code in which more frequently occurring letters of the English alphabet were represented by shorter codewords, and less frequently occurring letters by longer codewords. On average, messages as sent using these codewords are shorter than they would be using all codewords of the same length. Variable-length encoding can be done either in the source encoder or the compressor. A general procedure for variable-length encoding will be given in Chapter 5, so a discussion of this technique is put off until that chapter.

3.2 Run Length Encoding

Suppose a message consists of long sequences of a small number of symbols or characters. Then the message could be encoded as a list of the symbol and the number of times it occurs. For example, the message "a B B B B B a a a B B a a a a" could be encoded as "a 1 B 5 a 3 B 2 a 4". This technique works very well for a relatively small number of circumstances. One example is the German flag, which could be encoded as so many black pixels, so many red pixels, and so many yellow pixels, with a great saving over specifying every pixel. Another example comes from fax technology, where a document is scanned and long groups of white (or black) pixels are transmitted as merely the number of such pixels (since there are only white and black pixels, it is not even necessary to specify the color since it can be assumed to be the other color).

Run length encoding does not work well for messages without repeated sequences of the same symbol. For example, it may work well for drawings and even black-and-white scanned images, but it does not work

Figure 3.3: Flag of Germany (black band on top, red in middle, yellow on bottom)

well for photographs since small changes in shading from one pixel to the next would require many symbols to be defined.

3.3 Static Dictionary

If a code has unused codewords, these may be assigned, as abbreviations, to frequently occurring sequences of symbols. Then such sequences could be encoded with no more bits than would be needed for a single symbol. For example, if English text is being encoded in ASCII and the DEL character is regarded as unnecessary, then it might make sense to assign its codeword 127 to the common word "the". Practical codes offer numerous examples of this technique. The list of the codewords and their meanings is called a codebook, or dictionary. The compression technique considered here uses a dictionary which is **static** in the sense that it does not change from one message to the next.

An example will illustrate this technique. Before the the electric telegraph, there were other schemes for transmitting messages long distances. A mechanical telegraph described in some detail by Wilson[1] was put in place in 1796 by the British Admiralty to communicate between its headquarters in London and various ports, including Plymouth, Yarmouth, and Deal. It consisted of a series of cabins, each within sight of the next one, with six large shutters which could rotated to be horizontal (open) or vertical (closed). See Figure 3.4. In operation, all shutters were in the open position until a message was to be sent. Then all shutters were closed to signal the start of a message (operators at the cabins were supposed to look for new messages every five minutes). Then the message was sent in a sequence of shutter patterns, ending with the all-open pattern.

There were six shutters, and therefore 64 (2^6) patterns. Two of these were control codes (all open was "start" and "stop," and all closed was "idle"). The other 62 were available for 24 letters of the alphabet (26 if J and U were included, which was not essential because they were recent additions to the English alphabet), 10 digits, an end-of-word marker, and an end-of-page marker. This left over 20 unused patterns which were assigned to commonly occurring words or phrases. The particular abbreviations used varied from time to time, but included common words like "the", locations such as "Portsmouth", and other words of importance such as "French", "Admiral", "east", and "frigate". It also included phrases like "Commander of the West India Fleet" and "To sail, the first fair wind, to the southward".

Perhaps the most intriguing entries in the codebook were "Court-Martial to sit" and "Sentence of court-martial to be put into execution". Were courts-martial really common enough and messages about them frequent enough to justify dedicating two out of 64 code patterns to them?

As this example shows, long messages can be shortened considerably with a set of well chosen abbreviations. However, there is an inherent risk: the effect of errors is apt to be much greater. If full text is transmitted, a single error causes a misspelling or at worst the wrong word, which humans can usually detect and correct. With abbreviations, single errors could result in a possible but unintended meaning: "east" might be changed to "south", or more seriously, a single letter might be changed to "Sentence of court-martial to be put into execution" with significant consequences.

This telegraph system worked well. During one demonstration a message went from London to Plymouth and back, a distance of 500 miles, in three minutes. That's 13 times the speed of sound.

If abbreviations are used to compress messages, there must be a codebook showing all the abbreviations in use, that is distributed before the first message is sent. Because it is distributed only once, the cost of

[1] Geoffrey Wilson, "The Old Telegraphs," Phillimore and Co., Ltd., London and Chichester, U.K.; 1976; pp. 11-32.

