Data Compression
The Complete Reference

Auxiliary Material

This page inentionally left blank

Contents

The propositions of mathematics are devoid of all factual content;
they convey no information whatever on any empirical subject matter.

Carl G. Hempel, *On the Nature of Mathematical Truth*

A
The ASCII Code

Table A.1 shows the 128 ASCII codes. Each code is shown in both octal (leftmost column and top row) and hexadecimal (rightmost column and either the top or bottom rows). Octal numbers are typeset in italics, preceded by a quote ('). Hex numbers are typeset in a fixed-width font, preceded by a double-quote ("). To find the code of a character, substitute the code from the top or bottom row for the `x`. The octal code of "A," for example, is found by substituting the *'1* from the top row for the *x* in the *'10x* from the left column. Thus, the code is octal 101 or binary 1000001. Similarly, the hexadecimal code of "A" is a combination of `"4x` and *'1*, i.e., it is `"41`.

◇ **Exercise A.1:** What is the hex code of "I"?

The first 32 codes (0–31) as well as the last one (DEL) are control characters. SP (code 32) stands for a blank space. The other codes are for the letters, digits, and punctuation marks.

A.1 ASCII Features

1. The first 32 codes, and also the last code, are the control characters. Those are characters used in input/output and computer communications, and have no corresponding graphics, i.e., they cannot be printed out. They are described in Table A.2.

2. The ASCII codes are arbitrary. The code of "A" is $\texttt{41}_{16}$, but there is no special reason for assigning that particular value, and almost any other value would have served as well. About the only rule for assigning codes is that the code of "B" should follow, numerically, the code of "A." Thus "B" has the code $\texttt{42}_{16}$, "C" has $\texttt{43}_{16}$, and so on. The same is true for the lower-case letters and for the ten digits.

There is also a simple relationship between the codes of the upper- and lower-case letters. The code of "a" is obtained from the code of "A" by setting the 7th bit to 1.

3. The parity bit in Table A.1 is always 0. The ASCII code does not specify the value of the parity bit and any value can be used. Different computers use the ASCII code with even parity, odd parity, no parity, or even a fixed parity of 1.

4. The code of the control character DEL is all ones (except the parity that is, as usual, unspecified). This is a tradition from the old days of computing (and also from telegraphy), when punched paper tape was an important medium for input/output. When punching information on a paper tape, whenever the user noticed an error, they would delete the bad character by pressing the DEL key on the keyboard. This worked by backspacing the tape and punching a frame of all 1's on top of the holes of the bad character. When reading the tape, the tape reader would simply skip any frame of all 1's.

	′0	*′1*	*′2*	*′3*	*′4*	*′5*	*′6*	*′7*	
′00x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	″0x
′01x	BS	HT	LF	VT	FF	CR	SO	SI	
′02x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	″1x
′03x	CAN	EM	SUB	ESC	FS	GS	RS	US	
′04x	SP	!	"	#	$	%	&	’	″2x
′05x	(	)	*	+	,	-	.	/	
′06x	0	1	2	3	4	5	6	7	″3x
′07x	8	9	:	;	<	=	>	?	
′10x	@	A	B	C	D	E	F	G	″4x
′11x	H	I	J	K	L	M	N	O	
′12x	P	Q	R	S	T	U	V	W	″5x
′13x	X	Y	Z	[	\	]	^	_	
′14x	‘	a	b	c	d	e	f	g	″6x
′15x	h	i	j	k	l	m	n	o	
′16x	p	q	r	s	t	u	v	w	″7x
′17x	x	y	z	{	|	}	~	DEL	
	″8	″9	″A	″B	″C	″D	″E	″F	

Table A.1: The ASCII Code.

NUL (Null): No character, Used for filling in space in an I/O device when there are no characters.

SOH (Start of heading): Indicates the start of a heading on an I/O device. The heading may include information pertaining to the entire record that follows it.

STX (Start of text): Indicates the start of the text block in serial I/O.

ETX (End of text): Indicates the end of a block in serial I/O. Matches a STX.

EOT (End of transmission): Indicates the end of the entire transmission in serial I/O.

ENQ (Enquiry): An enquiry signal typically sent from a computer to an I/O device before the start of an I/O transfer, to verify that the device is there and is ready to accept or to send data.

ACK (Acknowledge): An affirmative response to an ENQ.

BEL (Bell): Causes the I/O device to ring a bell or to sound a buzzer or an alarm in order to call the operator's attention.

BS (Backspace): A command to the I/O device to backspace one character. Not every I/O device can respond to BS. A keyboard is a simple example of an input device that cannot go back to the previous character. Once a new key is pressed, the keyboard loses the previous one.

HT (Horizontal tab): Sent to an output device to indicate a horizontal movement to the next tab stop.

LF (Line feed): An important control code. Indicates to the output device to move vertically, to the beginning of the next line.

VT (Vertical tab): Commands an output device to move vertically to the next vertical tab stop.

FF (Form feed): Commands the output device to move the output medium vertically to the start of the next page. some output devices, such as a tape or a plotter, do not have any pages and for them the FF character is meaningless.

CR (Carriage return): Commands an output device to move horizontally, to the start of the line.

SO (Shift out): Indicates that the character codes that follow (until an SI is sensed), are not in the standard character set.

SI (Shift in): Terminates a non-standard string of text.

DLE (Data link escape): Changes the meaning of the character immediately following it.

DC1–DC4 (Device controls): Special characters for sending commands to I/O devices. Their meaning is not predefined.

NAK (Negative acknowledge): A negative response to an enquiry.

SYN (Synchronous idle): Sent by a synchronous serial transmitter when there is no data to send.

ETB (End transmission block): Indicates the end of a block of data in serial transmission. Is used to divide the data into blocks.

CAN (Cancel): Tells the receiving device to cancel (disregard) the previously received block because of a transmission error.

EM (End of medium): Sent by an I/O device when it has sensed the end of its medium. The medium can be a tape, paper, card, or anything else used to record and store information.

SUB (Substitute): This character is substituted by the receiving device, under certain conditions, for a character that has been received incorrectly (had a bad parity bit).

ESC (Escape): Alters the meaning of the immediately following character. This is used to extend the character set. Thus ESC followed by an "X" may mean something special to a certain program.

FS (File separator):
GS (Group separator):
RS (Record separator):
US (Unit separator):

} The four separators on the left have no predefined meaning in ASCII, except that FS is the most general separator (separates large groups) and US, the least general.

SP (Space): This is the familiar blank or space between words. It is non-printing and is therefore considered a control character rather than a punctuation mark.

DEL (Delete): This is sent immediately after a bad character has been sent. DEL Indicates deleting the preceding character (see note 4 earlier).

Table A.2: The ASCII Control Characters.

ASCII stupid question, get a stupid ANSI.

Anonymous

B
Basics of Probability

The key concept discussed here is that of the *discrete random variable*. Such a variable X is defined by considering the results of a sequence of random events E_i. We associate a real number x_i with each event E_i, and define X as the set of all the different x_is (a random variable can also be continuous, but this case is not discussed here).

A simple example is the set of results of throwing a die. The results are numbers between 1 and 6, obtained with equal probability. A more interesting example is the result of throwing two dice. The result is a number between 2 and 12, but these are obtained with different probabilities. A result of 2 can be obtained only if each die falls on 1; thus, it has low probability. A result of 12, similarly, is also obtained with low probability. A result of 6, however, is obtained when the two dice fall on (1,5), (2,4), or (3,3), so it is much more common. A practical example is the pixels of an image. In an image with 8-bit pixels, e.g., the value of a pixel is between 0 and 255, but not all values occur with the same probability. Imagine two images, **A**, with a lot of red and **B**, with mostly blue. The pixel distributions of those images form different random variables, because a pixel with, say, value 112 may occur in **A** with a probability 0.01 and in **B**, with probability 0.02.

◇ **Exercise B.1:** Consider the event of throwing three coins, where a coin can fall on a head (H) or a tail (T). Let our random variable X be the number of coins that fall on heads. List all the values of X and their probabilities.

It is clear that we can write $X = (x_1, p_1, x_2, p_2 \ldots, x_n, p_n)$ where x_i are the different values that X takes and p_i is the probability of x_i. The average (or *mean*) of X is $\overline{X} = (1/n)\sum x_i$.

The expected value, or *expectation*, of the variable X is denoted $E(X)$ and is defined as $E(X) = \sum p_i x_i = \sum x_i P(X = x_i)$. This is the sum of the possible values x_i of X, each weighted by its probability p_i. The notation $P(X = x_i)$ should be read "the probability that X will take the value x_i." If all the probabilities p_i are

the same ($p_i = 1/n$), then $E(X) = \overline{X}$. In the case of three coins, the expected number of heads is $E(X) = \overline{X} = (3+2+2+1+2+1+1+0)/8 = 3/2$. We don't expect, of course, any individual event (throwing) to produce 1.5 heads, but this will be the number obtained on average.

◇ **Exercise B.2:** What is the expected value of the random variable X that takes just one value v?

Next we consider the two random variables $X = (0, 20)$ and $Y = (10, 10)$. In either case, the expected value of the variable is 10. However, variable Y always equals 10, whereas X never takes this value. This indicates that the expectation of a random variable does not give enough information about it, and we should consider how far the individual values x_i are from the expectation $E(X)$ (how far the individual values deviate from the expected value). We thus want to derive an expression for the *variance* of a random variable. Intuitively it seems that Y should have a zero variance, while the variance of X should be nonzero.

The first method that comes to mind is to calculate the sum of the differences $[x_i - E(X)]$. This definition of the variance is simple, but also counterintuitive, since it produces a zero in the case of X. Better results are obtained when the absolute differences $|x_i - E(X)|$ are used, instead of the differences.

To understand why even this definition is not satisfactory, consider the case of $n-1$ values x_i, perhaps the results of $n-1$ measurements of lengths, that are all in the range $[0, 10]$. The expectation is also in this range. We now make the next measurement and get the value $x_n = 1000$. It makes sense to require that this single value, which is so far from the other ones and from the expectation, should be given more weight in the calculation of the variance. This is why the squares of the differences, instead of the differences themselves, are used in the definition of the variance. If the expectation is 10, then a value $x_i = 5$ contributes $(5-10)^2 = 25$, but a value $x_i = 110$ makes the much bigger contribution of $(110-10)^2 = 10000$.

This is why the variance of the random variable X is defined as (see also page 370)

$$\mathrm{Var}(X) = \sum p_i [x_i - E(X)]^2.$$

If all the p_i are identical, then $p_i = 1/n$ and the variance becomes

$$\mathrm{Var}(X) = \frac{1}{n} \sum [x_i - E(X)]^2.$$

The standard deviation is another useful statistical measure. It is defined as the square root of the variance.

The variance is a natural unit for measuring how a variable varies, but the standard deviation is also useful because a random variable X has a dimension such as height or weight. The dimension of the variance is the square of the dimension of the variable (height squared or weight squared), so the variance and the random variable cannot be used together in calculations (for example, they cannot be added or subtracted). The standard deviation, however, has the dimension of the variable.

> The computer informed here that three spaces accounted for eighty-one percent of variance.
>
> — Michael Crichton, *The Terminal Man*

The standard deviation is a measure of the "radius" of the variable. If X has a Gaussian distribution (Section B.3.1), 68% of its values will be within one standard deviation of the mean, and 95% will be within two standard deviations.

◇ **Exercise B.3:** The definition above implies $\mathrm{Var}(X) = E[(X - E(X))^2]$. Show that $\mathrm{Var}(X) = E(X^2) - [E(X)]^2$.

Bivariate observations are those in which values of two random variables, X and Y, are taken. Such observations are referred to as "paired," and pairing can occur in a number of situations. The following are some examples:

1. When there are two different variables for each case (e.g., age and shoe size, height and weight, sex and IQ, country's infant mortality and average education).

2. When the same variable is measured for each case at two different times (e.g., reading level before training and reading level after, IQ at age 3 and IQ at age 6).

3. When the same or different variables are measured from related cases (e.g., father's and son's educational attainment, husband's height and wife's height; mother's anxiety and child's security).

4. When the same or different variables are measured in unrelated cases at the same time (for example, unemployment rate in city A and in city B in a given month).

The two variables can be independent or dependent. Imagine two persons living in different areas, whose phone numbers (minus the area codes) are the same. We intuitively feel that these persons' heights are independent. On the other hand, the incomes of two persons having similar phone numbers within the same area code may be dependent (although perhaps just to a small degree), because such persons may live close to each other, so there is a chance that they interact or even work together.

The question is how to measure the relation between two random variables. It is easy to define independence. Two random variables X and Y are independent if

$$P(X = a, Y = b) = P(X = a)P(Y = b), \quad \text{for any } a, b.$$

The *covariance* of two random variables is a measure of the linear relation between them. The covariance indicates only the direction of the linear relation, not its strength. It is defined as

$$COV_{xy} = \frac{\sum(x_i - \overline{X})(y_i - \overline{Y})}{n-1}.$$

The following simple Matlab code calculates the covariance matrix of a square matrix where each column is a variable:

```
function xy=covarmat(x)
[m,n]=size(x);
xc=x-repmat(sum(x)/m,m,1); % subtract average
xy=xc' * xc/(m-1);
```

A positive covariance indicates that the values of one variable increase as the values of the other increase. A negative covariance indicates that the values of one variable decrease as the values of the other increase. A zero covariance indicates that there is no linear relation between the two variables; they are *decorrelated*. The precise value of the covariance normally does not have any meaning because it depends on the units of measurement of X and Y and on their variances. To actually measure the amount of correlation between two variables we use the statistical measure of *correlation*. It is defined as

$$R = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{[n\sum x_i^2 - (\sum x_i)^2][n\sum y_i^2 - (\sum y_i)^2]}}. \tag{B.1}$$

It measures the linear relation between two paired variables. The values of R range from -1 (perfect negative relation), to 0 (no relation), to $+1$ (perfect positive relation).

Assume that we have two arrays x_i and y_i of numbers, perhaps measurements of the height and weight of a group of people. The correlation coefficient of the two arrays is a measure of how well the variation of one of them (say, y_i) can be predicted from the variation of the other (x_i). We can also consider x_i and y_i variables instead of arrays. If we look at x_i as the independent variable and at y_i as the dependent variable, then the correlation coefficient of the two variables is a measure of how well the variation of the dependent variable can be predicted from the variation of the independent variable.

If $y_i = ax_i$ for every i (for some real a), then the variation of y_i can easily and accurately be predicted from that of x_i and the correlation coefficient is 1 (or -1 if a is negative). If y follows x (i.e., if $x_{i+1} < x_i$ implies that $y_{i+1} < y_i$, and $x_{i+1} > x_i$ implies that $y_{i+1} > y_i$), then the correlation coefficient is a positive number. In the opposite case (where $x_{i+1} < x_i$ implies $y_{i+1} > y_i$, and $x_{i+1} > x_i$ implies $y_{i+1} < y_i$), the correlation coefficient is a negative number.

> Variance is what any two statisticians are at.
>
> — Unknown

B.1 Joint and Union of Events

We first discuss the probabilities of *independent events*. Those are events that do not affect each other's probabilities. An example is the throw of a die or of a coin. Each outcome of a throw is independent of the preceding (and also of the following) throws. Even when two dice are thrown together, the two outcomes are independent. We denote the probability of outcome x by $P(x)$. The *joint probability* $P(x \cdot y)$ of

the independent events x and y is the probability that one event will result in x and another event, in y. This probability is simply the product of the individual probabilities. For example, if the probability of having snow today (event A) is 25% and that of having snow tomorrow (event B) is 40%, then the probability of having snow today *and* tomorrow, $P(A \cdot B)$, is the product $0.25 \times 0.4 = 0.1$, i.e., 10% (it is, of course, smaller than each of the individual probabilities). The probability of the *complement* $\bar{x}$ of an event x equals one minus the probability of the event, $P(\bar{x}) = 1 - P(x)$. Thus, the probability of *not* having snow today is $1 - 0.25 = 0.75$.

Another important concept is the probability of the *union* of events. For example, what is the probability of having snow today *or* tomorrow? To understand how this is computed, consider the opposite case. The opposite of it snowing today or snowing tomorrow is the case where it does not snow today (event $\bar{A}$) and it does not snow tomorrow (event $\bar{B}$). The probability of this event is, of course, the joint probability $P(\bar{A} \cdot \bar{B})$, i.e., $P(\bar{A})P(\bar{B})$, so we conclude that this joint event is the complement of the union that we are looking for. Thus, we end up with $P(A + B) = 1 - P(\bar{A} \cdot \bar{B}) = 1 - P(\bar{A})P(\bar{B})$. In the snowing example the result is $P(A + B) = 1 - P(\bar{A})P(\bar{B}) = 1 - (1 - 0.25)(1 - 0.4) = 1 - 0.45 = 0.55$. This result is greater than 25% and 40%, but it is still a probability, i.e., in the range $[0, 1]$.

◇ **Exercise B.4:** (For gamblers.) Calculate the probability of winning in the following two games. In game A, the player rolls two dice up to 24 times and wins if he rolls a double-six. In game B the player rolls a single die and wins if he gets a six in four rolls. These winning probabilities were first calculated by Blaise Pascal, one of the founders of modern probability theory, in 1654.

B.2 Conditional Probability

Not all events are independent. When dealing with dependent events, we have to calculate *conditional probabilities*. We usually ask the question: What is the probability of event A given that event B has occurred? This is the conditional probability of A (more precisely, the probability of A conditioned on B) and it is denoted by $P(A|B)$. The field of conditional probability is sometimes called *Bayesian statistics*, since it was first developed by the Reverend Thomas Bayes [1702–1761], who came up with the basic formula

$$P(A|B) = \frac{P(A \cdot B)}{P(B)}. \tag{B.2}$$

Example: Three cards are given. One is red on both sides, another is black on both sides, and the third is red on one side and black on the other side. You pick up a card without looking at it, place it on the table, then look at it. If you see a red side, what is the probability of the other side also being red? This probability is conditional and is denoted by $P(Rup|Rdown)$. Equation (B.2) becomes

$$P(Rup|Rdown) = \frac{P(Rup \cdot Rdown)}{P(Rdown)}.$$

The probability $P(Rup \cdot Rdown)$ is 1/3, since only one of the three cards has two red sides. The probability $P(Rdown)$ can be calculated as follows: We have three

cards. The probability of any card being picked up is 1/3. The probability that the first card will be picked up *and* will have red down is $(1/3)\times 1$, since both its sides are red. The probability that the second card will be picked up *and* will have red down is $(1/3)\times 0$, since none of its sides is red. The probability that the third card will be picked up *and* will have red down is $(1/3)(1/2)$, since only one of its sides is red. Thus, the total probability of picking a card and having red down is $(1/3)(1+0+1/2) = 1/2$.

The same result can be obtained by considering the events points in some *event space*. The first card will have red down no matter what, so it contributes a point (point 1, with probability 1/3) to the event space. The second card will never have red down, no matter what. It contributes another point, point 2, also with probability 1/3, to the event space. The third card will have red down if picked up in a certain way, and red on top if picked up another way. It therefore contributes two points (points 3 and 4, with probability 1/6 each) to the event space. We are interested in points 1 and 3, whose probabilities are 1/3 and 1/6, that add up to 1/2. Equation (B.2) therefore yields the conditional probability

$$P(Rup|Rdown) = \frac{P(Rup \cdot Rdown)}{P(Rdown)} = \frac{1/3}{1/2} = \frac{2}{3};$$

a nonintuitive result.

◇ **Exercise B.5:** Three companies A, B, and C are competing for a NASA contract to develop a new space vehicle. Experts agree that the chances of A, B, and C to win the contract are 2/5, 2/5, and 1/5, respectively. At a certain point company B withdraws from the competition. What are the winning chances of A and C now?

Example: A family has two children, where a child can be a boy or a girl. If nothing is known a priori about the children, then there are the four possibilities (BB, GG, BG, GB), and the probability of having two boys is 1/4. What is the conditional probability of the family having two boys, if it is known that one of the children is a boy? Intuitively this knowledge eliminates the GG case and reduces the possibilities to (BB, BG, GB), of which BB is what we are after. The conditional probability is therefore

$$P(\text{two boys}|\text{one boy}) = \frac{P(BB)}{P(BB)+P(BG)+P(GB)} = \frac{\frac{1}{4}}{\frac{1}{4}+\frac{1}{4}+\frac{1}{4}} = \frac{1}{3}.$$

Using Bayesian statistics we have to calculate $P(A|B)$ where A is the event of having two boys and B is the event of having a boy. The quantity $P(A \cdot B)$ is the probability of having two boys *and* having a boy. Well, if a family has two boys, then it has a boy, so looking at the three possibilities (BB, GG, BG), we find that $P(A \cdot B) = P(A) = 1/3$. The quantity $P(B)$ is the probability of having a boy. It equals 2/3 because this event makes up two of the three possibilities. Equation (B.2) yields

$$P(A|B) = \frac{P(A \cdot B)}{P(B)} = \frac{1/3}{2/3} = \frac{1}{2}.$$

An important practical case is where an event A has certain alternatives A_i. In such a case, Equation (B.2) can be generalized to

$$P(A_i|B) = \frac{P(A_i \cdot B)}{P(A_1 \cdot B) + P(A_2 \cdot B) + \cdots + P(A_n \cdot B)} = \frac{P(A_i)P(B|A_i)}{P(A_1)P(B|A_1) + P(A_2)P(B|A_2) + \cdots + P(A_n)P(B|A_n)}.$$

This is the well-known Bayes' theorem, published posthumously by Thomas Bayes in 1763. It requires knowledge of the probabilities $P(A_i)$ of the alternatives and of the conditional probabilities $P(B|A_i)$ of B relative to the alternatives. This theorem is used by the QM coder to construct the probability estimation tables used by JPEG, JBIG, and JBIG2 (see, for example, Table 2.68).