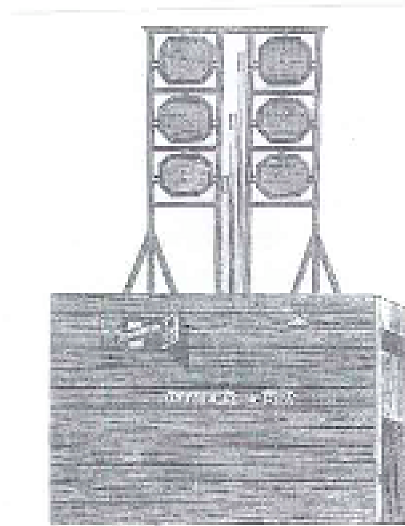


Figure 3.4: British shutter telegraph cabin, 1797, showing the six shutters closed, and an open window with a telescope to view another cabin (from T. Standage, "The Victorian Internet," Berkley Books, New York; 1998; p. 15).

distribution is low on a per-message basis. But it has to be carefully constructed to work with all messages expected, and cannot be changed to match the needs of individual messages.

This technique works well for sets of messages that are quite similar, as might have been the case with 18th-century naval communications. It is not well suited for more diverse messages. This mechanical telegraph was never adapted for commercial or public use, which would have given it more diverse set of messages without as many words or phrases in common.

3.4 Semi-adaptive Dictionary

The static-dictionary approach requires one dictionary, defined in advance, that applies to all messages. If a new dictionary could be defined for each message, the compression could be greater because the particular sequences of symbols found in the message could be made into dictionary entries.

These techniques have several drawbacks, however. First, the new dictionary would have to be transmitted along with the encoded message, resulting in increased overhead. Second, the message would have to be analyzed to discover the best set of dictionary entries, and therefore the entire message would have to be available for analysis before any part of it could be encoded (that is, this technique has large **latency**). Third, the computer calculating the dictionary would need to have enough memory to store the entire message.

These disadvantages have limited the use of semi-adaptive dictionary compression schemes.

3.5 Dynamic Dictionary

What would be best for many applications would be an encoding scheme using a dictionary that is calculated on the fly, as the message is processed, does not need to accompany the message, and can be used before the end of the message has been processed. On first consideration, this might seem impossible. However, such a scheme is known and in wide use. It is the LZW compression technique, named after Abraham Lempel, Jacob Ziv, and Terry Welch. Lempel and Ziv actually had a series of techniques, sometimes

referred to as LZ77 and LZ78, but the modification by Welch in 1984 gave the technique all the desired characteristics.

Welch wanted to reduce the number of bits sent to a recording head in disk drives, partly to increase the effective capacity of the disks, and partly to improve the speed of transmission of data. His scheme is described here, for both the encoder and decoder. It has been widely used, in many contexts, to lead to reversible compression of data. When applied to text files on a typical computer, it can lead to encoded files that are half the size of the original. It is used in popular compression products such as Stuffit or Disk Doubler. When used on color images of drawings, with large areas that are the exact same color, it can lead to files that are smaller than half the size compared with file formats in which each pixel is stored. The commonly used GIF image format uses LZW compression. When used with photographs, with their gradual changes of color, the savings are much more modest.

Because this is a reversible compression technique, the original data can be reconstructed exactly, without approximation, by the decoder.

The LZW technique seems to have many advantages. However, it had one major disadvantage. It was not freely available – it was patented. The U.S. patent, No. 4,558,302, expired June 20, 2003, and patents in other countries in June, 2004, but before it did, it generated a controversy that is still an unpleasant memory for many because of the way it was handled.

3.5.1 The LZW Patent

Welch worked for Sperry Research Center at the time he developed his technique, and he published a paper[2] describing the technique. Its advantages were quickly recognized, and it was used in a variety of compression schemes, including the Graphics Interchange Format GIF developed in 1987 by CompuServe (a national Internet Service Provider) for the purpose of reducing the size of image files in their computers. Those who defined GIF did not realize that the LZW algorithm, on which GIF was based, was patented. The article by Welch did not warn that a patent was pending. The World Wide Web came into prominence during the early 1990s, and the first graphical browsers accepted GIF images. Consequently, Web site developers routinely used GIF images, thinking the technology was in the public domain, which had been CompuServe's intention.