Example: A student has to select one science course among mathematics, physics, chemistry, and biology. Based on his personal knowledge of the student, his advisor figures that the probabilities that the student will select one of the four classes are 0.4, 0.3, 0.2, and 0.1, respectively (these probabilities add up to 1). The advisor does not know what course was actually selected by the student, but at the end of the semester, the advisor hears that the student got an A in the selected course. Based on the difficulties of these courses, the advisor estimates the probabilities of the student's getting an A in mathematics, physics, chemistry, and biology to be 0.1, 0.2, 0.3, and 0.9, respectively (these don't have to add up to 1). Using Bayes' theorem, the advisor can now revise the original probabilities that the student will select one of the four courses. The probability that the student has actually selected, for example, the mathematics course is

$$P(\text{selected math}|\text{got an A}) = \frac{0.4 \cdot 0.1}{0.4 \cdot 0.1 + 0.3 \cdot 0.2 + 0.2 \cdot 0.3 + 0.1 \cdot 0.9} = \frac{4}{25} = 0.16.$$

◇ **Exercise B.6:** Based on the fact that the student got an A, calculate the new probabilities that the student has actually selected physics, chemistry, and biology.

Example: This is the well-known *three prisoners* problem. Tom, Dick, and Harry are in prison. The next morning one of them is going to be executed and the other two will be freed. They don't know who the unlucky one is, but their guard knows. Close to midnight, Tom asks the guard, "I don't know who is going to be executed, but I do know that either Dick or Harry is going to be freed. Please tell me which one because, after all, it does not make any difference in my probability of being executed." The guard (an amateur statistician who never heard of Bayes) answers, "You now estimate the probability of your being executed to be 1/3. If I told you that, say, Dick is going to be freed, you would know that only you and Harry remain candidates for execution, so the probability of your being executed would go up to 1/2 and you won't sleep well tonight." The guard's answer is simple but wrong, since it does not consider conditional probability. The guard could have given Tom a name, such as Dick, but this knowledge would not have changed Tom's conditional probability of being executed. Here is the analysis.

Without any prior knowledge, each prisoner has probability 1/3 of being executed and probability 2/3 of being freed. We denote by A the event that Tom will be executed. Clearly $P(A) = 1/3$. We want to calculate the conditional probability that Tom will be executed, given that he was told by the guard that Dick would be freed. Event B is, therefore, the guard's telling Tom that Dick is going to be released. The subtle point is that event B is not necessarily the release of Dick; it's the guard's telling Tom that Dick is to be freed. The difference is easy to see by considering all the possible events points in an event space. There are four such points, as follows:

Point 1: Harry is to be executed. In this case the guard tells Tom that Dick is to be freed. Since each prisoner has probability 1/3 of being executed, the probability of this point is 1/3.

Point 2: Dick is to be executed. In this case the guard tells Tom that Harry is to be freed. The probability of this point is, again, 1/3.

Points 3 and 4: Tom is to be executed (with probability 1/3). In this case the guard tells Tom that either Harry or Dick are going to go free. These are points 3 and 4, respectively, in the event space, each with a probability of 1/6.

The joint probability $P(A \cdot B)$ that Tom will be executed *and* the guard says that Dick will be freed is, therefore, the probability of point 4, or 1/6. The probability of event B (the guard will say that Dick is to be released) is the sum of points 1 and 4, or $1/3+1/6$. The conditional probability that Tom will be executed, if the guard says that Dick is going to be freed is, therefore,

$$P(A|B) = \frac{P(A \cdot B)}{P(B)} = \frac{1/6}{1/3 + 1/6} = 1/3,$$

the same probability of Tom's being executed without the guard's saying anything.

(The author is indebted to J. Robert Henderson for pointing out this problem and its solution.)

The conclusion is: Stay out of trouble, but if you are in it (trouble, i.e.) place your trust in old Reverend Bayes.

B.3 Probability Distributions

Imagine an input stream where symbols have equal probabilities of occurrence. An example may be an image consisting of N 8-bit pixels where each of the 256 possible colors appears exactly $N/256$ times. We say that the colors in this image have a *flat distribution* of values. The graph whose x-axis shows the 256 colors and whose y-axis shows the number of times each color appears will be a horizontal line. Now imagine a similar image, but with mostly green tones. There are many pixels with different shades of green and few with any other colors. The same graph will be high for x values that represent shades of green and low elsewhere. The distribution of colors in this image is not flat.

> The theory of symbiogenesis assumes that the most probable explanation for improbably complex structures (living or otherwise) lies in the association of less complicated parts. Sentences are easier to construct by combining words than by combining letters. Sentences then combine into paragraphs, paragraphs combine into chapters, and, eventually, chapters combine to form a book—highly improbable, but vastly more probable than the chance of arriving at a book by searching the space of possible combinations at the level of letters or words.
>
> — George B. Dyson, *Darwin Among The Machines*.

B.3.1 Gaussian Distribution

The Gaussian (also known as the Normal) distribution is an important statistical tool used in many branches of science. It provides a good model for continuous distributions that occur in many everyday situations. Examples are the following:

1. The distribution of peoples' heights. Most people are of medium height. Few are tall or short. Even fewer are very tall or very short. Practically no one is a giant or a dwarf. Imagine a sample of people whose height is known. If the sample is large enough and is not biased, the graph describing the number of people of height h as a function of h will look very similar to Figure B.1.

2. The speed of gas molecules. The molecules of a gas are in constant motion. They move randomly, collide with each other and with objects around them, and change their velocities all the time. However, most molecules in a given volume of gas move at about the same speed, and only a few move much faster or much slower than this speed. This speed is related to the temperature of the gas. The higher this average speed, the hotter the gas feels to us. (This example is asymmetric, since the minimum speed is zero, but the maximum speed can be very high.)

3. Château Chambord in the Loire valley of France has a magnificent staircase, designed by Leonardo da Vinci in the form of a double ramp spiral. Worn out by the innumerable footsteps of generations of tourists, the marble tread of this staircase now looks like an inverted normal distribution curve. It is worn mostly in the middle, were most people tend to step, and the wear tapers off to either side from the center. This staircase, and others like it, are physical embodiments of the abstract mathematical concept of probability distribution.

4. Prime numbers are familiar to most people. They are attractive and important to mathematicians because any positive integer can be expressed as a product of prime numbers (its *prime factors*) in one way only. The prime numbers are thus the building blocks from which all other integers can be constructed. It turns out that the number of distinct prime factors is distributed normally. Few integers have just one or two distinct prime factors, few integers have many distinct prime factors, while most integers have a small number of distinct prime factors. This is known as the Erdős-Kac theorem.

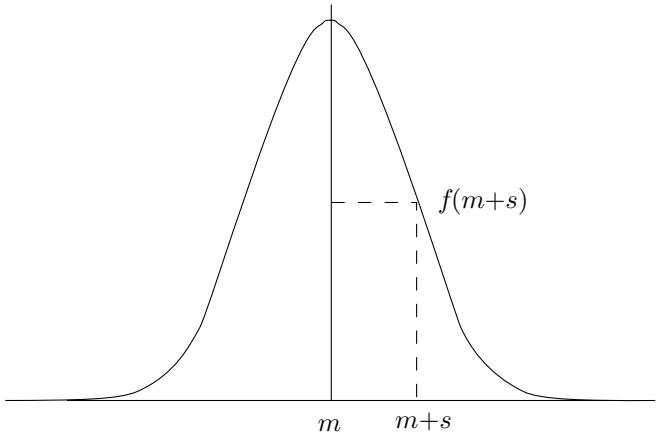


Figure B.1: Gaussian (Normal) Distribution.

> The general features of the bell curve make intuitive sense, but why it should have precisely the shape it does, which I can confidently predict with the help of a few lines of grade school arithmetic, and not some other shape—a little wider in the hip, say, or more pointed, like a witch's hat—that remains a mystery.
>
> — Hans Christian von Baeyer, *Maxwell's Demon*, 1998.

The Gaussian distribution with mean m and standard deviation s is defined as

$$f(x) = \frac{1}{s\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right\}.$$

This function has a maximum for $x = m$ (i.e., at the mean), where its value is $f(m) = 1/(s\sqrt{2\pi})$. It is also symmetric about $x = m$, since it depends on x according to $(x-m)^2$. It has the general "bell" shape of Figure B.1. At $x = m+s$ and $x = m-s$ its value is

$$f(m \pm s) = \frac{1}{s\sqrt{2\pi}} e^{-\frac{1}{2}} \approx \frac{0.6065}{s\sqrt{2\pi}},$$

which means that at one standard deviation from the mean it has dropped to about 60% of its maximum. At two standard deviations it drops to about 0.1353 of its maximum value.

The total area under the normal curve is one unit. The area one standard deviation from the mean equals ≈ 0.682. At two standard deviations it equals ≈ 0.9545 and at three standard deviations it is ≈ 0.9973.

(The Gaussian distribution and the concept of standard deviation were discovered and discussed by Abraham de Moivre in 1733, when he was 18.)

The precise shape of the curve depends on s. As s increases, the curve gets wider. For small values of s the curve approaches a narrow spike of height $1/(s\sqrt{2\pi})$ placed at $x = m$.

When we talk about random numbers we usually mean numbers that are uniformly distributed over a certain interval $[a, b]$. If we divide interval $[a, b]$ into equal-size subintervals, any of them would contain the same amount of random numbers. It is possible to draw random numbers that have different distributions, such as Gaussian. When we compute many random numbers that are normally (i.e., Gaussian) distributed with mean m and standard deviation s, then count the amount y of these numbers in a small subinterval $[x, x + \epsilon]$, plot (x, y) as a point, and repeat for many subintervals, we get the normal distribution with mean m and standard deviation s.

> Because of the very nature of the tables, it did not seem necessary to proofread every page of the final manuscript in order to catch random errors.
>
> — *A Million Random Digits With 100,000 Normal Deviates*, RAND Corp., 1955.

Here are two ways to compute random numbers that are normally distributed with mean 0 and standard deviation 1.

1. Draw n uniformly-distributed random numbers R_i in the range $[-a, +a]$ for any real a and calculate their average $(1/n)\sum R_i$. The average is the first of the normally-distributed random numbers N_i. Repeat this process to get N_2, N_3, and so on. The larger n, the closer to normal will be the distribution of these numbers. The reason that the N_i are normally distributed is that it is rare for the average of R_i to be $-a$ or $+a$ or close to these values, but it is common for it to be around 0. This is an aspect of the *law of large numbers* that says: if R_i are random numbers of any distribution, then the averages $(1/n)\sum R_i$ are normally distributed.

2. Method 1 above is simple but slow, since n should be large. The *Polar method* (see [Knuth 81] Vol. 2, Sec. 3.4.1) is more efficient. Let U_1 and U_2 be two uniformly-distributed random numbers in the range $[0, 1]$. We calculate two normally-distributed random numbers X_1, X_2 by the following two simple steps:

Step 1. Compute $V_1 := 2U_1 - 1$, $V_2 := 2U_2 - 1$, and $S := V_1^2 + V_2^2$.

Step 2. If $S \geq 1$ go to step 1; else compute $X_1 := V_1\sqrt{\frac{-2\ln S}{S}}$, $X_2 := V_2\sqrt{\frac{-2\ln S}{S}}$.

Once a sequence N_i is obtained of normally-distributed random numbers with mean 0 and standard deviation 1, it is easy to convert them to normally-distributed random numbers with mean m and standard deviation s. Just transform each N_i to $m + N_i \times s$.

Assuming that a function `Rnd()`, which returns uniformly-distributed random numbers in the range $[0, 1]$, is given. Gaussian random numbers with zero mean and a standard deviation of one can be obtained by the following:

```
x:=0.0;
for i:=1 to 12 do x:=x+Rnd();
Gauss:=x-6.0;
```

B.3.2 Laplace Distribution

The Laplace probability distribution is similar to the normal (Gaussian) distribution, but is narrower and sharply peaked. The general Laplace distribution with variance V and mean m is given by

$$L(V,x) = \frac{1}{\sqrt{2V}} \exp\left(-\sqrt{\frac{2}{V}}|x-m|\right).$$

Table B.2 shows some values for the Laplace distributions with $m = 0$ and $V = 3, 4, 5$, and 1,000.

	x					
V	0	2	4	6	8	10
3:	0.408248	0.0797489	0.015578	0.00304316	0.00059446	0.000116125
4:	0.353553	0.0859547	0.020897	0.00508042	0.00123513	0.000300282
5:	0.316228	0.0892598	0.025194	0.00711162	0.00200736	0.000566605
1,000:	0.022360	0.0204475	0.018698	0.0170982	0.0156353	0.0142976

Table B.2: Some Values of the Laplace Distribution with $V = 3, 4, 5$, and 1,000.

The factor $1/\sqrt{2V}$ is included in the definition of the Laplace distribution in order to scale the area under the curve of the distribution to 1.

◇ **Exercise B.7:** What is the indefinite integral of the Laplace distribution?

The Laplace distribution is used by the MLP image compression method (Section 4.19).

B.3.3 Discrete Distributions

Imagine n independent events being performed such that in each event a certain result (that we call a "success") occurs with the same probability p. A simple example is a coin throw, where a success is defined as the coin falling on head. The probability is, of course, 0.5. The number of successes in the n events is a random variable X that has a *binomial distribution*. The probability of X taking a value x is

$$P(X = x) = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x}, \quad \text{for } x = 0, 1, \ldots, n,$$

and the binomial distribution function is

$$P(X \le x) = \sum_{i=0}^{n} \frac{n!}{i!(n-i)!} p^i (1-p)^{n-i}.$$

There is a story about two friends, who were classmates in high school, talking about their jobs. One of them became a statistician and was working on population trends. He showed a reprint to his former classmate. The reprint started, as usual, with the Gaussian distribution, and the statistician explained to his former classmate the meaning of the symbols for the actual population, for the average population, and so on. His classmate was a bit incredulous and was not quite sure whether the statistician was pulling his leg.

"How can you know that?" was his query. "And what is this symbol here?"

"Oh," said the statistician, "this is π."

"What is that?"

"The ratio of the circumference of the circle to its diameter."

"Well now you are pushing your joke too far," said the classmate, "surely the population has nothing to do with the circumference of the circle."

— Eugene P. Wigner, 1960

The mean of X is np and its variance is $np(1-p)$.

If the number of occurrences of an event x per unit (unit of time, length, area, volume, or whatever) is, on average, a real positive number λ, then the actual number of such occurrences is a discrete random variable X with a Poisson distribution. The probability that X will take a value x is

$$P(X = x) = \frac{\lambda^x e^{-\lambda}}{x!}, \quad \text{for } x = 0, 1, \ldots, \infty,$$

and the distribution function is

$$P(X \leq x) = \sum_{i=0}^{x} \frac{\lambda^i e^{-\lambda}}{i!}.$$

The mean and the variance of the Poisson distribution both equal λ.

The theory of probability is at bottom nothing but common sense reduced to calculus.

Laplace, Pierre Simon de [1749–1827]

Socrates took poisson

Unknown

Siméon Denis Poisson [1781–1840]

Originally intended by his parents to study medicine, Poisson instead shifted to mathematics. He published his first book, on finite differences, at age 18.

Poisson taught at the École Polytechnique from 1802. In 1808 he became an astronomer at the Bureau des Longitudes. In 1809 he was appointed chair of pure mathematics in the newly established Faculté des Sciences.

His most important works are on definite integrals, Fourier series, probability theory, and mechanics.

In 1837 he published an important work on probability where he introduced the Poisson distribution. This distribution describes the probability that a random event will occur in a time or space interval under the conditions that the probability of the event occurring is very small, but the number of trials is very large, so that the event actually occurs a few times.

His important text *Traité de mécanique* was published in 1811 and again in 1833. This book was the standard work on mechanics for many years.

The mathematician Guglielmo Libri said of him: His only passion has been science: he lived and is dead for it.

One of Poisson's best known quotations is: Life is good for only two things, discovering mathematics and teaching mathematics.

C
Curves That Fill Space

A space-filling curve completely fills up part of space by passing through every point in that part. It does that by changing direction repeatedly. We will only discuss curves that fill up part of the two-dimensional plane, but the concept of a space-filling curve exists for any number of dimensions.

◇ **Exercise C.1:** Show an example of a space-filling curve in one dimension.

Several such curves are known and all are defined recursively. A typical definition starts with a simple curve C_0, shows how to use it to construct another, more complex curve C_1, and defines the final, space-filling curve as the limit of the sequence of curves $C_0, C_1, \ldots$.

C.1 The Hilbert Curve

(This discussion is based on the approach of [Wirth 76].) Perhaps the most familiar of these curves is the Hilbert curve, discovered by the great mathematician David Hilbert in 1891. The Hilbert curve [Hilbert 91] is the limit of a sequence $H_0, H_1, H_2 \ldots$ of curves, some of which are shown in Figure C.1. They are defined by the following:

0. H_0 is a single point.

1. H_1 consists of four copies of (the point) H_0, connected with three straight segments of length h at right angles to each other. Four orientations of this curve, labeled 1, 2, 3, and 4, are shown in Figure C.1a.

2. The next curve, H_2, in the sequence is constructed by connecting four copies of different orientations of H_1 with three straight segments of length $h/2$ (shown in bold in Figure C.1b). Again there are four possible orientations of H_2, and the one shown is #2. It is constructed of orientations 1223 of H_1, connected by segments

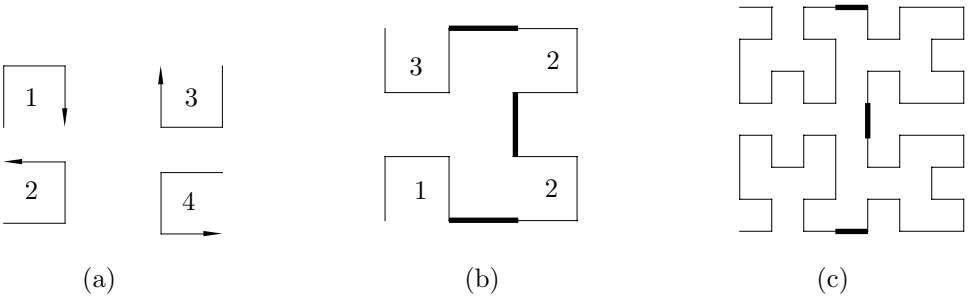


Figure C.1: Hilbert Curves of Orders 1, 2, and 3.

that go to the right, up, and to the left. The construction of the four orientations of H_2 is summarized in Table C.2.

Drawing this curve is thus done recursively. A procedure to draw orientation #4 of H_i is shown in Figure C.3. It makes four recursive calls to draw the four curves of order $i-1$, and draws straight segments between the calls, to connect them (the names A, B, C, and D are used instead of 1, 2, 3, and 4).

Figure C.4 is a complete Pascal program for H_n, compiled by the Metrowerks$^{\text{TM}}$ Macintosh Pascal compiler. Notice how the main program determines the initial values of the starting point (x, y) of the curve, and the segment size h. Variable $h0$ defines the size, in pixels, of the square containing the curve, and should be a power of 2.

Curve H_3 is shown in Figure C.1c. The particular curve shown is orientation 1223 of H_2.

Figures C.5, C.6 and C.7 show the Hilbert curves of orders 4, 5 and 6. It is easy to see how fast these curves become extremely complex.

C.2 The Sierpiński Curve

Another well-known space-filling curve is the Sierpiński curve. Figure C.8 shows curves S_1 and S_2, and Sierpiński has proved [Sierpiński 12] that the limit of the sequence $S_1, S_2, \ldots$ is a curve that passes through every point of the unit square $[0, 1] \times [0, 1]$.