By 1994, Unisys, the successor company to Sperry, realized the value of this patent, and decided to try to make money from it. They approached CompuServe, who didn't pay much attention at first, apparently not thinking that the threat was real. Finally, CompuServe took Unisys seriously, and the two companies together announced on December 24, 1994, that any developers writing software that creates or reads GIF images would have to license the technology from Unisys. Web site developers were not sure if their use of GIF images made them responsible for paying royalties, and they were not amused at the thought of paying for every GIF image on their sites. Soon Unisys saw that there was a public-relations disaster in the making, and they backed off on their demands. On January 27, 1995, they said they would not try to collect for use of existing images or for images produced by tools distributed before the end of 1994, but did insist on licensing graphics tools starting in 1995. Images produced by licensed tools would be allowed on the Web without additional payment.

In 1999, Unisys decided to collect from individual Web sites that might contain images from unlicensed tools, at the rate of $5000 per site, with no exception for nonprofits, and no smaller license fees for small, low-traffic sites. It is not known how many Web sites actually paid that amount; it is reported that in the first eight months only one did. The feeling among many Web site developers was one of frustration and anger. Although Unisys avoided a public-relations disaster in 1995, they had one on their hands in 1999. There were very real cultural differences between the free-wheeling Web community, often willing to share freely, and the business community, whose standard practices were designed to help make money.

A non-infringing public-domain image-compression standard called PNG was created to replace GIF, but the browser manufacturers were slow to adopt yet another image format. Also, everybody knew that the patents would expire soon. The controversy has now gone away except in the memory of a few who felt

[2] Welch, T.A. "A Technique for High Performance Data Compression," IEEE Computer, vol. 17, no. 6, pp. 8-19; 1984.

particularly strongly. It is still cited as justification for or against changes in the patent system, or even the concept of software patents in general.

As for PNG, it offers some technical advantages (particularly better transparency features) and, as of 2004, was well supported by almost all browsers, the most significant exception being Microsoft Internet Explorer for Windows.

3.5.2 How does LZW work?

The technique, for both encoding and decoding, is illustrated with a text example in Section 3.7.

3.6 Irreversible Techniques

This section is not yet available. Sorry.

It will include as examples floating-point numbers, JPEG image compression, and MP3 audio compression. The Discrete Cosine Transformation (DCT) used in JPEG compression is discussed in Section 3.8. A demonstration of MP3 compression is available at http://www.mtl.mit.edu/Courses/6.050/2007/notes/mp3.html.

3.7 Detail: LZW Compression

The LZW compression technique is described below and applied to two examples. Both encoders and decoders are defined. The LZW compression algorithm is "reversible," meaning that it does not lose any information—the decoder can reconstruct the original message exactly.

3.7.1 LZW Algorithm, Example 1

Consider the encoding and decoding of the text message

`itty bitty bit bin`

(this peculiar phrase was designed to have repeated strings so that the dictionary builds up rapidly).

The initial set of dictionary entries is 8-bit character code with values 0–255, with ASCII as the first 128 characters, including the ones in Table 3.1 which appear in the string above. Dictionary entry 256 is defined as "clear dictionary" or "start," and 257 as "end of transmission" or "stop." The encoded message is a sequence of numbers, the codes representing dictionary entries. Initially most dictionary entries consist of a single character, but new entries will be defined that stand for strings of two or more characters. The result is summarized in Table 3.2.

32	space	116	t
98	b	121	y
105	i	256	start
110	n	257	stop

Table 3.1: LZW Example 1 Starting Dictionary

Encoding algorithm: Define a place to keep new dictionary entries while they are being constructed and call it **new-entry**. Start with new-entry empty, and send the start code. Then append to the new-entry the characters, one by one, from the string being compressed. As soon as new-entry fails to match any existing dictionary entry, put new-entry into the dictionary, using the next available code, and send the code for the string without the last character (this entry is already in the dictionary). Then use the last character received as the first character of the next new-entry. When the input string ends, send the code for whatever is in new-entry followed by the stop code. That's all there is to it.

For the benefit of those who appreciate seeing algorithms written like a computer program, this encoding algorithm is shown in Figure 3.5. When this procedure is applied to the string in question, the first character

```
 0  # encoding algorithm
 1  clear dictionary
 2  send start code
 3  for each character {
 4    if new-entry appended with character is not in dictionary {
 5      send code for new-entry
 6      add new-entry appended with character as new dictionary entry
 7      set new-entry blank
 8    }
 9    append character to new-entry
10  }
11  send code for new-entry
12  send stop code
```

Figure 3.5: LZW encoding algorithm

Encoding				Transmission		Decoding		
Input		New dictionary entry		9-bit characters transmitted		New dictionary entry		Output
105	i	–	–	256	(start)	–	–	–
116	t	258	i t	105	i	–	–	i
116	t	259	t t	116	t	258	i t	t
121	y	260	t y	116	t	259	t t	t
32	space	261	y space	121	y	260	t y	y
98	b	262	space b	32	space	261	y space	space
105	i	263	b i	98	b	262	space b	b
116	t	–	–	–	–	–	–	–
116	t	264	i t t	258	i t	263	b i	i t
121	y	–	–	–	–	–	–	–
32	space	265	t y space	260	t y	264	i t t	t y
98	b		–	–	–	–	–	–
105	i	266	space-b i	262	space b	265	t y space	space b
116	t	–	–	–		–	–	–
32	space	267	i t space	258	i t	266	space b i	i t
98	b	–	–	–		–	–	–
105	i	–	–	–		–	–	–
110	n	268	space b i n	266	space b i	267	i t space	space b i
–	–	–	–	110	n	268	space b i n	n
–	–	–	–	257	(stop)	–	–	–

8-bit characters input

Table 3.2: LZW Example 1 Transmission Summary

is “i” and the string consisting of just that character is already in the dictionary. So the next character is appended to new-entry, and the result is “it” which is not in the dictionary. Therefore the string which was in the dictionary, “i,” is sent and the string “i t” is added to the dictionary, at the next available position, which is 258. The new-entry is reset to be just the last character, which was not sent, so it is “t”. The next character “t” is appended and the result is “tt” which is not in the dictionary. The process repeats until the end of the string is reached.

For a while at the beginning the additional dictionary entries are all two-character strings, and there is a string transmitted for every new character encountered. However, the first time one of those two-character strings is repeated, its code gets sent (using fewer bits than would be required for two characters sent separately) and a new three-character dictionary entry is defined. In this example it happens with the string “i t t” (this message was designed to make this happen earlier than would be expected with normal text). Later in this example, the code for a three-character string gets transmitted, and a four-character dictionary entry defined.

In this example the codes are sent to a receiver which is expected to decode the message and produce as output the original string. The receiver does not have access to the encoder’s dictionary and therefore must build up its own copy.

Decoding algorithm: If the start code is received, clear the dictionary and set new-entry empty. For the next received code, output the character represented by the code and also place it in new-entry. Then for subsequent codes received, append the first character of the string represented by the code to new-entry, insert the result in the dictionary, then output the string for the received code and also place it in new-entry to start the next dictionary entry. When the stop code is received, nothing needs to be done; new-entry can be abandoned.

This algorithm is shown in program format in Figure 3.6.

```
 0  # decoding algorithm
 1  for each received code until stop code {
 2    if code is start code {
 3      clear dictionary
 4      get next code
 5      get string from dictionary  # it will be a single character
 6    }
 7    else {
 8      get string from dictionary
 9      update last dictionary entry by appending first character of string
10    }
11    add string as new dictionary entry
12    output string
13  }
```

Figure 3.6: LZW decoding algorithm

Note that the coder and decoder each create the dictionary on the fly; the dictionary therefore does not have to be explicitly transmitted, and the coder deals with the text in a single pass.

Does this work, i.e., is the number of bits needed for transmission reduced? We sent 18 8-bit characters (144 bits) in 14 9-bit transmissions (126 bits), a savings of 12.5%, for this very short example. For typical text there is not much reduction for strings under 500 bytes. Larger text files are often compressed by a factor of 2, and drawings even more.

3.7.2 LZW Algorithm, Example 2

Encode and decode the text message

`itty bitty nitty grrritty bit bin`

(again, this peculiar phrase was designed to have repeated strings so that the dictionary forms rapidly; it also has a three-long sequence `rrr` which illustrates one aspect of this algorithm).

The initial set of dictionary entries include the characters in Table 3.3, which are found in the string, along with control characters for start and stop.