To construct this curve, we need to figure out how S_2 is constructed out of four copies of S_1. The first thing that comes to mind is to follow the construction method used for the Hilbert curve, i.e., to take four copies of S_1, eliminate one edge in each, and connect them. This, unfortunately, does not work, since the Sierpiński curve is very different from the Hilbert curve. It is closed, and it has one orientation only. A better approach is to start with four parts that constitute four orientations of one open curve, and connect them with straight segments. The segments are shown dashed in Figure C.8. Notice how Figure C.8a is constructed of four orientations of a basic, three-part curve connected by four short, dashed segments. Figure C.8b is similarly constructed of four orientations of a complex, 15-part curve, connected by the same short, dashed segments. If we denote the four basic curves A, B, C, and D, then the basic construction rule of the Sierpiński curve is S: A↘B↙C↖D↗,

1: 2 ↑ 1 → 1 ↓ 4
2: 1 → 2 ↑ 2 ← 3
3: 4 ↓ 3 ← 3 ↑ 2
4: 3 ← 4 ↓ 4 → 1

Table C.2: The Four Orientations of H_2.

```
PROCEDURE D(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  A(i-1); x:=x-h; MoveTo(x,y);
  D(i-1); y:=y-h; MoveTo(x,y);
  D(i-1); x:=x+h; MoveTo(x,y);
  C(i-1);
  END;
END (*A*);
```

Figure C.3: A Recursive Procedure.

```
PROGRAM Hilbert; (* A Hilbert curve *)
USES ScreenIO, Graphics;

CONST LB = 5; Width = 630; Height = 430;
(* LB=left bottom corner of window *)
n=6; (* n is the order of the curve*)
h0=8; (* h0 should be a power of 2 *)

VAR h,x,y,x0,y0: INTEGER;

PROCEDURE B (i: INTEGER); FORWARD;
PROCEDURE C (i: INTEGER); FORWARD;
PROCEDURE D (i: INTEGER); FORWARD;

PROCEDURE A(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  D(i-1); x:=x-h; MoveTo(x,y);
  A(i-1); y:=y-h; MoveTo(x,y);
  A(i-1); x:=x+h; MoveTo(x,y);
  B(i-1);
  END;
END (*A*);

PROCEDURE B(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  C(i-1); y:=y+h; MoveTo(x,y);
  B(i-1); x:=x+h; MoveTo(x,y);
  B(i-1); y:=y-h; MoveTo(x,y);
  A(i-1);
  END;
END (*B*);

PROCEDURE C(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  B(i-1); x:=x+h; MoveTo(x,y);
  C(i-1); y:=y+h; MoveTo(x,y);
  C(i-1); x:=x-h; MoveTo(x,y);
  D(i-1);
  END;
END (*C*);

PROCEDURE D(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  A(i-1); y:=y-h; MoveTo(x,y);
  D(i-1); x:=x-h; MoveTo(x,y);
  D(i-1); y:=y+h; MoveTo(x,y);
  C(i-1);
  END;
END (*D*);

BEGIN (* Main *)
  OpenGraphicWindow
   (LB,LB,Width,Height,'Hilbert curve');
  SetMode(paint);
  h:=h0; x0:=h DIV 2; y0:=x0; h:=h DIV 2;
  x0:=x0+(h DIV 2); y0:=y0+(h DIV 2);
  x:=x0+400; y:=y0+350; SetPen(x,y);
  A(n);

  ScBOL;
  ScWriteStr
   ('Hit a key & close window to quit');
  ScFreeze;
END.
```

Figure C.4: Pascal Program for a Hilbert Curve of Order i.

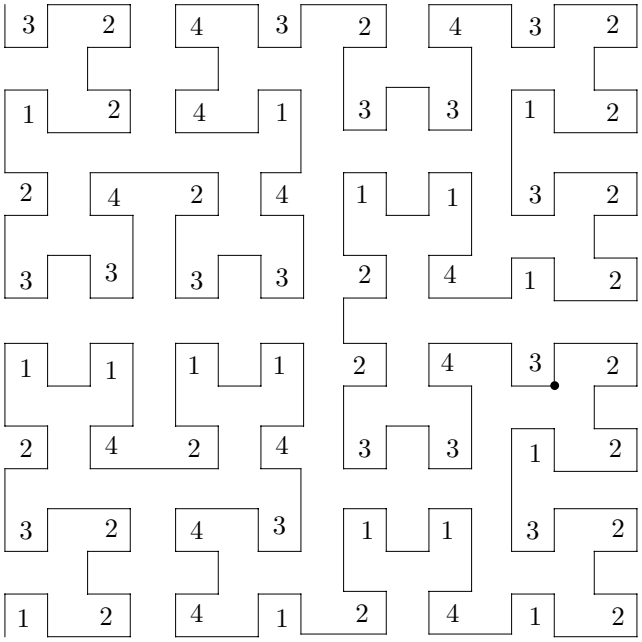


Figure C.5: Hilbert Curve of Order 4.

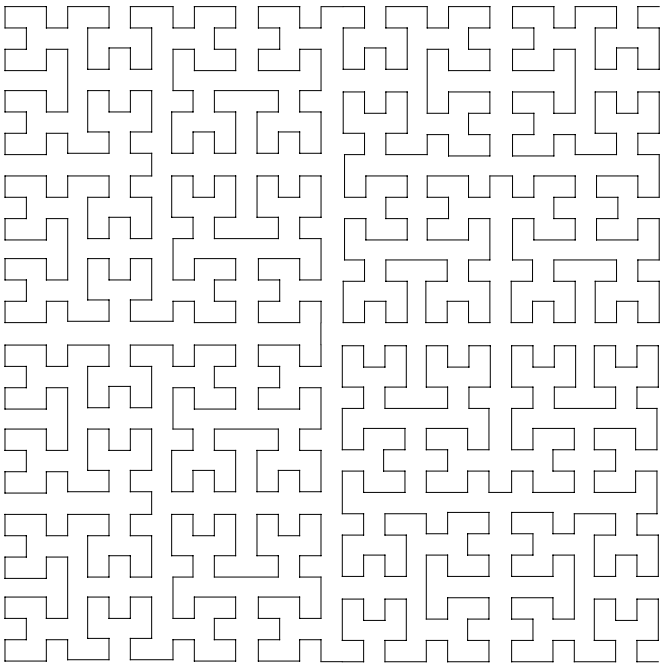


Figure C.6: Hilbert Curve of Order 5.

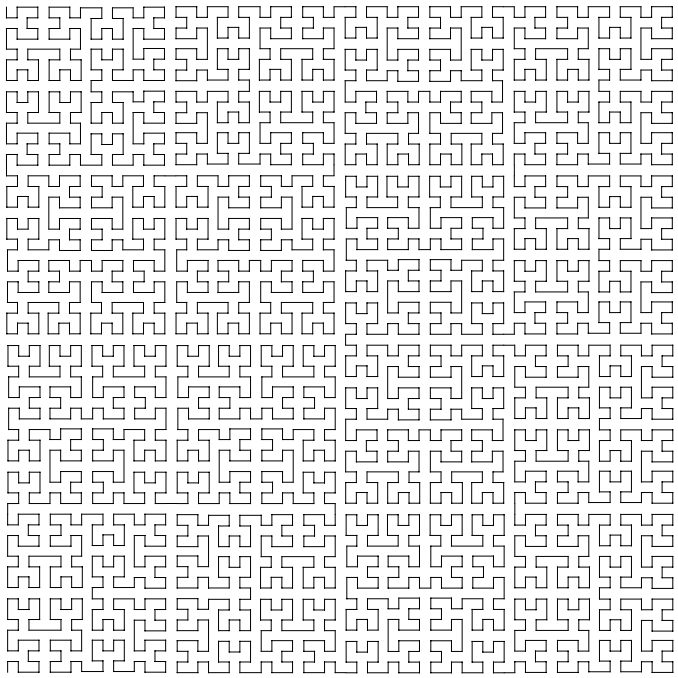
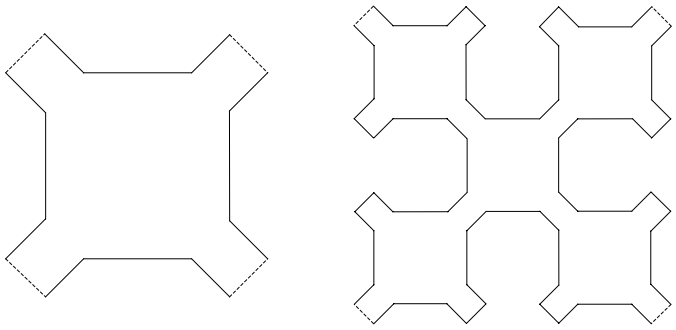


Figure C.7: Hilbert Curve of Order 6.



(a) (b)

Figure C.8: Sierpiński Curves of Orders 1 and 2.

and the recursion rules are:

$$\begin{array}{ll} \text{A:} & \text{A}\searrow\text{B}\rightarrow\rightarrow\text{D}\nearrow\text{A} \\ \text{B:} & \text{B}\swarrow\text{C}\downarrow\downarrow\text{A}\searrow\text{B} \\ \text{C:} & \text{C}\nwarrow\text{D}\leftarrow\leftarrow\text{B}\swarrow\text{C} \\ \text{D:} & \text{D}\nearrow\text{A}\uparrow\uparrow\text{C}\nwarrow\text{D} \end{array} \qquad \text{(C.1)}$$

Figure C.9 shows the five Sierpiński curves of orders 1 through 5 superimposed on each other. They were drawn by the Pascal program of Figure C.10.

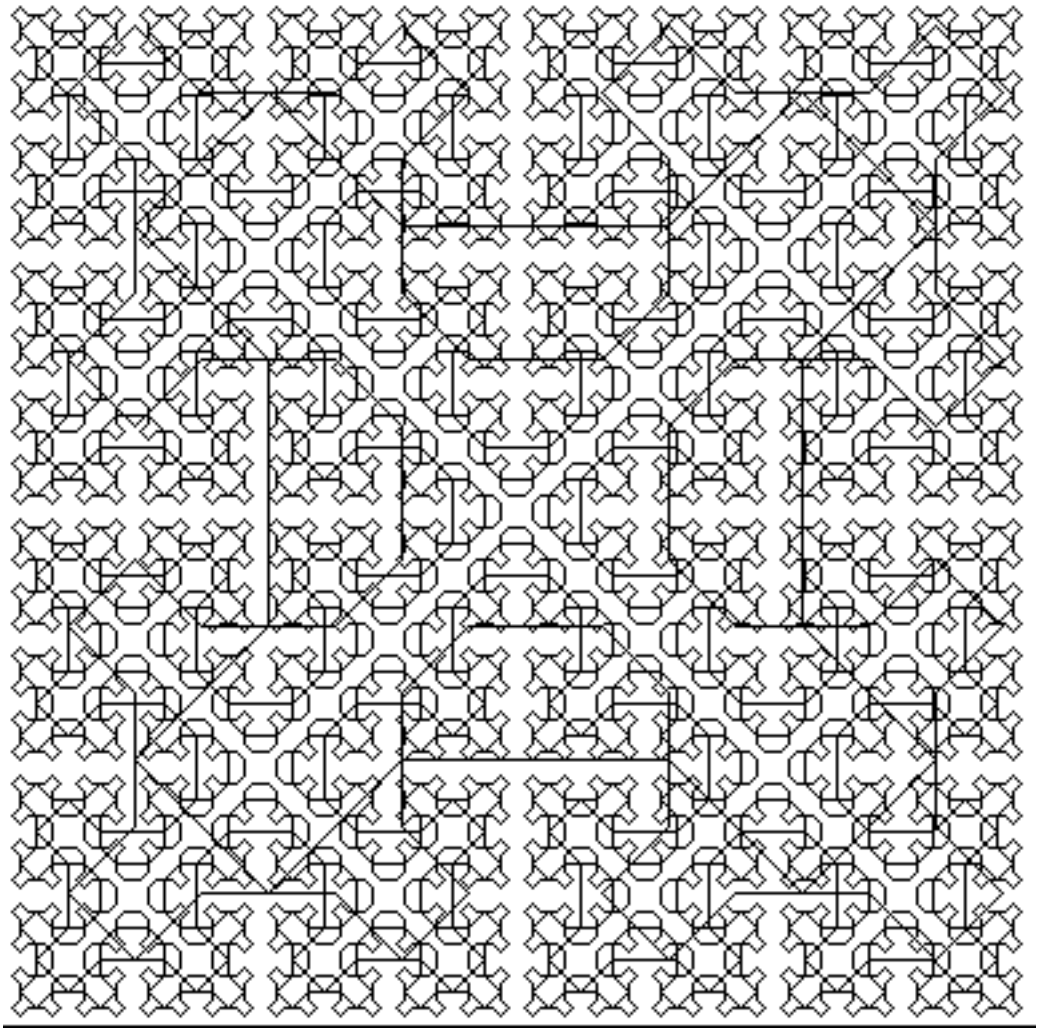


Figure C.9: Sierpiński Curves of Orders 1–5.

```
program Sierpinski;

USES ScreenIO, QuickDraw;

PROCEDURE SierpinskiP(size,number:word);
(* Adapted from Wirth, 2nd ed, p.115 *)

VAR
H, x, y:   INTEGER;

PROCEDURE B (level: word); FORWARD;
PROCEDURE C (level: word); FORWARD;
PROCEDURE D (level: word); FORWARD;

PROCEDURE A (level: word);
BEGIN
IF level > 0 THEN
BEGIN
A(level-1); Line(H,-H);
IF ScTellInput()>0 THEN BEGIN END;
B(level-1); Line(2*H,0);
IF ScTellInput()>0 THEN BEGIN END;
D(level-1); Line(H,H);
IF ScTellInput()>0 THEN BEGIN END;
A(level-1)
END
END {A};

PROCEDURE B (level: word);
BEGIN
IF level > 0 THEN
BEGIN
B(level-1); Line(-H,-H);
IF ScTellInput()>0 THEN BEGIN END;
C(level-1); Line(0,-2*H);
IF ScTellInput()>0 THEN BEGIN END;
A(level-1); Line(H,-H);
IF ScTellInput()>0 THEN BEGIN END;
B(level-1)
END
END {B};

PROCEDURE C (level: word);
BEGIN
IF level > 0 THEN
BEGIN
C(level-1); Line(-H,H);
IF ScTellInput()>0 THEN BEGIN END;
D(level-1); Line(-2*H,0);
IF ScTellInput()>0 THEN BEGIN END;
B(level-1); Line(-H,-H);
IF ScTellInput()>0 THEN BEGIN END;
C(level-1)
END
END {C};

PROCEDURE D (level: word);
BEGIN
IF level > 0 THEN
BEGIN
D(level-1); Line(H,H);
IF ScTellInput()>0 THEN BEGIN END;
A(level-1); Line(0,2*H);
IF ScTellInput()>0 THEN BEGIN END;
C(level-1); Line(-H,H);
IF ScTellInput()>0 THEN BEGIN END;
D(level-1)
END
END {D};

VAR
level: WORD;
BEXIT : BOOLEAN;

BEGIN (* SierpinskiP *)

level := 0;
H := size DIV 4;
x := 2 * H;
y := 3 * H;
    BEXIT := FALSE;
REPEAT
level := level + 1;; x := x - H;;
H := H DIV 2; y := y + H;;
MoveTo (x, y);
A (level); Line (H, -H);
IF ScTellInput()>0 THEN BEXIT:=TRUE;
B (level); Line (-H, -H);
IF ScTellInput()>0 THEN BEXIT:=TRUE;
C (level); Line (-H, H);
IF ScTellInput()>0 THEN BEXIT:=TRUE;
D (level); Line (H, H);
IF ScTellInput()>0 THEN BEXIT:=TRUE;
IF level = number THEN BEXIT:=TRUE;
 UNTIL BEXIT;
END {SierpinskiP};

VAR
ch:  CHAR;
prop:  termProp;

BEGIN
ScOpenWindow( 10, 10, 400, 400 );
ScGetProp (prop);
prop. showCurs := FALSE;
ScSetProp(prop); (*Hide alpha cursor*)
ScClear;
SierpinskiP(400,1);ScBeep (1);ScFreeze;
SierpinskiP(400,2);ScBeep (1);ScFreeze;
SierpinskiP(400,3);ScBeep (1);ScFreeze;
SierpinskiP(400,4);ScBeep (1);ScFreeze;
SierpinskiP(400,5);ScBeep (1);ScFreeze;
ScClose;
END {Sierpinski}.
```

Figure C.10: A Pascal Program for Sierpiński Curves of Orders 1–5.

◇ **Exercise C.2:** Figure I.6 shows three iterations of the Peano space-filling curve, developed in 1890. Use the techniques developed earlier for the Hilbert and Sierpiński curves, to describe how the Peano curve is constructed. (Hint: The curves shown are P_1, P_2, and P_3. The first curve, P_0 in this sequence is not shown.)

C.3 Traversing the Hilbert Curve

Space-filling curves are used in image compression (Section 4.29), which is why it is important to develop methods for a fast traversal of such a curve. Two approaches, both table-driven, are illustrated here for traversing the Hilbert curve.

The first approach [Cole 86] is based on the observation that the Hilbert curve H_i is constructed of four copies of its predecessor H_{i-1} placed at different orientations. A look at Figures C.1, C.5, C.6, and C.7 should convince the reader that H_i consists of 2^{2i} nodes connected with straight segments. The node numbers thus go from 0 to $2^{2i}-1$ and require $2i$ bits each. In order to traverse the curve we need a function that will compute the coordinates (x, y) of a node i from the node number i. The (x, y) coordinates of a node in H_i are i-bit numbers.

A look at Figure C.5 shows how successive nodes are initially located at the bottom left quadrant, and then move to the bottom right quadrant, the top right quadrant, and finally the top left one. This figure shows orientation #2 of the curve, so we can say that this orientation of H_i traverses quadrants 0, 1, 2, and 3, where quadrants are numbered $\binom{3\,2}{0\,1}$. It is now clear that the two leftmost bits of a node number determine its quadrant. Similarly, the next pair of bits in the node number determine its subquadrant within the quadrant, but here we run into the added complication that each subquadrant is placed at a different orientation in its quadrant. This approach thus uses Table C.11 to determine the coordinates of a node from its number.

Bit pair	x	y	Next table	Bit pair	x	y	Next table	Bit pair	x	y	Next table	Bit pair	x	y	Next table
00	0	0	2	00	0	0	1	00	1	1	4	00	1	1	3
01	1	0	1	01	0	1	2	01	0	1	3	01	1	0	4
10	1	1	1	10	1	1	2	10	0	0	3	10	0	0	4
11	0	1	4	11	1	0	3	11	1	0	2	11	0	1	1
	(1)				(2)				(3)				(4)		

Table C.11: Coordinates of nodes in H_i.

As an example, we compute the xy coordinates of node 109 (the 110th node) of orientation #2 of H_4. The H_4 curve has $2^{2\cdot 4} = 256$ nodes, so node numbers are 8 bits each, and $109 = 01101101_2$. We start with Table C.11(1). The 2 leftmost bits of the node number are 01, and table (1) tells us that the x coordinate start with 1, the y coordinate, with 0, and we should continue with table (1). The next pair of bits is 10, and table (1) tells us that the next bit of x is 1, the next bit of y is 1, and we should stay with table (1). The third pair of bits is 11, so table (1) tells us that the next bit of x is 0, the next bit of y is 1, and we should move to

table (4). The last pair of bits is 01, and table (4) tells us to append 1 and 0 to the coordinates of x and y, respectively. The coordinates are thus $x = 1101 = 13$, $y = 0110 = 6$, as can be verified directly from Figure C.5 (the small circle).

It is also possible to transform a pair of coordinates (x, y), each in the range $[0, 2^i - 1]$, to a node number in H_i by means of Table C.12.

xy pair	Int. pair	Next table	xy pair	Int. pair	Next table	xy pair	Int. pair	Next table	xy pair	Int. pair	Next table
00	00	2	00	00	1	00	10	3	00	10	4
01	11	4	01	01	2	01	01	3	01	11	1
10	01	1	10	11	3	10	11	2	10	01	4
11	10	1	11	10	2	11	00	4	11	00	3
	(1)			(2)			(3)			(4)	

Table C.12: Node Numbers in H_i.

◇ **Exercise C.3:** Use Table C.12 to compute the node number of the H_4 node whose coordinates are $(13, 6)$.

The second approach to Hilbert curve traversal uses Table C.2. Orientation #2 of the H_2 curve shown in Figure C.1(b) is traversed in order 1223. The same orientation of the H_3 curve of Figure C.1(c) is traversed in 2114 1223 1223 4332, but Table C.2 tells us that 2114 is the traversal order for orientation #1 of H_2, 1223 is the traversal for orientation #2 of H_2, and 4332 is for orientation #3. The traversal of orientation #2 of H_3 is thus also based on the sequence 1223. Similarly, orientation #2 of H_4 is traversed (Figure C.5) in the order

$$\begin{array}{l} 1223\ 2114\ 2114\ 3441 \quad 2114\ 1223\ 1223\ 4332 \\ 2114\ 1223\ 1223\ 4332 \quad 3441\ 4332\ 4332\ 1223, \end{array}$$

which is reduced to 2114 1223 1223 4332, which in turn is reduced to the same sequence 1223.

The idea is therefore to create the traversal order for orientation #2 of H_i by starting with the sequence 1223 and recursively expanding it $i - 1$ times, using Table C.2.

◇ **Exercise C.4:** (Easy.) Show how to apply this method to traversing orientation #1 of H_i.