32	space	114	r
98	b	116	t
103	g	121	y
105	i	256	start
110	n	257	stop

Table 3.3: LZW Example 2 Starting Dictionary

The same algorithms used in Example 1 can be applied here. The result is shown in Table 3.4. Note that the dictionary builds up quite rapidly, and there is one instance of a four-character dictionary entry transmitted. Was this compression effective? Definitely. A total of 33 8-bit characters (264 bits) were sent in 22 9-bit transmissions (198 bits, even including the start and stop characters), for a saving of 25% in bits.

There is one place in this example where the decoder needs to do something unusual. Normally, on receipt of a transmitted codeword, the decoder can look up its string in the dictionary and then output it and use its first character to complete the partially formed last dictionary entry, and then start the next dictionary entry. Thus only one dictionary lookup is needed. However, the algorithm presented above uses

two lookups, one for the first character, and a later one for the entire string. Why not use just one lookup for greater efficiency?

There is a special case illustrated by the transmission of code 271 in this example, where the string corresponding to the received code is not complete. The first character can be found but then before the entire string is retrieved, the entry must be completed. This happens when a character or a string appears for the first time three times in a row, and is therefore rare. The algorithm above works correctly, at a cost of an extra lookup that is seldom needed and may slow the algorithm down. A faster algorithm with a single dictionary lookup works reliably only if it detects this situation and treats it as a special case.

Encoding: Input		Encoding: New dictionary entry		Transmission		Decoding: New dictionary entry		Decoding: Output
105	i	–	–	256	(start)	–	–	–
116	t	258	i t	105	i	–	–	i
116	t	259	t t	116	t	258	i t	t
121	y	260	t y	116	t	259	t t	t
32	space	261	y space	121	y	260	t y	y
98	b	262	space b	32	space	261	y space	space
105	i	263	b i	98	b	262	space b	b
116	t	–	–	–	–	–	–	–
116	t	264	i t t	258	i t	263	b i	i t
121	y	–	–	–	–	–	–	–
32	space	265	t y space	260	t y	264	i t t	t y
110	n	266	space n	32	space	265	t y space	space
105	i	267	n i	110	n	266	space n	n
116	t	–	–	–	–	–	–	–
116	t	–	–	–	–	–	–	–
121	y	268	i t t y	264	i t t	267	n i	i t t
32	space	–	–	–	–	–	–	–
103	g	269	y space g	261	y space	268	i t t y	y space
114	r	270	g r	103	g	269	y space g	g
114	r	271	r r	114	r	270	g r	r
114	r	–	–	–	–	–	–	–
105	i	272	r r i	271	r r	271	r r	r r
116	t	–	–	–	–	–	–	–
116	t	–	–	–	–	–	–	–
121	y	–	–	–	–	–	–	–
32	space	273	i t t y space	268	i t t y	272	r r i	i t t y
98	b	–	–	–	–	–	–	–
105	i	274	space b i	262	space b	273	i t t y space	space b
116	t	–	–	–	–	–	–	–
32	space	275	i t space	258	i t	274	space b i	it
98	b	–	–	–	–	–	–	–
105	i	–	–	–	–	–	–	–
110	n	276	space b i n	274	space b i	275	i t space	space b i
–	–	–	–	110	n	276	space b i n	n
–	–	–	–	257	(stop)	–	–	–

Table 3.4: LZW Example 2 Transmission Summary

3.8 Detail: 2-D Discrete Cosine Transformation

This section is based on notes written by Luis Pérez-Breva, February 3rd 2005, and notes from Joseph C. Huang, February 25, 2000.

The Discrete Cosine Transformation (DCT) is an integral part of the JPEG (Joint Photographic Experts Group) compression algorithm. DCT is used to convert the information in an array of picture elements (pixels) into a form in which the information that is most relevant for human perception can be identified and retained, and the information less relevant may be discarded.

DCT is one of many discrete linear transformations that might be considered for image compression. It has the advantage that fast algorithms (related to the FFT, Fast Fourier Transform) are available.

Several mathematical notations are possible to describe DCT. The most succinct is that using vectors and matrices. A vector is a one-dimensional array of numbers (or other things). It can be denoted as a single character or between large square brackets with the individual elements shown in a vertical column (a row representation, with the elements arranged horizontally, is also possible, but is usually regarded as the transpose of the vector). In these notes we will use bold-face letters ($\mathbf{V}$) for vectors. A matrix is a two-dimensional array of numbers (or other things) which again may be represented by a single character or in an array between large square brackets. In these notes we will use a font called "blackboard bold" ($\mathbb{M}$) for matrices. When it is necessary to indicate particular elements of a vector or matrix, a symbol with one or two subscripts is used. In the case of a vector, the single subscript is an integer in the range from 0 through $n-1$ where n is the number of elements in the vector. In the case of a matrix, the first subscript denotes the row and the second the column, each an integer in the range from 0 through $n-1$ where n is either the number of rows or the number of columns, which are the same only if the matrix is square.