A MATLAB function `hilbert.m` to compute the traversal of the curve is available at [Matlab 99]. It has been written by Daniel Leo Lau (`lau@ece.udel.edu`). The call `hilbert(4)` produces the 4×4 matrix

$$\begin{array}{cccc} 5 & 6 & 9 & 10 \\ 4 & 7 & 8 & 11 \\ 3 & 2 & 13 & 12 \\ 0 & 1 & 14 & 15 \end{array}.$$

C.4 Traversing the Peano Curve

The Peano curves P_0, P_1, and P_2 of Figure Ans.62 have 1, 3^2, and 3^4 nodes, respectively. In general, P_n has 3^{2n} nodes, numbered $0, 1, 2, \ldots, 3^{2n}-1$. This suggests that the Peano curve [Peano 90] is somehow based on the number 3, in contrast to the Hilbert curve, which is based on 2. The coordinates of the nodes vary from $(0,0)$ to $(n-1, n-1)$. It turns out that there is a correspondence between the node numbers and their coordinates [Cole 85], which uses base-3 reflected Gray codes (Section 4.2.1)

A reflected Gray code [Gray 53] is a permutation of the i-digit integers such that consecutive integers differ by one digit only. Here is one way to derive these codes for binary numbers. Start with $i = 1$. There are only two 1-bit digits, namely, 0 and 1, and they differ by 1 bit only. To get the RGC for $i = 2$ proceed as follows:

1. Copy the sequence $(0, 1)$.

2. Append (on the left or on the right) a 0 bit to the original sequence and a bit of 1 to the copy. The result is $(00, 01)$, $(10, 11)$.

3. Reflect (reverse) the second sequence. The result is $(11, 10)$.

4. Concatenate the two sequences to get $(00, 01, 11, 10)$.

It is easy to see that consecutive numbers differ by one bit only.

◇ **Exercise C.5:** Follow the rules above to get the binary RGC for $i = 3$.

Notice that the first and last numbers in an RGC also differ by one bit. RGCs can be created for any number system using the following notation and rules: Let $a = a_1 a_2 \cdots a_m$ be a non-negative, base-n integer (i.e., $0 \le a_i < n$). Define the quantity $p_j = (\sum_{i=1}^{j} a_i) \bmod 2$, and denote the base-$n$ RGC of a by $a' = b_1 b_2 \cdots b_m$. The digits b_i of a' can be computed by

$$b_1 = a_1; \quad b_i = \begin{cases} a_i & \text{if } p_{i-1} = 0; \\ n-1-a_i & \text{if } p_{i-1} = 1. \end{cases} \text{ Odd } n \qquad i = 2, 3, \ldots, m.$$

$$\begin{cases} a_i & \text{if } a_{i-1} \text{ is even;} \\ n-1-a_i & \text{if } a_{i-1} \text{ is odd.} \end{cases} \text{ Even } n$$

[Note that $(a')' = a$ for both even and odd n.] For example, the RGC of the sequence of base-3 numbers (trits) 000, 001, 002, 010, 011, 012, 020, 021, 022, 100, 101... is 000, 001, 002, 012, 011, 010, 020, 021, 022, 122, 121... .

The connection between Peano curves and RGCs is: Let a be a node in the peano curve P_m. Write a as a ternary (base-3) number with $2m$ trits $a = a_1 a_2 \cdots a_{2m}$. Let $a' = b_1 b_2 \cdots b_{2m}$ be the RGC equivalent of a. Compute the two numbers $x' = b_2 b_4 b_6 \cdots b_{2m}$ and $y' = b_1 b_3 b_5 \cdots b_{2m-1}$. The number x' is the RGC of a number x, and similarly for y'. The two numbers (x, y) are the coordinates of node a in P_m.

Bibliography

Backus, J. W. (1959) "The Syntax and Semantics of the Proposed International Algebraic Language," in *Proceedings of the International Conference on Information processing*, pp. 125–132, UNESCO.

Chomsky, N. (1956) "Three Models for the Description of Language," *IRE Transactions on Information Theory* **2**(3):113–124.

Cole, A. J. (1985) "A Note on Peano Polygons and Gray Codes," *International Journal of Computer Mathematics* **18**:3–13.

Cole, A. J. (1986) "Direct Transformations Between Sets of Integers and Hilbert Polygons," *International Journal of Computer Mathematics* **20**:115–122.

Gray, Frank (1953) "Pulse Code Communication," United States Patent 2,632,058, March 17.

Hilbert, D. (1891) "Ueber Stetige Abbildung Einer Linie auf ein Flachenstuck," *Math. Annalen* **38**:459–460.

Lindenmayer, A. (1968) "Mathematical Models for Cellular Interaction in Development," *Journal of Theoretical Biology* **18**:280–315.

Matlab (1999) is URL `http://www.mathworks.com/`.

Naur, P. et al. (1960) "Report on the Algorithmic Language ALGOL 60," *Communications of the ACM* **3**(5):299–314, revised in *Communications of the ACM* **6**(1):1–17.

Peano, G. (1890) "Sur Une Courbe Qui Remplit Toute Une Aire Plaine," *Math. Annalen* **36**:157–160.

Sierpiński, W. (1912) "Sur Une Nouvelle Courbe Qui Remplit Toute Une Aire Plaine," *Bull. Acad. Sci. Cracovie* Serie A:462–478.

Sagan, Hans (1994) *Space-Filling Curves*, New York, Springer Verlag.

Wirth, N. (1976) *Algorithms + Data Structures = Programs*, Englewood Cliffs, NJ, Prentice-Hall, 2nd Ed.

All letters come and go—L is here to stay.
Grzegorz Rozenberg

D
Data Structures

A computer program is a set of instructions (or statements) that specify operations on data items. Sometimes, data items are independent and each is stored separately in memory. In such a case, each item becomes a *variable* in the program and is given a name. A set of data items may, however, be related, and such items should be stored together in memory as a *data structure*. Examples are:

1. A data base with information about people. Each element of the data base may consist of a name (first, middle, and last), an identifying number, an address, and other data such as age, salary, or title. The name may be stored in an array. The identification number becomes an integer variable, the address is stored in another array, and so on. These arrays and variables are then grouped to become one node in a linked list.

2. A matrix of numbers, such as the quantization matrices used in certain image compression methods. Such numbers can be stored in a two-dimensional array.

Arrays and lists are examples of *data structures*. Two things should be described in a data structure, the way data items are related, and the operations that the program should be able to perform on the structure and on the data items. Examples of operations are inserting/deleting an item, replacing an item, searching for an item, and increasing/decreasing the size of the structure. Once the programmer knows how the data items are related and what operations are needed, the data structure can be designed. Data structures are sometimes simple, such as an array, a stack, or a list, but they can get very complex, since one structure may combine lists, stacks, hash tables, and arrays in a complex way.

The main data structure described in this chapter is the hash table, but we start with a short survey of some basic data structures.

D.1 Arrays

The array is a common, useful, and important data structure. Intuitively, we can think of an array as a set of consecutive memory locations grouped under one name, where each individual location is accessed by its *index*. While this is a practical description of arrays, it is not a good definition, since it defines this data structure by means of its actual representation in memory. A more formal definition is

An array is a set of pairs (index, value).

This defines an array as a mapping from the set of indexes to the set of values.

In principle, the index can be any data type, but in practice it is normally an integer. In the old FORTRAN programming language the first array element has index 1. In the C language the first index is 0, and in Pascal the programmer can specify the first array index. Most programming languages require the size of all arrays to be static, i.e., the size of an array cannot be changed at run time.

A program may use many data items, and storing them in an array is convenient, since the programmer has to memorize just one name, the name of the array, for all these items. However, each item stored in an array has an index, and the programmer either has to memorize the indexes, use an application where it is not necessary to memorize each index, or search the array for any particular item.

An important feature of arrays is that they can have more than one dimension. A two-dimensional array $A[m, n]$ is a matrix. Similarly, a three-dimensional array $A[l, m, n]$ is a three-dimensional matrix. Alternatively, we can think of it as a one-dimensional array of l elements, each a matrix of dimensions $m \times n$.

Even though a two-dimensional array can be considered a matrix, it is physically stored in memory as a set of consecutive locations. The array can be stored in memory row by row or column by column. Figure D.1 shows that in the former case, an array item $A[i, j]$ in an $m \times n$ array is stored in location $(i-1)n+j$ from the start of the array (assuming that row and column indexes start at 1).

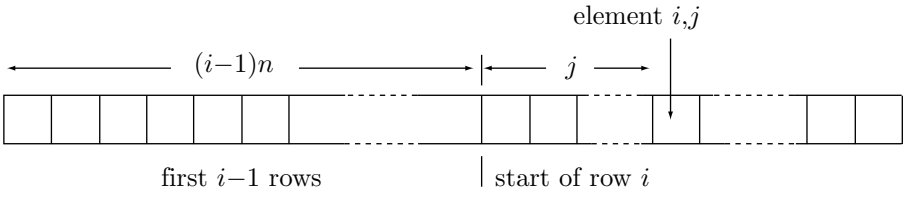


Figure D.1: Position of Array Element $A[i, j]$.

◇ **Exercise D.1:** Assume that an array A of dimensions $m \times n$ is stored in memory column by column and that row and column indexes start at 0. What is the distance, in memory, of array element $A[i, j]$ from the first array element $A[0, 0]$?

A simple example of the use of an array is storing the $n+1$ coefficients of a polynomial of degree n in an array of size $n+2$ or longer (see Equation (I.18) for the definition of a polynomial). In this case the programmer implicitly knows what's in each array location.

◇ **Exercise D.2:** Why should the array size be $n + 2$ and not $n + 1$?

D.2 Stacks and Queues

The stack is also a common data structure. Intuitively we think of a stack as a container open at one end. The only operations on a stack are insertion and deletion (commonly referred to as *push* and *pop*, respectively), and they are done from the open end. Figure D.2a shows how three data items are pushed into an empty stack and how the only item that can be popped out is the last that was pushed in. This is why a stack is sometimes referred to as a LIFO (last-in first-out) structure. The stack is normally implemented as an array, but because of the way it is used, the program needs to keep track of only one item, the top one (i.e., the last one in), at any given time. The program therefore maintains a pointer called *top of stack* that points at that data item.

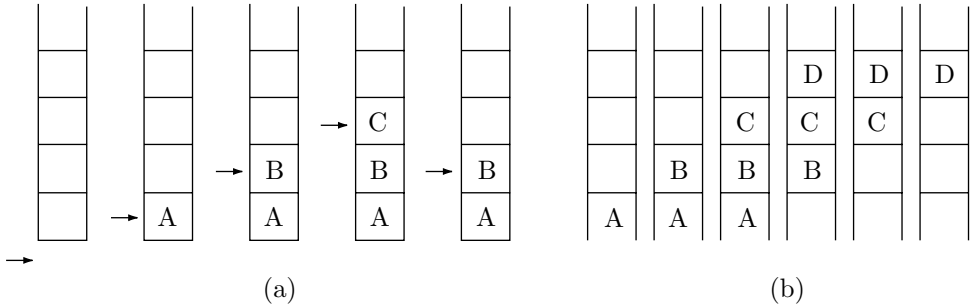


Figure D.2: Stack and Queue.

A *queue* is a data structure where the only item that can be removed is the oldest one. A queue is therefore based on the FIFO (first-in first-out) principle. Figure D.2b shows a queue during four insertions and three deletions. It is obvious that these operations move the data from the start to the end of the queue, which is why the *circular queue* (discussed in Section 3.2.1) is a more useful data structure than the linear queue.

D.3 Lists

A list (or a linked list) is a data structure made of nodes that point to each other. A node may be a single variable, an array, a stack, another list, or any other structure, but the main feature of lists is the use of *pointers*. This makes it easy to control the size of the list dynamically. A programming language may include statements such as `get_node` (to construct a new node from the pool of available storage) and `put_node` (to return the storage used by a node to that pool). Figure D.3 shows several ways to organize lists. A list can be singly-linked or doubly-linked, it can be cyclical, and the list elements may themselves be lists.

A queue may be implemented as a linked list. Inserting a new item is done by creating a new node and adding it to the list. Deleting an item is done by deleting the first (oldest) node.

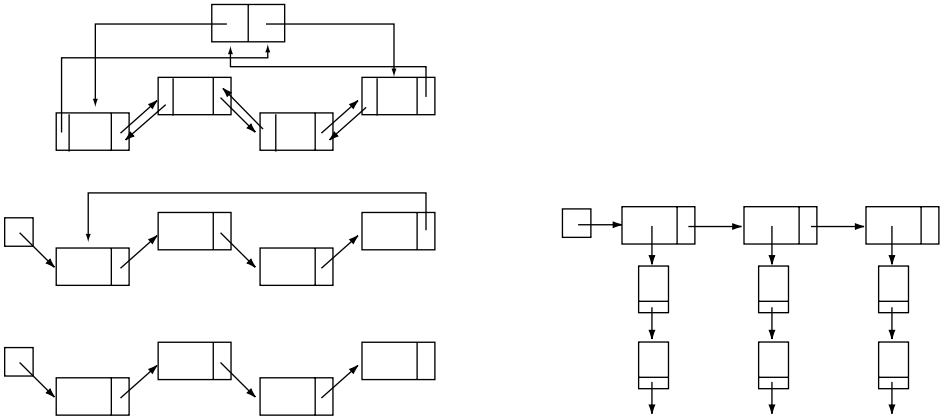


Figure D.3: Linked Lists.

D.4 Trees

A tree can be loosely defined as a data structure consisting of nodes connected with directed edges, where all the nodes are connected, there are no cycles, and one node is considered special. This node is called the *root* of the tree. Figure D.4a shows such a structure where any of the nodes can be considered the root. Notice the dashed edge. Including this edge in the structure would introduce a cycle, and thus change it from a tree to a general graph. Figure D.4b shows the same structure with node a chosen as the root and the remaining nodes rearranged to form the familiar shape of a tree. If there are edges leading directly from a node a to nodes b, c, and d, then the three nodes are called the *children* of a and a is their *parent*. If a node does not have any children, it is a *leaf*. A node that is neither a leaf nor the root is an *interior* node. The *depth* (or level) of a node is the length of the path from the root to the node. The root itself has depth zero. The height of a tree is the largest depth (or, alternatively, one less than the number of levels of the tree). Figure D.4b shows that each node in a tree is the root of a subtree (which may be empty).

In a typical practical implementation, a node a is a short array containing a data item and pointers. In most cases the array contains 1, 2, or 3 pointers, each pointing to the start of a linked list. The first pointer may point to a list containing nodes that are siblings of a (notice that the root does not have any siblings). The second pointer may point to a list containing nodes that are children of a (if a is a leaf, this pointer is null), and the third pointer may point to the parent of a (a null pointer, if a is the root). This way it is possible to travel in the tree to the right (from a to its next sibling on the right), down (from a to its first child), and up (from a to its parent). Such a tree may also change dynamically, with nodes being added, deleted, and modified.

Sometimes, it is useful to add another component (or field) to the array, with a code marking the node as either existing or deleted. This way a node can be

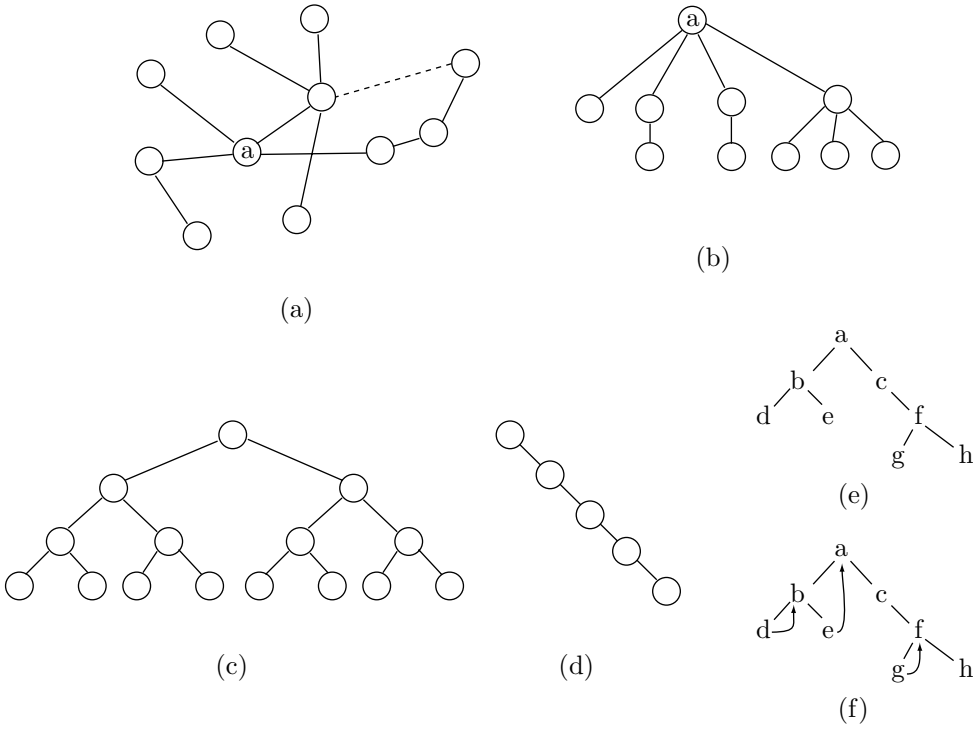


Figure D.4: Various Trees.

effectively deleted from the tree by setting this field to "deleted," without having to actually delete it and change pointers. This technique is called "lazy deletion" and is useful in applications where there is no need to return deleted nodes to the pool of available storage.

If a node can have just a few children, it may be possible to implement the entire tree in an array, without any pointers. The simplest example is a complete binary tree, where the two children of node a are stored in locations $2a$ and $2a+1$ and the parent of a can be found at array location $\lfloor a/2 \rfloor$ (this is discussed in Section 2.15).

◇ **Exercise D.3:** Can a ternary tree be implemented in this way?

A binary tree can be complete, skewed, or anything in between. This is illustrated by Figure D.4c,d,e.

An important operation on a tree is a *traversal*. A traversal follows pointers in such a way that each node of the tree is visited once. There are four types of tree traversals:

Post-order. All the children of a node are visited, then the node itself is visited. This is done recursively by the two recursive calls `postorder(L)`; `postorder(R)`; (where L and R are the two children of the root), followed by `visit(root)`. The post-order traversal of the tree of Figure D.4e is $(((D,E),B),(((G,H),F),C)),A$.

Pre-order: Visit a node, then all its children.

In-order: This is for binary tree traversal. Visit the left subtree of a node, then the node, then its right subtree.

Level-order: Visit all the nodes of level L, then proceed to level $L+1$.

◇ **Exercise D.4:** Show the pre-order, in-order, and level-order traversals of the tree of Figure D.4e.

A tree traversal is normally done recursively, except that level-order traversal uses a queue instead of a stack.

In a simple implementation, each node of a binary tree has two pointers, for its left and right children. It is obvious that not all the pointers are used. For example, in the binary tree of Figure D.4e there are 9 unused pointers (8 in the leaves and one in node C). Such unused pointer fields can be used for extra pointers called *threads*. One way to define threads is to find a node A with an unused field of a right child, and to store in this field a pointer to the successor of A in inorder traversal. The resulting three threads of the tree of Figure D.4e are shown in Figure D.4f. Obviously, threads can be defined differently and can be very useful in special applications, where the tree has to be traversed or searched in nonstandard ways. The price of adding the threads is an extra bit (a flag) in each node A, indicating whether an ordinary pointer to the right child, or a thread is stored in A.

Imagine an interior node a in a binary tree. It has two children, l and r, that are the roots of the left and right subtrees of a, respectively. If l has the largest data value in its subtree, and r satisfies the same thing for its subtree, then the binary tree is called a *heap*.

A binary search tree is an important type of tree. In such a tree all nodes in the left subtree of node a have data values that are smaller than the data value of a. Similarly, all the nodes in the right subtree of a have data values larger than a. If the data are not numbers, the relations "less than" and "greater than" have to defined. Binary search trees are described and used in Section 3.3. The main use of such a tree is quick search. Searching for a node in a binary search tree takes at most H steps, where H is the height of the tree.

Imagine a binary search tree that starts empty. When nodes are inserted into the tree, it grows. The order in which the nodes are inserted determines the shape of the tree. If the nodes being inserted have random data values, the resulting tree will be balanced, i.e., very similar to a complete tree. Its height will be approximately $\log_2 n$ where n is the number of nodes. Searching such a binary search tree with a million nodes takes at most 20 steps. If, on the other hand, the new nodes have monotonous data values (ascending or descending), the tree will end up being skewed. Its height (and thus the maximum search time) will be n.

Binary search trees are commonly used in applications that require many searches, so it is desirable to find a way to keep such a tree as close to balanced as possible, regardless of the order of node insertions. The AVL tree (named for its two developers, Adelson-Velskii and Landis) is such a data structure. Each node in an AVL tree has a *balance factor*, defined as the height of its left subtree minus the height of its right subtree. This balance factor is allowed to take only the three values 0, 1, and -1, i.e., the heights of the two subtrees of any node may differ by at most 1.

When the balance factor of a node becomes greater than 1, we say that the node is "out of balance on the left;" when it becomes less than -1, we say it is "out of balance on the right." These situations are immediately corrected, and the node brought back into balance. An AVL tree is a special case of a *height-balanced binary tree*, a structure defined by:

1. An empty binary tree is height balanced.
2. A nonempty binary tree is height balanced if its left and right subtrees are height balanced with balance factors of 0, 1, or -1.

An AVL tree is a height-balanced binary *search* tree, a special case of a height-balanced binary tree. Figure D.5a shows an example of an AVL tree that illustrates one important attribute of these trees, namely they don't have to be complete. The leaves of an AVL tree don't have to be on the same level or even on adjacent levels. However, it can be shown that an AVL tree is not very different from a complete binary tree because its height can be at most $\sqrt{2}\log_2 n$. An AVL tree is created either empty of with 1 node, so initially it is height-balanced. It is kept height-balanced after each insertion and deletion by special adjustments (called *rotations*) that restore the balance. The details can be found in texts on data structures.

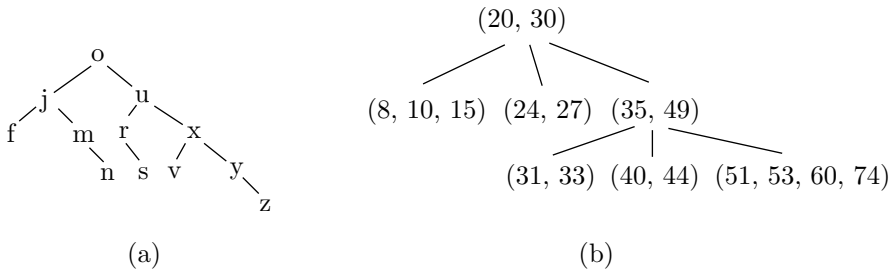


Figure D.5: (a) An AVL Tree. (b) A B-Tree.

The last type of tree to be mentioned here is the B-tree. This is an important type of tree because it is used by many operating systems to maintain the file directory of a disk. The growth of a B-tree is especially controlled to keep it well-balanced, leading to fast searches, but a B-tree, in spite of its name, is not a binary tree.

A node in a B-tree can have several children, and it is this property, together with the tree being well-balanced, that makes it a natural candidate for a disk directory. The following extreme example shows why. Imagine a tree where each node can have 100 children. If the tree has a height of 4, it can contain up to

$$1 + 100 + 100^2 + 100^3 = 1,010,101$$

nodes. One node among more than a million can be found in at most four steps! The price for this is, of course, a complex node structure, allowing for up to 100 children. In a disk directory each node is kept on the disk as a block. The size of a disk block varies, but is typically a few hundred bytes. Therefore, a block has

room for much information in the form of keys, pointers, and flags. At the same time, a disk access is much slower than memory access, so finding an item in the directory should involve as few disk accesses as possible. Once the disk is accessed, an entire block is read into memory. In memory, the block can be searched and its data processed quickly.

A node in a B-tree contains a count field, a pointer to a list of entries (records to be searched, where each record has a key and possibly some other data), and a pointer to a list of branches. The lists of entries and branches must be ordered, but they can be any ordered structures, such as linked lists, arrays, or binary search trees. The count field m contains the number of entries (m) and the number of branches ($m + 1$).

Figure D.5b shows a simple B-tree. Only the list of entries is shown, and for each entry only the key is shown. This is enough to understand how the tree is searched. The root contains keys 20 and 30. This means that branch 0 from the root leads to an entry list for all entries with keys that are less than 20. Branch 1 from the root leads to an entry list for all entries with keys in the range $[20, 30)$ and branch 2 leads to the list for entries with keys ≥ 30. To search for 34, for example, we start at the root, take branch 2 (since $33 \geq 30$), then branch 0 (since $33 < 35$). We arrive at the node with entry list $(31, 33)$ and take branch 2 (since $33 \geq 35$). This brings us to a null node, which is how we discover that key 34 is not in the tree.

D.5 Graphs

A graph is a general data structure consisting of nodes (or vertices) and edges (or arcs) connecting them. The graph is a general structure because not all nodes have to be connected and no node has to be special (such as the head or root of the graph). Figure D.6 shows examples of graphs. A graph may even consist of several, disconnected units. Edges may be directed or undirected, and may have labels (indicating weights or costs) associated with them. The main operations on graphs are the following:

1. Construct an empty graph g.
2. Insert a node a into an existing graph g.
3. Construct an edge e between nodes a and b of graph g.
4. Delete a node a from g. All the edges adjacent to the node should also be deleted.
5. Delete edge e between nodes a and b of graph g.
6. Determine whether graph g is empty.
7. Construct a list of all edges adjacent to node a of g.
8. Traverse graph g (i.e., visit each node once).
9. Find the minimum-cost path that leads from node a to node b in g.

There may be other, specialized operations for specific applications.

In an undirected graph, an edge between a and b is adjacent to both nodes. In a directed graph, an edge from a to b is said to be "adjacent from a" and "adjacent to b."

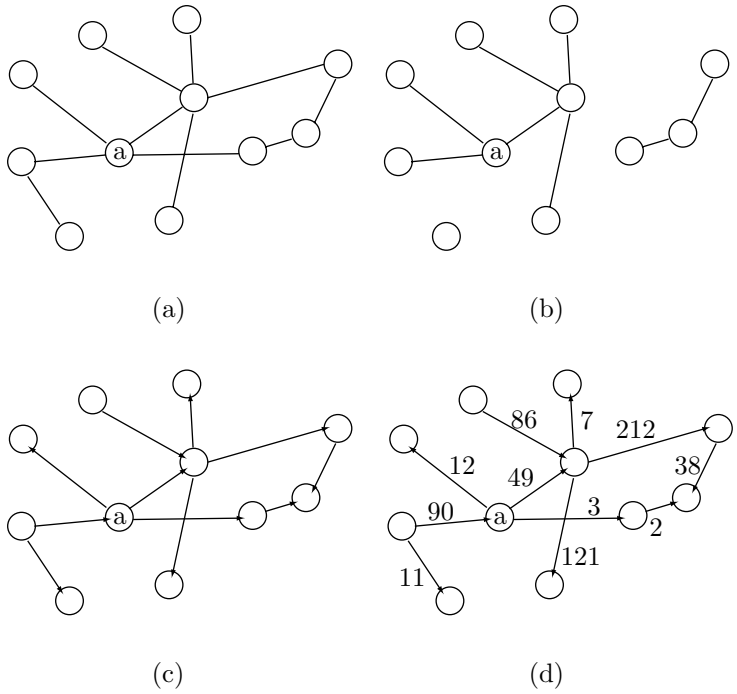


Figure D.6: Various Graphs.

D.6 Hashing

A hash table is a data structure allowing for fast insertions, searches, and deletions of data items. The table itself is just an array H, and the principle of hashing is to define a function h such that $h(k)$ produces an index to array H, where k is the key of a data item. The following examples illustrate the meaning of the terms "data item" and "key."

1. The LZRW1 method (Section 3.8), uses hashing to store pointers. The method uses the first three characters in the look-ahead buffer as a key which is hashed into a 12-bit number I used to index the hash table, an array of $2^{12} = 4,096$ pointers. The actual data stored in each location of the LZRW1 hash table is a pointer.

2. Virtually all computer languages use variables. A variable provides a name for a value that will be stored in memory, in a certain address A, when the program is eventually executed. When the program is compiled, each variable has two attributes, its name N (a string of characters assigned by the programmer) and its memory address A, assigned by the compiler. The compiler uses a hash table to store all the information about variables. The data item in this example is the address A of a variable; the key is the variable's name N. The compiler reads the name from the program source file, hashes it, finds it in the hash table, and retrieves the address in order to compile the current instruction. If the variable is not found

in the hash table, it is assigned an address, and both the name and address are stored in the table (in principle, only the address need be stored, but the name is also stored because of collisions; see below).

The hash function h takes as argument a key, which may be a number or a string. It scrambles or hashes the bits of the key to produce an index to array H. In practice the array size is normally 2^n, so the result produced by h should be an n-bit number. Hashing is a good data structure, since any operation on the hash table, adding, searching, or deleting, can be done in one step, regardless of the table size. The only problem is *collisions*. In most applications it is possible for two distinct keys k_1 and k_2 to get hashed to the same index, i.e., $h(k_1) = h(k_2)$ for $k_1 \neq k_2$. Example 2 above makes it easy to understand the reason for this. Assuming variable names of five letters, there may be $26^5 = 11,881,376$ variable names. Any program would use just a small percentage of this number, perhaps a few hundred or a few thousand names. The size of the hash table thus doesn't have to exceed a few thousand entries, and hashing 11.8 million names into a few thousand index values must involve many collisions.

◇ **Exercise D.5:** How many names are possible if a name consists of exactly eight letters and digits?

Terminology: Two different keys that hash to the same index are called *synonyms*. If a hash table contains m keys out of a set of M possible ones, then m/M is the *density* of the table, and $\alpha = m/2^n$ is its *loading factor*.

D.7 Hash Functions

A hash function should be easy to compute and should minimize collisions. The function should make use of all the bits of the key, such that changing even one bit would normally (although not always) produce a different index. An ideal hash function should also produce indexes that are uniformly distributed (calling the function many times with random keys should produce each index the same number of times). A function that produces, e.g., index 118 most of the time is obviously biased and leads to collisions. The function should also assume that many keys may be similar. In the case of variable names, e.g., programmers sometimes assign names such as "`A1`", "`A2`", "`A3`" to variables. A hash function that uses just the leftmost bits of a key would produce the same index for such names, leading to many collisions. Following are some examples of hash functions used in practice.

Mid-Square: The key k is considered an integer, it is squared and the middle n bits of k^2 extracted to become the index. Squaring k has the advantage that the middle bits of k^2 depend on *all* the bits of k. Thus two keys differing in one bit would tend to produce different indexes. A variation, suitable for large keys, is to divide the bits of the original key into several groups, add all the groups, square the result, and extract its middle n bits.

The keys "`A1`", "`A2`", and "`A3`", e.g., become the 16-bit numbers

$$01000001|00110001, \quad 01000001|00110010, \quad \text{and } 01000001|00110011.$$

After squaring and extracting the middle 8 bits, the resulting indexes are 158, 166, and 175, respectively.

Modulo: $h(k) = k \bmod m$. The result is the remainder of the integer division k/m, a number in the range $[0, m-1]$. In order for the result to be a valid index, the hash table size should be m. The value of m is critical and should be selected carefully. If m is a power of 2, say, 2^i, then the remainder of k/m is simply the i rightmost bits of k. This would be a very biased hash function. If m is even, then the remainder of k/m has the same parity as k (it is odd when k is odd and even when k is even). This again is a bad choice for m, since it produces a biased hash function that maps odd keys to odd location of H and even keys to even locations. If p is a prime number evenly dividing m then keys that are permutations of each other (e.g., "`ABC`", "`ACB`", and "`CBA`") may often be mapped to indexes that differ by p or by a multiple of p, again causing non-uniform distribution of the keys.

It can be shown that the modulo hash function achieves best results when m is a prime number that does not evenly divide $8^a \pm b$ where a and b are small numbers. In practice, good choices for m are prime numbers whose prime divisors are > 20.

Folding: This function is suitable for large keys. The bits constituting the key are divided into several groups, which are then added. The middle n bits of the sum are extracted to become the index. A variation is *reverse folding* where every other group of bits is reversed before being added.

> No, no. Look, here's the hash on the side because I didn't know how much you took.
>
> —Amy Wright as Shelley in *Stardust Memories* (1980).

D.8 Collision Handling

When an index i is produced by the hash function $h(k)$, the software using hashing should first check $H[i]$ for a collision. There must, therefore, be a way for the software to tell whether entry $H[i]$ is empty or occupied. Initializing all entries of H to zero is normally not enough, since zero may be a valid data item. A simple approach is to have an additional array F, of size $2^n/8$ bytes, where each bit is associated with an entry of H. Each bit of F acts as a flag indicating whether the corresponding entry of H is empty or not. The entire array F is initially set to zeros, indicating that all entries of H are empty. When the software decides to insert a data item in $H[i]$ it has to find the bit in F that corresponds to entry i and check it. The software should therefore calculate $j = \lfloor i/8 \rfloor$, $k = i - 8j$, and check bit k of byte F[j]. If the bit is zero, entry H[i] is empty and can be used for a new data item. The bit then has to be set, which is done by using k to select one of the eight masks

00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000

and logically OR it with $F[j]$. If the bit is 1, entry $H[i]$ is already occupied, and this is a collision. The software should be able to check and tell whether entry $H[i]$ contains the data item d that corresponds to key k. This is why the keys have to be saved, together with the data items, in the hash table.

What should the software do in case of a collision? The simplest thing is to check entries $H[i+1]$, $H[i+2]$,..., $H[2^n-1]$, $H[0]$, $H[1]$... until an empty entry is found or until the search reaches entry $H[i-1]$. In the latter case the software knows

that the data item is not in the table (if this was a search for an item) or that the table is full (if this was an attempt to insert a new item in the table). This process is called *linear search*. Searching for a data item, which in principle should take one step, can now, because of collisions, take up to 2^n steps. Experience also shows that a linear search causes occupied entries in the table to cluster, which is intuitively easy to understand. If the hash function is not ideal and hashes many keys to, say, index 54, then table entries 54, 55,... will quickly fill up, creating a cluster. Clusters also tend to grow and merge, creating even larger clusters and thereby increasing the search time. A theoretical analysis shows that the expected number of steps needed to locate an item when linear search is used is $(2-\alpha)/(2-2\alpha)$, where α is the loading factor (percent full of the table). For $\alpha = 0.5$ we can expect 1.5 steps on the average, but for $\alpha = 0.75$ the expected number of steps rises to 2.5, and for $\alpha = 0.9$ it become 5.5. It is clear that when linear search is used, the loading factor should be kept low (perhaps below 0.6–0.7). If more items need to be added to the table, a good solution is to declare a new table, twice as large as the original one, transfer all items from the old table to the new one (using a new hash function), and delete the old table.

> Dinner was at one o'clock; and on Monday, Tuesday, and Wednesday it consisted of beef, roast, hashed, and minced, and on Thursday, Friday, and Saturday of mutton. On Sunday they ate one of their own chickens.
>
> —W. Somerset Maugham, *Of Human Bondage*

A more sophisticated method of handling collisions is *quadratic search*. Assume that H is an array of size N. When entry $H[i]$ is found to be occupied, the software checks entries $H[(i \pm j^2) \bmod N]$ where $0 \le j \le (N-1)/2$. It can be shown that if N is a prime number of the form $4j+3$ (where j is an integer) quadratic search will end up examining every entry of H.

A third way to treat collisions is to rehash. The software should have a choice of several hashing functions $h_1, h_2, \ldots$. If $i = h_1(k)$ and $H[i]$ is occupied, the software should calculate $i = h_2(k)$ then try the new $H[i]$. Still another way is to generate an array R of N pseudo-random numbers in the range $[0, N-1]$ where each number appears once. If entry $H[i]$ is occupied, the software should set $i = (i + R[i]) \bmod N$ and try the new $H[i]$.

It is possible to design a *perfect hash function* that, for a given set of data items, will not have any collisions. This makes sense for sets of data that never change. Examples are the Bible, the works of Shakespeare, or any data written on a CD-ROM. The size N of the hash table should, in such a case, be normally larger than the number of data items. It is also possible to design a *minimal perfect hash function* where the hash table size equals the size of the data (i.e., no entries remain empty after all data items have been inserted). See [Czech 92], [Fox 91], and [Havas 93] for details on these special hash functions.

Bibliography

Czech, Z. J., et al. (1992) "An Optimal Algorithm for Generating Minimal Perfect Hash Functions," *Information Processing Letters* **43**:257–264.

Fox, E. A. et al. (1991) "Order Preserving Minimal Perfect Hash Functions and Information Retrieval," *ACM Transactions on Information Systems* **9**(2):281–308.

Havas, G. et al. (1993) *Graphs, Hypergraphs and Hashing* in Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93), Berlin, Springer-Verlag.

"Why," said he, "a magician could call up a lot of genies, and they would hash you up like nothing before you could say Jack Robinson. They are as tall as a tree and as big around as a church."

Mark Twain, *The Adventures Of Huckleberry Finn*

E
Error Correcting Codes

The problem of adding reliability to data has already been mentioned in Section 2.12. This appendix discusses general methods for detecting and correcting errors. Reliability is, in a sense, the opposite of data compression, since it is achieved by *increasing data redundancy*. Nevertheless, many practical situations call for reliable data, so a good data compression program should be able to use codes for increased reliability, if necessary.

Every time information is transmitted, on any channel, it may get corrupted by noise. In fact, even when information is stored in a storage device, it may become bad, because no hardware is absolutely reliable. This also applies to non-computer information. Speech sent on the air gets garbled by noise, wind, high temperature, etc. Speech, in fact, is a good starting point for understanding the principles of error-detecting and -correcting codes. Imagine a noisy cocktail party where everybody talks simultaneously, on top of blaring music. We know that even in such a situation it is possible to carry on a conversation, except that more attention than usual is needed.

E.1 First Principles

What makes our language so robust, so immune to errors? There are two properties, *redundancy* and *context*.

- Our language is redundant because only a very small fraction of all possible words are valid. A huge number of words can be constructed with the 26 letters of English. Just the number of seven-letter words, e.g., is $26^7 \approx 8.031$ billion. Yet only about 50,000 words are commonly used, and even the Oxford Dictionary lists "only" about 500,000 words. When we hear a garbled word our brain searches through many similar words, for the "closest" valid word. Computers are very good at such searches, which is why redundancy is the basis for error-detecting and error-correcting codes.

■ Our brains work by associations. This is why we humans excel in using the context of a message to repair errors in the message. In receiving a sentence with a garbled word or a word that doesn't belong, such as "pass the thustard please," we first use our memory to find words that are associated with "thustard." Then we use our accumulated life experience to select, among many possible candidates, the word that best fits in the present context. If we are on the freeway, we pass the bastard in front of us; if we are at dinner, we pass the mustard (or custard). Another example is the (corrupted) written sentence

if u cn rd ths u cn bcm a c prgmr!

which we can easily understand. Computers don't have much life experience and are notoriously bad at such tasks, which is why context is not used in computer codes. In extreme cases, where much of the sentence is bad, even we may not be able to correct it, and we may ask for a retransmission "say it again, Sam."

The idea of using redundancy to add reliability to information is due to Claude Shannon, the founder of information theory. It is not a trivial idea, since we are conditioned against it. Most of the time, we try to *eliminate* redundancy in computer data, in order to save space. In fact, all the data-compression methods discussed here do just that.

Figure E.1 shows the stages that a piece of computer data may go through when it is created, stored, transmitted, received, and used at the receiving end.

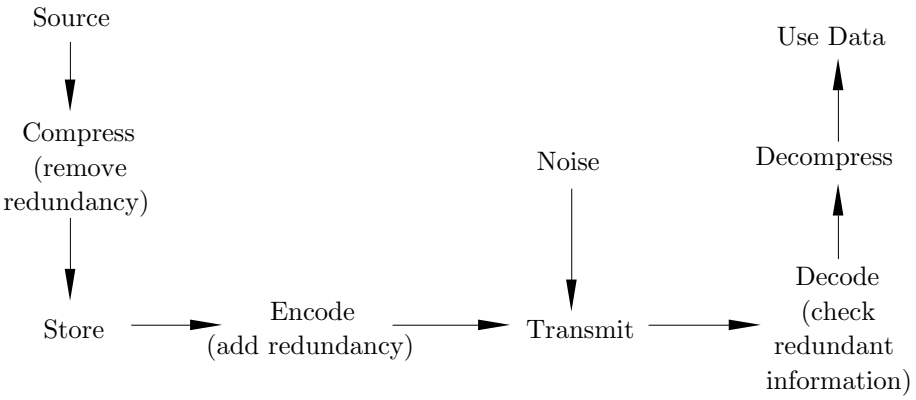


Figure E.1: Manipulating Information.

We discuss two approaches to reliable codes. The first approach is to *duplicate* the code, which leads to the idea of *voting codes*; the second one uses *check bits* and is based on the concept of *Hamming distance*.

E.2 Voting Codes

The first idea that usually occurs, when one thinks about redundancy, is to duplicate the message and send two copies. Thus if the code 110 has to be sent, one can send 110|110. A little thinking shows that this may, perhaps, be a good idea for error detection, but not for error correction. If the receiver receives two different copies, it cannot tell which one is good. What about triplicating the message? We can send 110|110|110 and tell the receiver to decide which of the three copies is good by comparing them. If all three are identical, the receiver assumes that they are correct. Moreover, if only two are identical and the third one different, the receiver assumes that the two identical copies are correct. This is the principle of *voting codes*. If all three copies are different, the receiver at least knows that an error has occurred, i.e., it can detect an error even though it cannot correct it.

To keep the analysis simple, let's limit ourselves to just 1-bit errors. When the three copies are received, the following cases are possible:

1. All three are identical. There are two subcases:
   1a: All three are good. This is the normal case.
   1b: All three have been corrupted in the same way. This is a rare case.
2. Two copies are identical, and the third one is different. Again, there are two subcases:
   2a: The two identical ones are good. This is the normal case.
   2b: They are bad. This is, we hope, a rare case.
3. All three copies are different.

Using the principle of voting, we assume that the three identical copies in case 1 are good. In case 1a our assumption is correct, and in case 1b, it isn't. Similarly, in case 2a the receiver makes the right decision, and in case 2b, the wrong one. In case 3, the receiver cannot correct the error, but at least it can detect one, so it does not make a wrong decision.

The only cases where the receiver makes the wrong decision (where the principle of voting does not work) are therefore 1b and 2b. A little thinking shows that the probability of case 1b is much smaller than that of 2b. We will, therefore, try to estimate the probability of case 2b and, if it is small enough, there would be no need to worry about case 1b.