3.8.1 Discrete Linear Transformations

In general, a discrete linear transformation takes a vector as input and returns another vector of the same size. The elements of the output vector are a linear combination of the elements of the input vector, and therefore this transformation can be carried out by matrix multiplication.

For example, consider the following matrix multiplication:

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 5 \\ 3 \end{bmatrix}. \tag{3.1}$$

If the vectors and matrices in this equation are named

$$\mathbb{C} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \mathbf{I} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}, \quad \mathbf{O} = \begin{bmatrix} 5 \\ 3 \end{bmatrix},$$

then Equation 3.1 becomes

$$\mathbf{O} = \mathbb{C}\mathbf{I}. \tag{3.2}$$

We can now think of $\mathbb{C}$ as a discrete linear transformation that transforms the input vector $\mathbf{I}$ into the output vector $\mathbf{O}$. Incidentally, this particular transformation $\mathbb{C}$ is one that transforms the input vector $\mathbf{I}$ into a vector that contains the *sum* (5) and the *difference* (3) of its components.[3]

The procedure is the same for a 3×3 matrix acting on a 3 element vector:

$$\begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} \longrightarrow \begin{bmatrix} o_1 & = \sum_j c_{1,j} i_j \\ o_2 & = \sum_j c_{2,j} i_j \\ o_3 & = \sum_j c_{3,j} i_j \end{bmatrix} \tag{3.3}$$

which again can be written in the succinct form

$$\mathbf{O} = \mathbb{C}\mathbf{I}. \tag{3.4}$$

[3] It happens that $\mathbb{C}$ is $\sqrt{2}$ times the 2×2 Discrete Cosine Transformation matrix defined in Section 3.8.2.

where now the vectors are of size 3 and the matrix is 3×3.

In general, for a transformation of this form, if the matrix $\mathbb{C}$ has an inverse $\mathbb{C}^{-1}$ then the vector **I** can be reconstructed from its transform by

$$\mathbf{I} = \mathbb{C}^{-1}\mathbf{O}. \tag{3.5}$$

Equations 3.3 and 3.1 illustrate the linear transformation when the input is a column vector. The procedure for a row-vector is similar, but the order of the vector and matrix is reversed, and the transformation matrix is transposed.[4] This change is consistent with viewing a row vector as the transpose of the column vector. For example:

$$\begin{bmatrix} 4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} 5 & 3 \end{bmatrix}. \tag{3.6}$$

Vectors are useful for dealing with objects that have a one-dimensional character, such as a sound waveform sampled a finite number of times. Images are inherently two-dimensional in nature, and it is natural to use matrices to represent the properties of a pixel. Video is inherently three-dimensional (two space and one time) and it is natural to use three-dimensional arrays of numbers to represent them. The succinct vector-matrix notation given here extends gracefully to two-dimensional systems, but not to higher dimensions (other mathematical notations have to be used).

Extending linear transformations to act on matrices, not just vectors, is not difficult. For example, consider a very small image of six pixels, three rows of two pixels each, or two columns of three pixels each. A number representing some property of each pixel (such as its brightness on a scale of 0 to 1) could form a 3×2 matrix:

$$\begin{bmatrix} i_{1,1} & i_{1,2} \\ i_{2,1} & i_{2,2} \\ i_{3,1} & i_{3,2} \end{bmatrix} \tag{3.7}$$