It is hard to calculate the probability of two copies being garbled *in the same way*. We will, therefore, calculate the probability that one bit gets changed in two of the three copies (either in the same or in different bit positions). If this probability is small enough, there is no need to worry about case 2b, since its probability is even lower.

We denote by p the probability that one bit will get corrupted in our transmissions. The probability that one bit will go bad in two of the three copies is $\binom{3}{2}p^2 = 3p^2$. It is not simply p^2, since it is possible to select two objects out of three in $\binom{3}{2}$ = three ways.

[The notation $\binom{m}{n}$ is pronounced "m over n" and is defined as

$$\frac{m!}{n!(m-n)!}.$$

It is the number of ways n objects can be selected out of a set of m objects.]

Let's assume that $p = 10^{-6}$ (on average, one error in a million bits transmitted) and we want to send 10^8 bits. Without duplication we can expect $10^8 \times 10^{-6} = 100$ errors, an unacceptably high rate. With three copies sent, we have to send a total of 3×10^8 bits, and the probability that two out of the three copies will go wrong is 3×10^{-12}. The expected number of errors is thus $(3 \times 10^8) \times (3 \times 10^{-12}) = 9 \times 10^{-4} = 0.0009$ errors, a comfortably small number.

If higher reliability is needed, more copies can be sent. A code where each symbol is duplicated and sent nine times is extremely reliable. Voting codes are thus simple, reliable, and have only one disadvantage, they are too long. In practical situations, sending nine, or even three, copies of each message may be prohibitively expensive. This is why much research has been done in the field of coding in the last 40 years and, today, many sophisticated codes are known that are more reliable and shorter than voting codes.

E.3 Check Bits

In practice, error-detection and correction is usually done by means of *check bits*, which are added to the original *information bits* of each word of the message. In general, k check bits are appended to the original m information bits, to produce a *codeword* of $n = m + k$ bits. Such a code is referred to as an (n, m) code. The codeword is then transmitted to the receiver. Only certain combinations of the information bits and check bits are valid, in analogy to a natural language. The receiver knows what the valid codewords are. If a non-valid codeword is received, the receiver considers it an error. In Section E.7 we show that, by adding more check bits, the receiver can also correct certain errors, not just detect them. The principle of error correction is that, on receiving a bad codeword, the receiver selects the valid codeword that is the "closest" to it.

Example: A set of 128 symbols needs to be coded. This implies $m = 7$. If we select $k = 4$, we end up with 128 valid codewords, each 11 bits long. This is therefore an $(11, 7)$ code. The valid codewords are selected from a total of $2^{11} = 2,048$ possible codewords, so there remain $2,048 - 128 = 1,920$ non-valid codewords. The big difference between the number of valid and non-valid codewords means that if a codeword gets corrupted, chances are it will change to a non-valid one.

It may, of course, happen that a valid codeword gets changed, during transmission, to another valid codeword. Our codes are thus not completely reliable, but can be made more and more reliable by adding more check bits and by selecting the valid codewords carefully. One of the basic theorems of information theory says that codes can be made as reliable as desired by adding check bits, as long as n (the size of a codeword) does not exceed the channel's capacity.

It is important to understand the meaning of the word "error" in data transmission. When a codeword is received, the receiver always receives n bits, but some of them may be bad. A bad bit does not disappear, nor does it change into something other than a bit. A bad bit simply changes its value, either from 0 to 1, or from 1 to 0. This makes it relatively easy to correct the bit. The code should tell the receiver which bits are bad, and the receiver can then easily correct those bits by inverting them.

In practice, bits may be sent on a wire as voltages. A binary 0 may, e.g., be represented by any voltage in the range 3–25 volts. A binary 1 may similarly be represented by the voltage range of −25v to −3v. Such voltages tend to drop over long lines, and have to be amplified periodically. In the telephone network there is an amplifier (a *repeater*) every 20 miles or so. It looks at every bit received, decides whether it is a 0 or a 1 by measuring the voltage, and sends it to the next repeater as a clean, fresh pulse. If the voltage has deteriorated enough in passage, the repeater may make a wrong decision when sensing it, which introduces an error into the transmission. At present, typical transmission lines have error rates of about one in a billion, but under extreme conditions—such as in a lightning storm, or when the electric power suddenly fluctuates—the error rate may suddenly increase, creating a burst of errors.

E.4 Parity Bits

A parity bit can be added to a group of m information bits to complete the total number of 1 bits to an odd number. Thus the (odd) parity of the group 10110 is 0, since the original group plus the parity bit would have an odd number (3) of ones. Even parity can also be used, and the only difference between odd and even parity is that, in the case of even parity, a group of all zeros is valid, whereas, with odd parity, any group of bits with a parity bit added cannot be all zeros.

Parity bits can be used to design simple, but not very efficient, error-correcting codes. To correct 1-bit errors, the message can be organized as a *rectangle* of dimensions $(r-1)\times(s-1)$. A parity bit is added to each row of $s-1$ bits, and to each column of $r-1$ bits. The total size of the message (Table E.2a) becomes $s\times r$.

0	1	0	0	**1**
1	0	1	0	**0**
0	1	1	1	**1**
0	0	0	0	**0**
1	1	0	1	**1**
0	**1**	**0**	**0**	**1**

(a)

0	1	0	0	**1**
1	0	1	**0**	
0	1	**0**		
0	**0**			
1				

(b)

Table E.2: Parity Bits.

If only one bit gets bad, a check of all $s-1+r-1$ parity bits will discover it, since only one of the $s-1$ parities and only one of the $r-1$ ones will be bad.

The overhead of a code is defined as the number of parity bits divided by the number of information bits. The overhead of the rectangular code is, therefore,

$$\frac{(s-1+r-1)}{(s-1)(r-1)}\approx\frac{s+r}{s\times r-(s+r)}.$$

A similar, slightly more efficient, code is a triangular configuration, where the information bits are arranged in a triangle, with the parity bits at the diagonal

(Table E.2b). Each parity bit is the parity of all the bits in its row *and* column. If the top row contains r information bits, the entire triangle has $r(r+1)/2$ information bits and r parity bits. The overhead is thus

$$\frac{r}{r(r+1)/2} = \frac{2}{r+1}.$$

It is also possible to arrange the information bits in a number of two-dimensional planes, to obtain a three-dimensional cube, three of whose six outer surfaces are made up of parity bits.

It is not obvious how to generalize these methods to more than 1-bit error correction.

Symbol	code_1	code_2	code_3	code_4	code_5	code_6	code_7
A	0000	0000	001	001001	01011	110100	110
B	1111	1111	010	010010	10010	010011	0
C	0110	0110	100	100100	01100	001101	10
D	0111	1001	111	111111	10101	101010	111
k:	2	2	1	4	3	4	

Table E.3: Code Examples With $m = 2$.

E.5 Hamming Distance and Error Detecting

Richard Hamming developed the concept of distance, in the 1950s, as a general way to use check bits for error detection and correction.

To illustrate this concept, we start with a simple example involving just four symbols A, B, C, and D. Only 2 information bits are required, but the codes of Table E.3 add some check bits, for a total of 3–6 bits per symbol. code_1 is simple. Its four codewords were selected from the 16 possible 4-bit numbers, and are not the best possible ones. When the receiver receives one of them, say, 0111, it assumes that there is no error and the symbol received is D. When a non-valid codeword is received, the receiver signals an error. Since code_1 is not the best possible, not every error is detected. Even if we limit ourselves to single-bit errors, this code is not very good. There are 16 possible single-bit errors in our 4-bit codewords and, of those, the following 4 cannot be detected: A 0110 changed during transmission to 0111, a 0111 changed to 0110, a 1111 corrupted to 0111, and a 0111 garbled to 1111. The error detection rate is thus 12 out of 16, or 75%. In comparison, code_2 does a much better job; it can detect every single-bit error.

◇ **Exercise E.1:** Prove the above statement.

We therefore say that the four codewords of code_2 are sufficiently *distant* from each other. The concept of distance of codewords is, fortunately, easy to define.

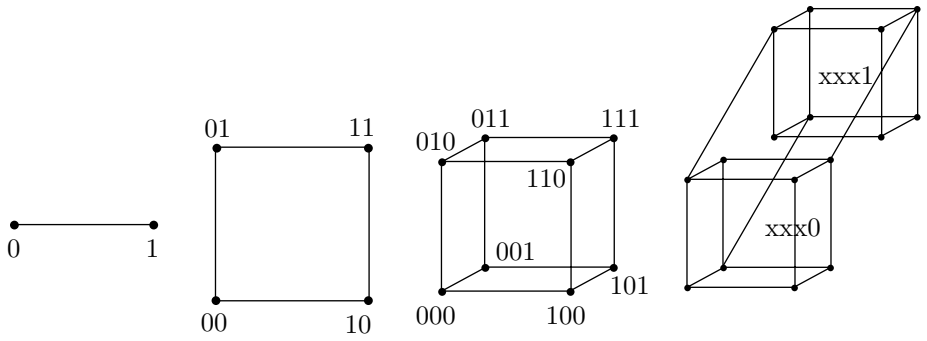


Figure E.4: Cubes of Various Dimensions and Corner Numbering.

Definitions: (1) Two codewords are a Hamming distance d apart if they differ in exactly d of their n bits and (2) a code has a Hamming distance of d if every pair of codewords in the code is, at least, a Hamming distance d apart.

These definitions have a simple geometric interpretation. Imagine a hypercube in n-dimensional space. Each of its 2^n corners can be numbered by an n-bit number (Figure E.4), such that each of the n bits corresponds to one of the n dimensions. In such a cube, points that are directly connected have a Hamming distance of 1, points with a common neighbor have a Hamming distance of 2, etc. If a code with a Hamming distance of 2 is needed, only points that are not directly connected should be selected as valid codewords.

The reason code_2 can detect all single-bit errors is that it has a Hamming distance of 2. The distance between valid codewords is 2, so a 1-bit error always changes a valid codeword into a non-valid one. When two bits go bad, a valid codeword is moved to another codeword at distance 2. If we want that other codeword to be non-valid, the code must have at least distance 3.

In general, a code with a Hamming distance of $d+1$ can detect all d-bit errors. code_3 has a Hamming distance of 2 and thus can detect all 1-bit errors even though it is short ($n = 3$).

◇ **Exercise E.2:** Find the Hamming distance of code_4.

It is now obvious that we can increase the reliability of our transmissions to any desired level, but this feature does not come free. As always, there is a tradeoff, or a price to pay, in the form of the overhead. Our codes are much longer than m bits per symbol because of the added check bits. A measure of the price is $n/m = (m+k)/m = 1+k/m$, where the quantity k/m is called the *overhead* of the code. In the case of code_1 the overhead is 2 and, in the case of code_3, it is 3/2.

Example: A code with a single check bit, which is a parity bit (even or odd). Any single-bit error can easily be detected, since it creates a non-valid codeword. Such a code therefore has a Hamming distance of 2. code_3 above uses a single, odd, parity bit.

Example: A 2-bit error-detecting code for the same four symbols (see code_4).

It must have a Hamming distance of 4, and one way of generating it is to duplicate code_3.

E.6 Hamming Codes

The principle of error-correcting codes is to separate the codes even farther by adding more redundancy (more check bits). When an invalid codeword is received, the receiver corrects the error by selecting the valid codeword that is closest to the one received. code_5 has a Hamming distance of 3. When one of its four codewords has a single bit changed, it is 1-bit distant from the original one, but is still 2 bits distant from any of the other codewords. Thus, if there is only one error, the receiver can always correct it. The receiver does that by comparing every codeword received to the list of valid codewords. If no match is found, the receiver assumes that a 1-bit error has occurred, and it corrects the error by selecting the codeword that is closest to the one received.

In general, when d bits go wrong in a codeword C_1, it turns into an invalid codeword C_2 at a distance d from C_1. If the distance between C_2 and the other valid codewords is at least $d+1$, then C_2 is closer to C_1 than it is to any other valid codeword. This is why a code with a Hamming distance of $d+(d+1)=2d+1$ is needed to correct all d-bit errors.

How are the codewords selected? The problem is to select a good set of 2^m codewords out of the 2^n possible ones. The first approach uses brute force. It is easy to write a computer program that will examine all the possible sets of 2^m codewords, and select one that has the right distance. The problems with this approach are: (1) The time and storage required at the receiving end to verify and correct the codes received and (2) the amount of time it takes to examine all the possible sets of codewords.

1. The receiver must have a list of all the 2^n possible codewords. For each codeword it must have a flag indicating whether it is valid and, if not, which valid codeword is the closest to it. Every codeword received has to be searched and located in this list in order to verify it.

2. In the case of four symbols, only four codewords need be selected. For code_1 and code_2, they had to be selected from among 16 possible numbers, which can be done in $\binom{16}{4} = 7,280$ ways. It is possible to write a simple program that will systematically examine sets of four codewords until it finds a set with the required distance. In the case of code_4, the four codewords had to selected from a set of 64 numbers, which can be done in $\binom{64}{4} = 635,376$ ways. It is still possible to write a program that will systematically explore all the possible codeword selections for this case. In practical cases, however, where sets with hundreds of symbols are involved, the number of possibilities in selecting sets of codewords is too large even for the fastest computers to handle comfortably.

Clearly, a clever algorithm is needed, to select the best codewords, and to verify them on the fly, as they are being received. The transmitter should use the algorithm to generate the codewords when they have to be sent, and the receiver should use it to check them when they are received. The approach described here is due to Richard Hamming. In Hamming's codes [Hamming 86] the n bits of a codeword are indexed from 1 to n. The check bits are those with indexes that are

powers of 2. Thus bits b_1, b_2, b_4, $b_8, \ldots$ are check bits, and b_3, b_5, b_6, b_7, $b_9, \ldots$ are information bits. The index of each information bit can be written as the sum of the indexes of certain check bits. Thus b_7 can be written as b_{1+2+4} and is, therefore, used in determining the values of check bits b_1, b_2, b_4. *The check bits are simply parity bits.* The value of b_2, e.g., is the parity (odd or even) of b_3, b_6, b_7, $b_{10}, \ldots$ etc., since $3 = 2 + 1$, $6 = 2 + 4$, $7 = 2 + 1 + 4$, $10 = 2 + 8, \ldots$.

Example: A 1-bit error-correcting code for the set of symbols A, B, C, and D. It must have a Hamming distance of $2d + 1 = 3$. Two information bits are needed to code the four symbols, so they must be: b_3 and b_5. The parity bits are therefore b_1, b_2, and b_4. Since $3 = 1 + 2$ and $5 = 1 + 4$, the 3 parity bits are defined as b_1 is the parity of bits b_3 and b_5, b_2 is the parity of b_3, and b_4 is the parity of b_5. This is how code_5 of Table E.3 was constructed.

Example: A 1-bit error-correcting code for a set of 256 symbols. It must have a Hamming distance of $2d + 1 = 3$. Eight information bits are required to code the 256 symbols, so they must be b_3, b_5, b_6, b_7, b_9, b_{10}, b_{11}, and b_{12}. The parity bits are, therefore, b_1, b_2, b_4, and b_8. The total size of the code is 12 bits. The following relations define the 4 parity bits:
$3 = 1 + 2$, $5 = 1 + 4$, $6 = 2 + 4$, $7 = 1 + 2 + 4$, $9 = 1 + 8$, $10 = 2 + 8$, $11 = 1 + 2 + 8$, and $12 = 4 + 8$.
They imply that b_1 is the parity of b_3, b_5, b_7, b_9, and b_{11}.

◇ **Exercise E.3:** What are the definitions of the other parity bits?

◇ **Exercise E.4:** Construct a 1-bit error-correcting Hamming code for 16-bit codes ($m = 16$).

A common question at this point is how the number of parity bits is determined. The answer is that it is determined implicitly. We know that m data bits are needed, and we also know that bits $b_1, b_2, b_4, b_8, \ldots$ should be the parity bits. We thus allocate the first m bits of the set $b_3, b_5, b_6, b_7, b_9, b_{10}, b_{11}, \ldots$ to the data, and this implicitly determines the number of parity bits needed.

What is the size of a general Hamming code? The case of a 1-bit error-correcting code is easy to analyze. Given a set of 2^m symbols, 2^m valid codewords are needed. We are looking for the smallest k required to construct codewords of size $m + k$ and Hamming distance 3. The 2^m valid codewords should be selected from a total of 2^n numbers (where $n = m + k$), such that each codeword consists of m information bits and k check bits.

Since we want any single-bit error in a codeword to be corrected, such an error should not take us too far from the original codeword. A single-bit error takes us to a codeword at distance 1 from the original one. As a result, all codewords at distance 1 from the original codeword should be non-valid. Each of the original 2^m codewords is n bits long and thus has n codewords at distance 1 from it. They should be declared non-valid. This means that the total number of codewords needed (valid plus non-valid) is $2^m + n2^m = (1 + n)2^m$. This number has to be selected from the 2^n available numbers, so we end up with the relation $(1 + n)2^m \leq 2^n$. Since $2^n = 2^{m+k}$, we get $1 + n \leq 2^k$ or $k \geq \log_2(1 + n)$. The following table illustrates the meaning of this relation for certain values of m.

n:	4	7	12	21	38	71
k:	2	3	4	5	6	7
$m = n - k$:	2	4	8	16	32	64
k/m:	1	.75	.5	.31	.19	.11

There is a geometric interpretation that provides another way of obtaining the same result. We imagine 2^m spheres of radius one tightly packed in our n-dimensional cube. Each sphere is centered on one of the corners and encompasses all its immediate corner neighbors. The *volume* of a sphere is defined as the number of corners it includes, which is $1+n$. The spheres are tightly packed, but they don't overlap, so their total volume is $(1+n)2^m$, and this should not exceed the total volume of the cube, which is 2^n.

The case of a 2-bit error-correcting code is similarly analysed. Each valid codeword should define a set that includes itself, the n codewords at distance 1 from it, and the set of $\binom{n}{2}$ codewords at distance 2 from it, a total of $\binom{n}{0}+\binom{n}{1}+\binom{n}{2} = 1+n+n(n-1)/2$. Those sets should be non-overlapping, which implies the relation

$$\big(1+n+n(n-1)/2\big)2^m \le 2^n \Rightarrow 1+n+n(n-1)/2 \le 2^k \Rightarrow k \ge \log_2\big(1+n+n(n-1)/2\big).$$

In the geometric interpretation we again imagine 2^m spheres of radius 2 each. Each sphere centered around a corner and containing the corner, its n immediate neighbors, and its $\binom{n}{2}$ second place neighbors (corners differing from the center corner by 2 bits).

E.7 The SEC-DED Code

However, even though we can estimate the length of a 2-bit error-correcting Hamming code, we don't know how to construct it! The best that can be done today with Hamming codes is single-error-correction combined with double-error-detection. An example of such a SEC-DED code is code_6. It was created by simply adding a parity bit to code_5.

◇ **Exercise E.5:** Table E.3 contains one more code, code_7. What is it?

The receiver checks the SEC-DED code in two steps. In step 1, the single parity bit is checked. If it is bad, the receiver assumes that a 1-bit error occurred, and it uses the other parity bits, in step 2, to correct the error. It may happen, of course, that three or even five bits are bad, but the simple SEC-DED code cannot detect such errors.

If the single parity is good, then there are either no errors, or two bits are bad. The receiver switches to step 2, where it uses the other parity bits to distinguish between these two cases. Again, there could be four or six bad bits, but this code cannot handle these cases.

The SEC-DED code has a Hamming distance of 4. In general, a code for c-bit error correction and d-bit error detection should have a distance of $c+d+1$.

E.8 Generating Polynomials

There are many approaches to the problem of developing codes for more than 1-bit error correction. They are, however, more complicated than Hamming's method, and require a background in group theory and Galois fields. In this section we briefly sketch one such approach, using the concept of a *generating polynomial*.

We use the case $m = 4$ for illustration. Sixteen codewords are needed, which can be used to code any set of 16 symbols. We know from the discussion above that, for 1-bit error correction, 3 parity bits are needed, bringing the total size of the code to $n = 7$. Here is an example of such a code:

0000000	0001011	0010110	0011101	0100111	0101100	0110001	0111010
1000101	1001110	1010011	1011000	1100010	1101001	1110100	1111111

Note that it has the following properties:

- The sum (modulo 2) of any two codewords equals another codeword. This implies that the sum of any number of codewords is a codeword. The 16 codewords above thus form a *group* under this operation.
(Addition and subtraction modulo-2 is done by $0+0 = 1+1 = 0$, $0+1 = 1+0 = 1$, $1 - 0 = 0 - 1 = 1$. The definition of a group should be reviewed in any text on algebra.)

- Any circular shift of a codeword is another codeword. This code is thus *cyclic*.

- It has a Hamming distance of 3, as required for 1-bit error correction.

Interesting properties! The 16 codewords were selected from the 128 possible ones by means of a generator polynomial. The idea is to look at each codeword as a polynomial, where the bits are the coefficients. Here are some 7-bit codewords associated with polynomials of degree 6.

1	0	0	1	1	1	1
x^6			$+x^3$	$+x^2$	$+x$	$+1$
0	1	1	0	0	1	0
	x^5	$+x^4$			$+x$	
0	1	0	0	1	1	1
	x^5			$+x^2$	$+x$	$+1$

The 16 codewords above have been selected by finding the degree-6 polynomials that are evenly divisible (modulo 2) by the generating polynomial $x^3 + x + 1$. For example, the third codeword '0100111' in the table corresponds to the polynomial $x^5 + x^2 + x + 1$, which is divisible by $x^3 + x + 1$ because $x^5 + x^2 + x + 1 = (x^3 + x + 1)(x^2 + 1)$.

To understand how such polynomials can be calculated, let's consider similar operations on numbers. Suppose we want to know the largest multiple of 7 that is ≤ 30. We divide 30 by 7, obtaining a remainder of 2, and then subtract the 2 from the 30, getting 28. Similarly with polynomials. Let's start with the 4 information bits 0010, and calculate the remaining 3 parity bits. We write 0010*ppp*, which gives us the polynomial x^4. We divide x^4 by the generating polynomial, obtaining a

remainder of $x^2 + x$. Subtracting that remainder from x^4 gives us something that will be evenly divisible by the generating polynomial. The result of the subtraction is $x^4 + x^2 + x$, so the complete codeword is 0010110.

Any generating polynomial can get us the first two properties. To get the third property (the necessary Hamming distance), the right generating polynomial should be used, and it can be selected by examining its roots. This topic is outside the scope of this book, but it is discussed in any text on error-correcting codes. A common example of a generating polynomial is CRC (Section 3.23).

Bibliography

Hamming, Richard (1950) "Error Detecting and Error Correcting Codes," *Bell Systems Technical Journal* **29**:147–160, April.

Hamming, Richard (1986) *Coding and Information Theory*, 2nd Ed., Englewood Cliffs, NJ, Prentice-Hall.

Lin, Shu (1970) *An Introduction to Error Correcting Codes*, Englewood Cliffs, NJ, Prentice-Hall.

Errors using inadequate data are much
less than those using no data at all.

Charles Babbage (1792–1871)

Give me fruitful error any time, full of seeds, bursting with its
own corrections. You can keep your sterile truth for yourself.

Wilfredo Pareto (1848–1923)

F
Finite State Automata

A *finite automaton* (FA, also called a *finite-state automaton* or a *finite-state machine*) is a mathematical tool used to describe processes involving inputs and outputs. An FA can be in one of several states and can switch between states depending on symbols that it inputs. Once it settles in a state, it reads the next input symbol, performs some computational task associated with the new input, outputs a symbol, and switches to a new state depending on the input. Notice that the new state may be identical to the current state. Figure F.2a shows an example of an FA with four states labeled 1 through 4. The input consists of a string of symbols from the alphabet `abc` and the output is a string of symbols from `mnpq`. The FA starts in state 1. If it inputs an `a`, it stays in state 1 and outputs an `m`. If it inputs a `b`, it switches to state 2 and outputs a `p`. Notice that this particular FA can get "stuck" if it happens to be in a state that does not tell it what to do with a certain input. This will happen, for example, with the input string "`abbc...`", since the `c` will be input while the FA is in state 4, where it expects an `a`. We say that "`abbc...`" is not accepted by the FA. The edges of the graph are labeled by pairs (input, output).

Figure F.2b shows an example of a two-state FA that can be used to add binary numbers. The FA starts in state 1 (no carry) and inputs a pair of bits. If the pair is 11, the FA outputs a 0 and switches to state 2 (carry), where the next pair of bits is input and is added to a carry bit of 1. Table F.1 shows the individual steps in adding the two 6-bit numbers 14 and 23 (these are actually 5-bit numbers, but the simple design of this FA requires that an extra 0 bit be appended to the left end of every number). Since adding numbers is done from right to left, the six columns of the table should also be read in this direction.

Figure F.2c shows an example of a three-state FA where each edge is labeled by an input symbol and a probability, but no output. The diagram illustrates an FA that inputs a stream of bits where the first bit has a 50% probability of being a 0 or a 1, and each consecutive bit is likely to be identical to its predecessor. A zero follows another zero with a probability of 2/3, and a one follows a one with a

Step:	6	5	4	3	2	1
14:	0	0	1	1	1	0
23:	0	1	0	1	1	1
Input:	00	01	10	11	11	01
Initial state:	2	2	2	2	1	1
Output (37):	1	0	0	1	0	1
Final state:	1	2	2	2	2	1

Table F.1: Adding 14 and 23.

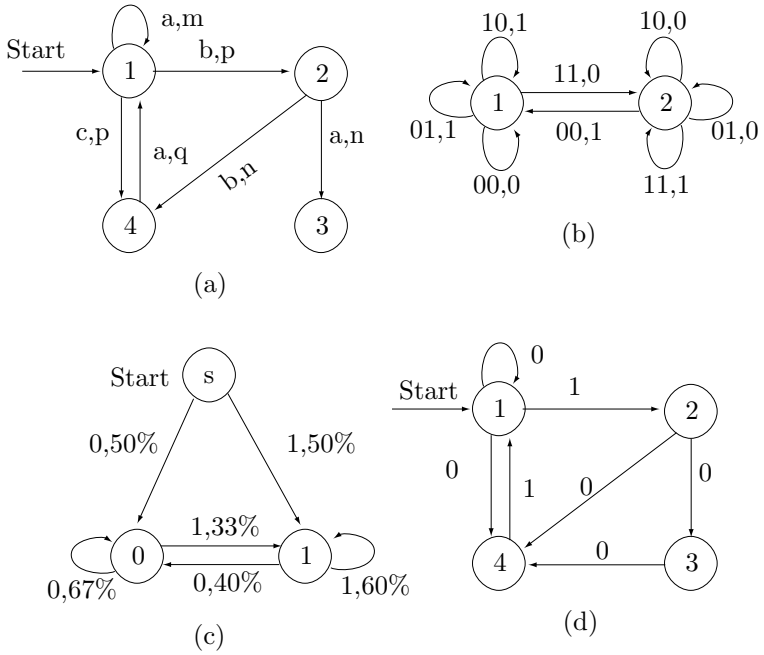


Figure F.2: Examples of FAs.

probability of 3/5. This FA does not produce any outputs, but any practical FA should, of course, do something (i.e., produce some output).

In general, an FA is fully defined by specifying the following:

1. The set of states S.
2. The alphabet A of the input and output symbols.
3. A function $f: S \times A \to S \times A$. An example is $f(X, j) = (Y, i)$ meaning, if the current state is X and the next input is j, switch to state Y and output i.
4. The initial state S_0.

Mathematically, an FA is a quartet (S, A, f, S_0).

An FA is normally *deterministic* (DFA), since for every state and input there is just one output and one next state. However, there can also be *nondeterministic* FA,

denoted by NFA. Figure F.2d shows an example. When a zero is input in state 2, the NFA can transit to either state 3 or state 4. NFAs are useful in proving theorems and also in the theory of languages and the theory of computations. Following the states and inputs, it is trivial to verify that the input string "10101010..." is accepted by the NFA of Figure F.2d (it alternates between states 1, 2, and 4), as are the strings 100 and 101, but the string 1000 is not accepted. It should be noted that every DFA is an NFA.

◇ **Exercise F.1:** Is the string "1001010010..." accepted by the NFA of Figure F.2d?

The set of all input strings accepted by an FA is called the *language* of the FA. It can be shown that the languages accepted by finite automata are the languages denoted by *regular expressions*, so we conclude this short chapter by defining regular expressions. We start by illustrating the concepts of *concatenation* and *closure*.

Let $L_1 = (10, 1)$ and $L_2 = (011, 11)$ be two sets of symbols (binary strings). The concatenation of L_1 and L_2 is denoted by L_1L_2 and is the set of strings that are concatenations of 10 (or 1) and 011 (or 11). Thus, $L_1L_2 = (10011, 1011, 111)$. The closure of a set L of symbols is denoted by L^*. It is the set of all strings made of symbols from L (including the empty set). Thus, if $L = (10, 11)$, then $L^* = ((), 10, 11, 1010, 1011, 1110, 1111, \ldots)$, where the empty pair of parentheses denotes the empty set. If L is a set of symbols, then the regular expressions over L are defined recursively as follows:

1. The empty symbol is a regular expression.
2. The empty set () is a regular expression.
3. For each $s \in L$, the set (s) is a regular expression.
4. If r and s are regular expressions, then $(r + s)$ (union of sets), (rs) (set concatenation), and (r^*) (set closure) are regular expressions.

Finite automata are used by several compression methods, among them WFA (Section 4.31) and dynamic Markov coding (Section 8.8).

[Hopcroft and Ullman 79] is a detailed introduction to FAs and their properties.

Bibliography

Hopcroft, John E. and Jeffrey D. Ullman (1979) *Introduction to Automata Theory, Languages, and Computation*, Reading, MA, Addison-Wesley.

CAT, n. A soft, indestructible automaton provided by nature to be kicked when things go wrong in the domestic circle.

Ambrose Bierce. *The Devil's Dictionary*

G
Gallery of Images

New data compression methods that are developed and implemented have to be tested. Testing different methods on the same data makes it possible to compare their performance both in compression efficiency and speed. This is why there are standard collections of test data such as the Calgary Corpus and the Canterbury Corpus (mentioned in the Preface), and the ITU-T set of eight training documents for fax compression (Section 2.13.1).

The need for standard test data has also been felt in the field of image compression, and there currently exist collections of still images commonly used by researchers and implementors in this field. Three of the four images shown here, namely "lena," "mandril," and "peppers" are arguably the most well-known of them. They are continuous-tone images, although "lena" has some features of a discrete-tone image.

Each image is accompanied by a detail, showing individual pixels. It is easy to see why the "peppers" image is continuous-tone. Adjacent pixels that differ much in color are fairly rare in this image. Most neighbor pixels are very similar. In contrast, the "mandril" image, even though natural, is a bad example of a continuous-tone image. The detail (showing part of the right eye and the area around it) shows that many pixels differ considerably from their immediate neighbors because of the animal's facial hair in this area. This image compresses badly under any compression method. However, the nose area, with mostly blue and red, is continuous-tone. The "lena" image is mostly pure continuous-tone, especially the wall and the bare skin areas. The hat is good continuous-tone, whereas the hair and the plume on the hat are bad continuous-tone. The straight lines on the wall and the curved parts of the mirror are features of a discrete-tone image.

The "lena" image is widely used by the image processing community, in addition to being popular in image compression. Because of the interest in it, its origin and history have been researched and are well documented. This image is part of

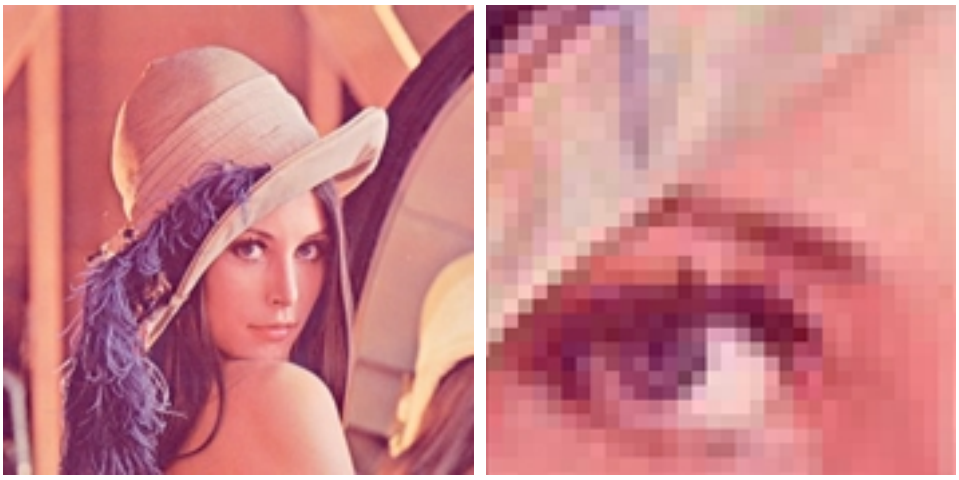


Figure G.1: Lena and Detail.

the *Playboy* centerfold for November, 1972. It features the Swedish playmate Lena Soderberg (née Sjooblom) and it was discovered, clipped, and scanned in the early 1970s by an unknown researcher at the University of Southern California for use as a test image for his image compression research. It has since become the most important, well-known, and commonly-used image in the history of imaging and electronic communications. As a result, Lena is currently considered by many the First Lady of the Internet. *Playboy*, which normally prosecutes unauthorized users of its images, has found out about the unusual use of one of its copyrighted images, but decided to give its blessings to this particular "application."

Lena herself currently lives and works in Sweden. She was told of her "fame" in 1988, was surprised and amused by it, and was invited to attend the 50th Anniversary IS&T (the society for Imaging Science and Technology) conference in Boston in May, 1997. At the conference she autographed her picture, posed for new pictures (one is available on the internet) and gave a presentation (about herself, not image compression).

The three images are widely available for downloading on the internet.

Figure G.5 shows a typical discrete-tone image, with a detail shown in Figure G.6. Notice the straight lines and the text, where certain characters appear several times (a source of redundancy). This particular image has few colors, but in general, a discrete-tone image may have many colors.

> The village of Lena, Illinois is located approximately 9 miles west of Freeport, Illinois and 50 miles east of Dubuque, Iowa. We are on the edge of the rolling hills of Northwestern Illinois and only 25 miles south of Monroe, Wisconsin. The current population of Lena is approximately 2800 souls engaged in the farming business.
>
> From `http://www.lena.il.us/History.htm`

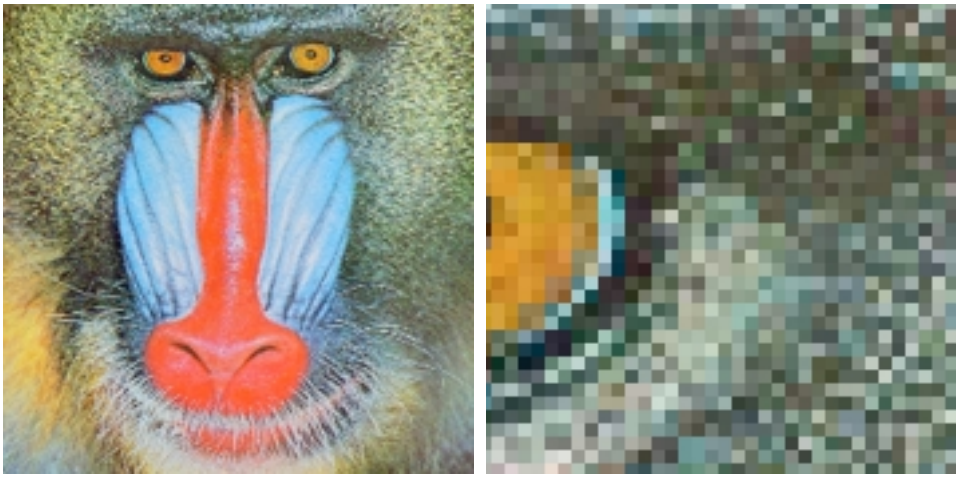


Figure G.2: Mandril and Detail.

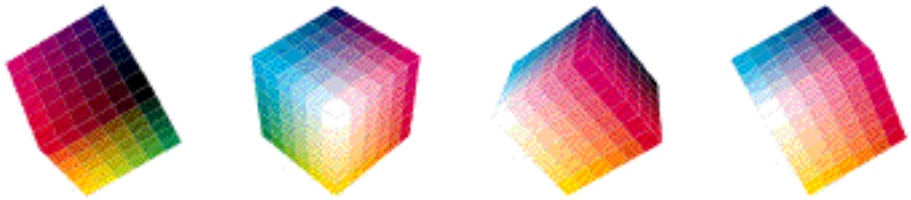


Figure G.3: The RGB Cube.

Figure G.4: Peppers and Detail.

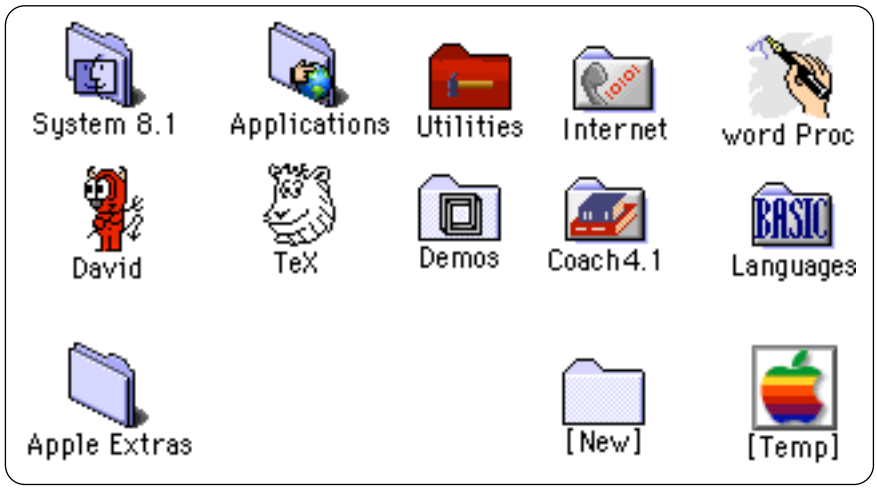


Figure G.5: A Discrete-Tone Image.

Figure G.6: A Discrete-Tone Image (Detail).

An ounce of image is worth a pound of performance.
Anonymous

H
Human Visual System

Image and video compression deal with large quantities of data, and one way to achieve good compression of images is to lose some of this data. The data loss must be done selectively and the guiding principle is to lose data for which the human visual system is not sensitive. This requires a detailed knowledge of color, its representations, and the way the eye perceives it. The hardware used to display color is also an important consideration in any computer application that uses color. The CRT (Section 6.1.1) is currently a common output device. It displays color by means of light emitting phosphors, but those cannot reproduce pure spectral colors. Color printers use various inks (in the form of liquid, wax, or powder) to mix colors on paper, and they suffer from the same shortcoming. Printers, CRTs, LCD displays, and all other output devices have their limitations and work best when using certain color representations. This appendix is an introduction to color, its representations, and the way the eye perceives color.

Some of the ideas and results presented here were developed by the International Committee on Illumination (Commission Internationale de l'Éclairage, or CIE), an international organization with a long history of illuminating various aspects of light and illumination.

H.1 Color and the Eye

We rarely find ourselves in complete darkness. In fact, most of the time we are flooded with light. This is why people get used to having light around and, as a result, they only rarely ask themselves *what is light*?

The best current explanation is that light can be understood (or interpreted) in two different ways, as a wave, or as a stream of particles. The latter interpretation sees light as a stream of massless particles, called *photons*, moving at a constant speed. The most important attribute of a photon is its frequency, since the photon's energy is proportional to it. Photons are useful in physics to explain a multitude of phenomena (such as the interaction of matter and light). In computer graphics,

the most important property of light is its color, which is why we use the former interpretation and we consider light to be a wave.

What "waves" (or undulates) in light is the electric and magnetic fields. When a region of space is flooded with light, those fields change periodically as we move from point to point in space. If we stay at one point, the fields also change periodically with time. Visible light is thus a (small) part of the electromagnetic spectrum (Figure H.1) which includes radio waves, ultraviolet, infrared, X-rays, and other types of radiation.

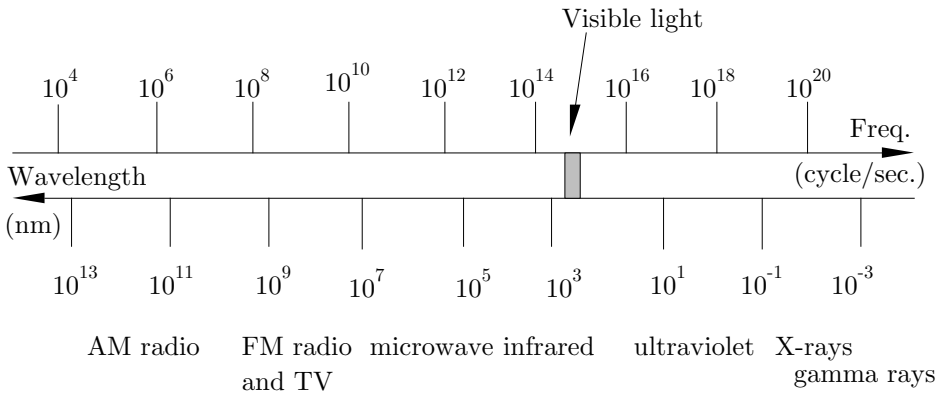


Figure H.1: The Electromagnetic Spectrum.

The most important properties of a wave are its frequency f, its wavelength λ, and its speed. Light moves, obviously, at the speed of light (in vacuum, it is $c \approx 3\times 10^{10}$cm/s). The three quantities are related by $f\lambda = c$. It is important to realize that the speed of light depends on the medium in which it moves. As light moves from vacuum to air to glass, it slows down (in glass, the speed of light is about $0.65c$). Its wavelength also decreases, but its frequency remains constant. Nevertheless, it is customary to relate colors to the wavelength and not to the frequency. Since visible light has very short wavelengths, a convenient unit is the nanometer (1 nm=10^{-9} m).