The most general linear transformation that leads to a 3×2 output matrix would require 36 coefficients. When the arrangement of the elements in a matrix reflects the underlying object being represented, a less general set of linear transformations, that operate on the rows and columns separately, using different matrices $\mathbb{C}$ and $\mathbb{D}$, may be useful:

$$\begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \begin{bmatrix} i_{1,1} & i_{1,2} \\ i_{2,1} & i_{2,2} \\ i_{3,1} & i_{3,2} \end{bmatrix} \begin{bmatrix} d_{1,1} & d_{1,2} \\ d_{2,1} & d_{2,2} \end{bmatrix} \longrightarrow \begin{bmatrix} o_{1,1} & o_{1,2} \\ o_{2,1} & o_{2,2} \\ o_{3,1} & o_{3,2} \end{bmatrix} \tag{3.8}$$

or, in matrix notation,

$$\mathbb{O} = \mathbb{C}\mathbb{I}\mathbb{D}, \tag{3.9}$$

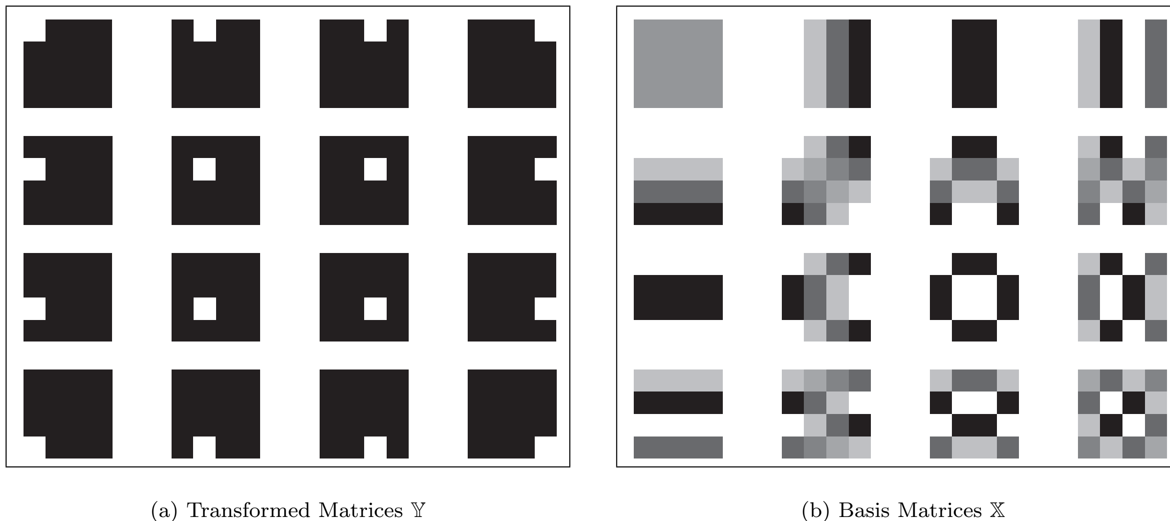
Note that the matrices at left $\mathbb{C}$ and right $\mathbb{D}$ in this case are generally of different size, and may or may not be of the same general character. (An important special case is when $\mathbb{I}$ is square, i.e., it contains the same number of rows and columns. In this case the output matrix $\mathbb{O}$ is also square, and $\mathbb{C}$ and $\mathbb{D}$ are the same size.)

3.8.2 Discrete Cosine Transformation

In the language of linear algebra, the formula

$$\mathbb{Y} = \mathbb{C}\mathbb{X}\mathbb{D} \tag{3.10}$$

[4] Transposing a matrix means flipping its elements about the main diagonal, so the i, j element of the transpose is the j, i element of the original matrix. Transposed matrices are denoted with a superscript T, as in $\mathbb{C}^T$. In general the transpose of a product of two matrices (or vectors) is the product of the two transposed matrices or vectors, in reverse order: $(\mathbb{A}\mathbb{B})^T = \mathbb{B}^T\mathbb{A}^T$.



(a) Transformed Matrices $\mathbb{Y}$ (b) Basis Matrices $\mathbb{X}$

Figure 3.7: (a) 4×4 pixel images representing the coefficients appearing in the matrix $\mathbb{Y}$ from equation 3.15. And, (b) corresponding Inverse Discrete Cosine Transformations, these ICDTs can be interpreted as a the base images that correspond to the coefficients of $\mathbb{Y}$.

represents a transformation of the matrix $\mathbb{X}$ into a matrix of coefficients $\mathbb{Y}$. Assuming that the transformation matrices $\mathbb{C}$ and $\mathbb{D}$ have inverses $\mathbb{C}^{-1}$ and $\mathbb{D}^{-1}$ respectively, the original matrix can be reconstructed from the coefficients by the reverse transformation:

$$\mathbb{X} = \mathbb{C}^{-1}\mathbb{Y}\mathbb{D}^{-1}. \tag{3.11}$$

This interpretation of $\mathbb{Y}$ as coefficients useful for the reconstruction of $\mathbb{X}$ is especially useful for the Discrete Cosine Transformation.

The Discrete Cosine Transformation is a Discrete Linear Transformation of the type discussed above

$$\mathbb{Y} = \mathbb{C}^T\mathbb{X}\mathbb{C}, \tag{3.12}$$

where the matrices are all of size $N \times N$ and the two transformation matrices are transposes of each other. The transformation is called the Cosine transformation because the matrix $\mathbb{C}$ is defined as

$$\{\mathbb{C}\}_{m,n} = k_n \cos\left[\frac{(2m+1)n\pi}{2N}\right] \text{where } k_n = \begin{cases} \sqrt{1/N} & \text{if } n = 0 \\ \sqrt{2/N} & \text{otherwise} \end{cases} \tag{3.13}$$

where $m, n = 0, 1, \ldots (N-1)$. This matrix $\mathbb{C}$ has an inverse which is equal to its transpose:

$$\mathbb{C}^{-1} = \mathbb{C}^T. \tag{3.14}$$

Using Equation 3.12 with $\mathbb{C}$ as defined in Equation 3.13, we can compute the DCT $\mathbb{Y}$ of any matrix $\mathbb{X}$, where the matrix $\mathbb{X}$ may represent the pixels of a given image. In the context of the DCT, the inverse procedure outlined in equation 3.11 is called the Inverse Discrete Cosine Transformation (IDCT):

$$\mathbb{X} = \mathbb{C}\mathbb{Y}\mathbb{C}^T. \tag{3.15}$$

With this equation, we can compute the set of base matrices of the DCT, that is: *the set of matrices to which each of the elements of* $\mathbb{Y}$ *corresponds via the DCT*. Let us construct the set of all possible images with a single non-zero pixel each. These images will represent the individual coefficients of the matrix $\mathbb{Y}$.

Figure 3.7(a) shows the set for 4×4 pixel images. Figure 3.7(b) shows the result of applying the IDCT to the images in Figure 3.7(a). The set of images in Figure 3.7(b) are called basis because the DCT of any of them will yield a matrix $\mathbb{Y}$ that has a single non-zero coefficient, and thus they represent the base images in which the DCT "decomposes" any input image.

Recalling our overview of Discrete Linear Transformations above, should we want to recover an image $\mathbb{X}$ from its DCT $\mathbb{Y}$ we would just take each element of $\mathbb{Y}$ and multiply it by the corresponding matrix from 3.7(b). Indeed, Figure 3.7(b) introduces a very remarkable property of the DCT basis: it encodes spatial frequency. Compression can be achieved by ignoring those spatial frequencies that have smaller DCT coefficients. Think about the image of a chessboard—it has a high spatial frequency component, and almost all of the low frequency components can be removed. Conversely, blurred images tend to have fewer higher spatial frequency components, and then high frequency components, lower right in the Figure 3.7(b), can be set to zero as an "acceptable approximation". This is the principle for irreversible compression behind JPEG.

Figure 3.8 shows MATLAB code to generate the basis images as shown above for the 4×4, 8×8, and 16×16 DCT.

```
% N is the size of the NxN image being DCTed.
% This code, will consider 4x4, 8x8, and 16x16 images
% and will construct the basis of DCT transforms
for N = [4 8 16];
    % Create The transformation Matrix C
    C = zeros(N,N);
    mrange=0:(N-1);           %m will indicate rows
    nrange=mrange;            %n will indicate columns
    k=ones(N,N)*sqrt(2/N);    %create normalization matrix
    k(:,1)=sqrt(1/N);         %note different normalization for first column
    C=k.*cos((2*mrange+1)'*nrange*pi/(2*N));    %construct the transform matrix
    %note that we are interested in the Inverse Discrete Cosine Transformation
    %so we must invert (i.e. transpose) C
    C=C';
    % Get Basis Matrices
    figure;colormap('gray');  %open the figure and set the color to grayscale
    for m = mrange
        for n = nrange
            %create a matrix that just has one pixel
            Y=zeros(N,N);
            Y(n+1,m+1)=1;

            X = C'*Y*C;       % X is the transformed matrix.
            subplot(N,N,m*N+n+1);imagesc(X); %plot X
            axis square;      %arrange scale of axis
            axis off;         %hide axis
        end
    end
end
```

Figure 3.8: Base matrix generator