Visible light ranges from about 400 nm to about 700 nm and the color is determined by the wavelength. A wavelength of 420 nm, for example, corresponds to pure violet, while 620 nm is perceived by the eye as pure red. Using special lasers, it is possible to create pure (monochromatic) light consisting of one wavelength (Figure H.2a). Most light sources, however, output light that is a mixture of several (or even many) wavelengths, normally with one wavelength dominating (Figures H.2b and H.12).

The colors of the spectrum that are most visible to the human eye are (Figure H.7) violet (390–430), blue-violet (460–480), cyan, green (490–530), yellow (550–580), orange (590–640), and red (650–800).

White light is a mixture of all wavelengths, but what is gray light? It turns out that the wavelength of light is not its only important attribute. The *intensity*

is another attribute that should be considered. Gray light is a mixture of all wavelengths, but at a low intensity. When doing computer graphics, the main problem is how to specify the precise color of each pixel to the hardware. In many real-life situations, it is sufficient to say "I think I would like a navy blue suit," but computer hardware requires, of course, a much more precise specification. It, therefore, comes as a surprise to discover that color can be completely specified by just three parameters. Their meanings depend on the particular *color model* used. The RGB model is popular in computer graphics. In the printing industry, the CMYK model is normally used. Many artists use the HLS model. These models are discussed below.

Figure H.2b shows a simplified diagram of light smeared over the entire range of visible wavelengths, with a spike at about 620 nm (red), where it has a much higher intensity. This light can be described by specifying its *hue*, *saturation*, and *luminance*. The hue of the color is its dominant wavelength—620 nm in our example. The luminance is related to the intensity of the light. It is defined as the total power included in the spectrum and it is proportional to the area under the curve, which is $L = (700-400)A + B(D-A)$. The saturation is defined as the percentage of the luminance that resides in the dominant wavelength. In our case, it is $B(D-A)/L$. When there is no dominant wavelength (i.e., when $D = A$ or $B = 0$), the saturation is zero and the light is white. Large saturation means either large $D - A$ or small L. In either case, there is less white in the color and we see more of the red hue. Large saturation therefore corresponds to *pure color*.

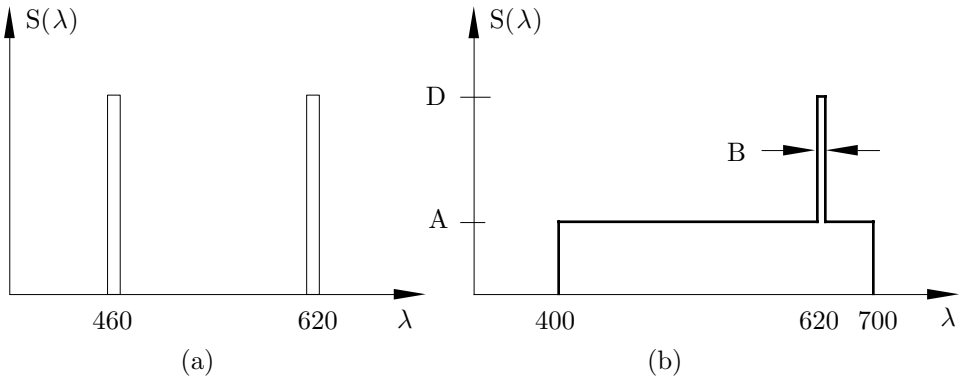


Figure H.2: (a) Pure Colors. (b) A Dominant Wavelength.

H.2 The HLS Color Model

This model was introduced in 1978 by Tektronics, aiming for an intuitive way to specify colors. The name stands for hue, lightness, and saturation. Lightness (or value) refers to the amount of black in the color. It controls the brightness of the color. Maximum lightness always creates white, regardless of the hue. Minimum lightness results in black. Saturation (or chroma) refers to the amount of white in the color. It controls the purity or vividness of the color. Low saturation means

more white in the color, resulting in a pastel color. Very low saturation results in a washed-out color. For a pure, vivid color, the saturation should be maximum. The achromatic colors black, white, and gray have zero saturation and differ in their values.

The HLS model is summarized in the double cone of Figure H.3. The vertical axis corresponds to L (lightness). It starts at zero (black) at the bottom and ends at one (white) at the top. The distance from the central axis corresponds to S (saturation). All points on the axis have zero saturation, so they correspond to shades of gray. Points farther away from the axis have more saturation; they correspond to more vivid colors. The H parameter (hue) corresponds to the hue of the color. This parameter is measured as an angle of rotation around the hexagon.

H.3 The HSV Color Model

The HSV model also uses hue, saturation, and value (lightness). It is summarized in the cone of Figure H.4. This is a single cone where the value V, which corresponds to lightness, goes from 0 (black) at the bottom, to 1 (white) at the flat top. The S and H parameters have the same meanings as in the double HLS cone.

> It's the weird color-scheme that freaks me. Every time you try to operate one of these weird black controls, which are labeled in black on a black background, a small black light lights up black to let you know you've done it!
>
> — Mark Wing-Davey (as Zaphod Beeblebrox) in *The Hitchhiker's Guide to the Galaxy* (1981).

H.4 The RGB Color Model

A *primary hue* is a color in a color model that cannot be made from the other colors used in that model. Primary hues serve as a basis for mixing and creating all other colors in the color model. Any color created by mixing *two* primary hues in a color model is a *secondary hue* in that model.

In the RGB color model, the three primaries are Red, Green, and Blue. They can be combined, two at a time to create the secondary hues. Magenta (pinkish hue) is (R+B), cyan (bluish hue) is (B+G), and yellow is (R+G). There are two reasons for using the red, green, and blue colors as primaries: (1) the cones in the eye are very sensitive to these colors (Figure H.7) and (2) adding red, green, and blue can produce many colors (although not all colors, see discussion of RGB color gamut on page 906).

The RGB color model is useful in computer graphics because of the way color CRTs work. They creates different colors by light emitted from phosphors of different types. The colors are then mixed in the eye of the observer, creating the impression of a perfect mixture. Assuming a range of $[0, 255]$ for each RGB color component, here are some examples of mixed colors:

$$\text{red} = (255, 0, 0),\ \text{magenta} = (255, 0, 255),\ \text{white} = (255, 255, 255),$$
$$50\%\ \text{gray} = (127, 127, 127),\ \text{light gray} = (25, 25, 25).$$

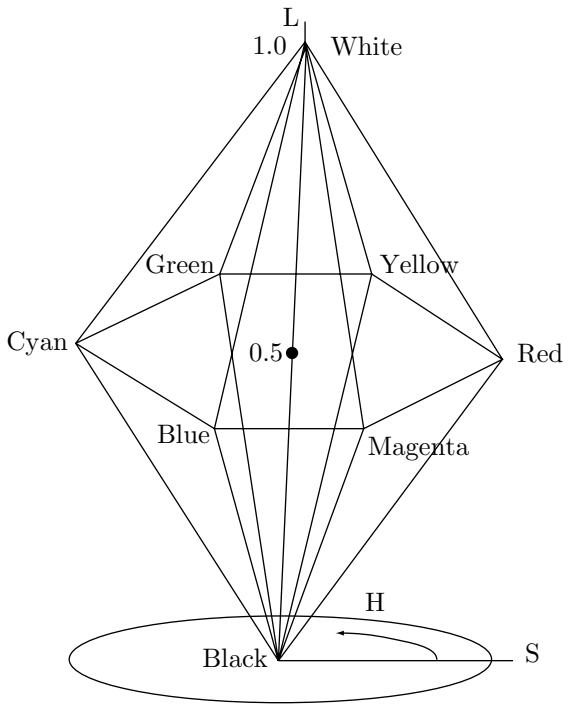


Figure H.3: The HLS Double Hexcone.

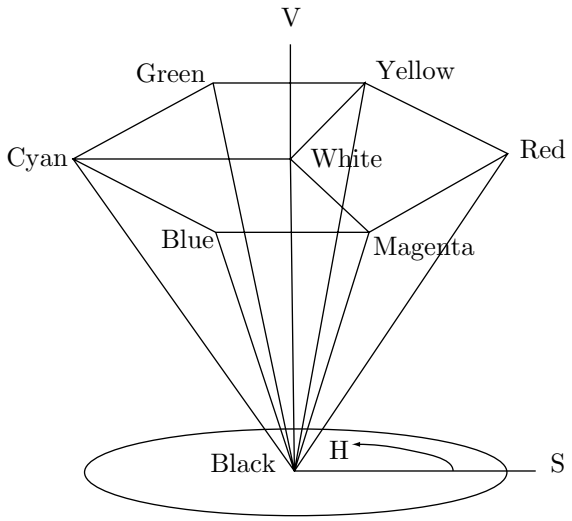


Figure H.4: The HSV Hexcone.

H.4.1 The RGB Cube

The *color gamut* of a color model is the entire range of colors that can be produced by the model. The color gamut of the RGB model can be summarized in a diagram shaped like a cube (Figures H.5 and G.3). Every point in the cube has three coordinates (r, g, b)—each in the range $[0, 1]$—which give the intensities of red, green, and blue of the point. Small values, close to $(0, 0, 0)$, mean a dark shade, whereas anything close to $(1, 1, 1)$ is very bright. Point $(1, 1, 1)$ itself corresponds to pure white. Point $(1, 0, 0)$ corresponds to red and point $(0, 1, 0)$ corresponds to green. Therefore, point $(1, 1, 0)$ describes a mixture of red and green, that is, yellow.

The RGB cube is useful because coordinates of points in it can readily be translated into values stored in the color lookup table of the computer.

◇ **Exercise H.1:** (Easy.) What colors correspond to the diagonal line connecting the black and white corners of the RGB cube?

H.5 Additive and Subtractive Colors

To create a mixture of several colors, we sometimes have to add them and sometimes have to subtract them. Imagine a white wall in a dark room. There is no light for the wall to reflect, so it looks black. We now shine red light on it. Since the wall is white (reflects all colors), it will reflect the red light and will look red. The same is true for green light. If we now shine both red and green colors on the wall, it will reflect both, which our brain interprets as yellow. We say that in this case the colors are added.

To understand the concept of subtracting colors, imagine a white sheet of paper in a bright environment. The paper reflects all colors, so it looks white. If we want to paint a certain spot red, we have to cover it with a chemical (red paint) that absorbs all colors except red. We say that the red paint *subtracts* the green and blue from the original white reflection, so the spot now reflects just red light. Similarly, if we want a yellow spot, we have to use yellow paint, which is a substance that absorbs blue and reflects red and green.

We conclude that in the case where we shine white light on a reflecting surface, we have to subtract colors in order to get the precise color that we want. In the case where we shine light of several colors on such a surface, we have to add colors to get any desired mixture.

> The various colours that may be obtained by the mixture of other colours, are innumerable. I only propose here to give the best and simplest modes of preparing those which are required for use. Compound colours, formed by the union of only two colours, are called by painters virgin tints. The smaller the number of colours of which any compound colour is composed, the purer and the richer it will be. They are prepared as follows:
>
> —Daniel Young, 1861, *Young's Translation of Scientific Secrets*.

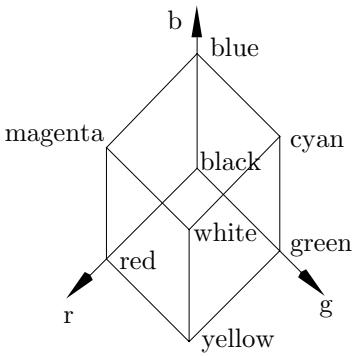


Figure H.5: The RGB Cube.

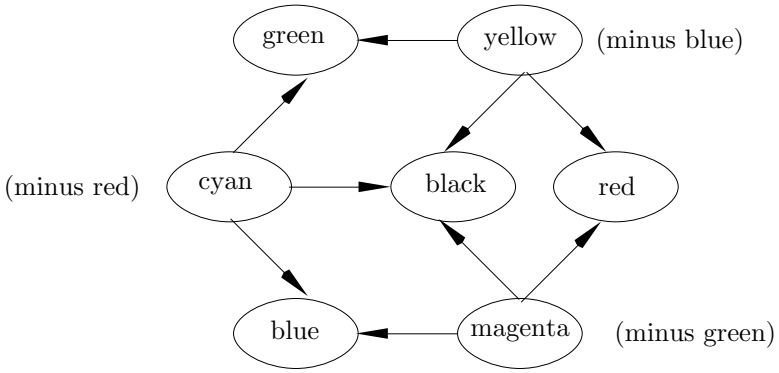


Figure H.6: RGB and CMYK Relationships.

For example, the human eye and its controlling software implicitly embody the false theory that yellow light consists of a mixture of red and green light (in the sense that yellow light gives us the same sensation as a mixture of red light and green light does). In reality, all three types of light have different frequencies and cannot be created by mixing light of other frequencies. The fact that a mixture of red and green light appears to us to be yellow light has nothing whatever to do with the properties of light, but is a property of our eyes. It is a result of a design compromise that occurred at some time during our distant ancestors' evolution.

—David Deutsch, *The Fabric Of Reality*.

H.5.1 Subtractive Color Models

There are two such models: painter's pigments and printing pigments.

Painter's Pigments: The primary colors of this color model are red, yellow, and blue. They were chosen because past artists believed that they were pure colors, containing no traces of any other colors. These three primaries can be mixed, two at a time, to produce the secondaries purple (R+B), green (B+Y), and orange (Y+R). Mixing equal amounts of all three primaries subtracts all colors and, hence, yields black.

Printing Pigments: This color model is also known as *process color* and is the result of development in color ink and printing processes. The three primaries are magenta, yellow, and cyan. The three secondaries are blue (M+C), red (M+Y), and green (C+Y). Mixing equal amounts of all three primaries should yield black, but, because of the properties of real inks, this black is normally not dark enough. In practice, true black is included in this model as an artificial fourth primary (also because black ink is cheaper). It is used when grayscale or black printing is required. The model is therefore sometimes called CMYK (K for black, to avoid confusion with blue) and color printing is known as the four-color process. Figure H.6 shows the relationships between the three CMY primaries and their secondaries.

Because of the particular primaries and secondaries of the CMY model there is a simple relationship between it and the RGB model. The relation is

$$(r, g, b) = (1, 1, 1) - (c, m, y).$$

This relationship shows that, for example, increasing the amount of cyan in a color, reduces the amount of red.

Traditional color printing uses color separation. The first step is to photograph the original image through different color filters. Each filter separates a primary color from the multi colored original. A blue filter separates the yellow parts of the original and creates a transparency with those parts printed in black and white. A red filter separates the cyan parts and a green filter separates the magenta parts. Another transparency is prepared, with the black parts. Each of the four transparencies is then converted to a halftone image (Section H.11) and the four images become masters for the final printing. They are placed in different stages of the printing machine and, as the paper moves through the machine, each stage adds halftone dots of colored ink to the paper. The result is a picture made of four halftone grids, each in one of the CMYK colors. The grids are not superimposed on the paper, but are printed offset. The eye sees dots colored in the four primaries, and the brain creates a mixed color that depends on the number of halftone dots of each primary.

When such a color print is held close to the eye, the individual dots in the four colors can be seen. Today, there are dye sublimation printers that mix wax of different dyes inside the printer to create a drop of wax of the right color which is then deposited on the paper. No halftoning is used. The result is a picture in vivid colors, but these printers are currently too expensive for home use. For more information on color printing and the CMYK model, see [Stone et al. 88].

To simplify the task of editors and graphics designers, several color standards have been developed. Instead of figuring out the ratios of CMYK, the graphics designer looks at a table that has many color samples, selects one, and uses its name to specify it to the printer. One such standard in common use today is the PANTONE matching system. It is described in [Pantone 91].

◇ **Exercise H.2:** A surface has a certain color because of its ability to absorb and reflect light. A surface that absorbs most of the light frequencies appears dark; a surface that reflects most frequencies appears bright. What colors are absorbed and what are reflected by a yellow surface?

> Blueness doth express trueness.
> — Ben Jonson

H.6 Complementary Colors

The concept of complementary colors is based on the idea that two colors appear psychologically harmonious if their mixture produces white. Imagine the entire color spectrum. The sum of all the colors produces white. If we subtract one color, say, blue, the sum of the remaining colors produces the complementary color, yellow. Blue and yellow are thus complementary colors (a dyad) in an additive color model. Other dyads are green and magenta, red and cyan, yellow-orange and cyan-blue, cyan-green and red-magenta, and yellow-green and blue-violet.

Subtractive Complementary Colors: These are based on the idea that two colors look harmonious if their mixture yields a shade of gray. The subtractive dyads are a yellow and violet, red and green, blue and orange, yellow-orange and blue-violet, blue-green and red-orange, and yellow-green and red-violet.

Complementary colors produce strong visual contrast, which creates a feeling of color vibrations or activity.

◇ **Exercise H.3:** Is there such a thing as additive color triads?

H.7 Human Vision

We see light that enters the eye and falls on the retina, where there are two types of photosensitive cells. They contain pigments that absorb visible light and hence give us the sense of vision. One type is the *rods*, which are numerous, are spread all over the retina, and respond only to light and dark. They are very sensitive and can respond to a single photon of light. There are about 110,000,000 to 125,000,000 rods in the eye [Osterberg 35]. The other type is the *cones*, located in one small area of the retina (the fovea). They number about 6,400,000, are sensitive to color, but require more intense light, in the order of hundreds of photons. Incidentally, the cones are very sensitive to red, green, and blue (Figure H.7), which is one reason why CRTs use these colors as primaries. There are three types of cones. The A cones are sensitive to red light. The B cones are sensitive to green light (a little more than the A cones), and the C cones are sensitive to blue light, but their sensitivity is about 1/30 that of the A or B cones.

The trichromatic theory of color vision says that light of a given wavelength stimulates each of the three types of cones in a different way, and it is the *ratio* of those stimuli, not their absolute values, that creates the sensation of color in the brain. As a result, any light intensity and background illumination that happen to produce the same ratio of stimuli will seem to have the same color. This theory explains why any color representation uses three parameters.

Each of the light sensors in the eye, rods and cones, sends a light sensation to the brain that is essentially a pixel, and the brain combines these pixels to a continuous image. The human eye is, thus, similar to a digital camera. Once this is realized, we naturally want to compare the resolution of the eye to that of a modern digital camera. Current digital cameras have from 300,000 sensors (for a cheap camera) to about six million sensors (for a high-quality one).

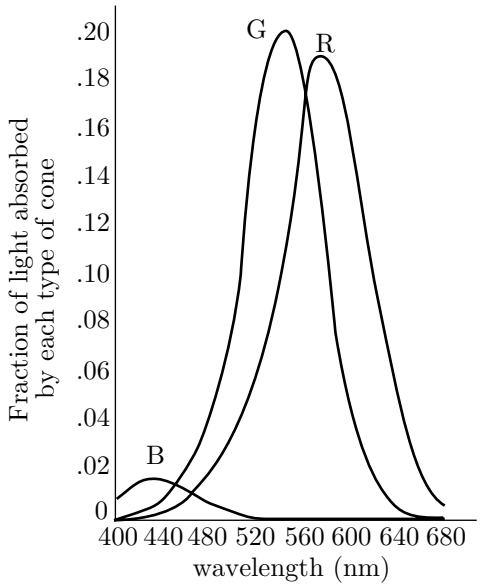


Figure H.7: Sensitivity of the Cones.

Thus, the eye features a much higher resolution, but its effective resolution is even higher when we consider that the eye can move and refocus itself about three to four times a second. This means that in a single second, the eye can sense and send to the brain about half a billion pixels. Assuming that our camera takes a snapshot once a second, the ratio of the resolutions is about 100.

Certain colors—such as red, orange, and yellow—are psychologically associated with heat. They are considered *warm* and cause a picture to appear larger and closer than it really is. Other colors—such as blue, violet, and green—are associated with cool things (air, sky, water, ice) and are therefore called *cool* colors. They cause a picture to look smaller and farther away.

Color

Colors, like features, follow the changes of the emotions.
— Pablo Picasso
There is no blue without yellow and without orange.
— Vincent Van Gogh

H.8 Luminance and Chrominance

The RGB and other color representations discussed earlier have one important drawback, they are not based on the human visual system. We know that the three types of cones in the eye are sensitive to green and (a little less) to red, and are less sensitive to blue. It is also known that the eye can best resolve small spatial details in the middle of the visible spectrum, which is green, and is poorest in this respect in the blue. This is important to an image compression algorithm, since it means that there is no need to preserve small details in blue regions of an image, but small green details are important and should be preserved.

It turns out that a color representation based on the two quantities *luminance* and *chrominance* can take advantage of these features more than the RGB representation. Luminance is a quantity that is closely related to the brightness of a light source as perceived by the eye. It is defined as proportional to the light energy emitted by the light source per unit area. Chrominance is related to the way the eye perceives hue and saturation.

One important aspect of the human visual system is that the perceptual response L^* of the eye to a light source with luminance Y is not linear but is proportional to $Y^{1/3}$. Specifically, the CIE has found that

$$L^* = \begin{cases} 116(Y/Y_n)^{1/3} - 16, & \text{if } Y/Y_n > 0.008856, \\ 903.3(Y/Y_n), & \text{otherwise,} \end{cases}$$

where the quantity L^* is called *lightness* and Y_n is the white reference luminance (see the discussion of illuminant white in Section H.10). The region $Y/Y_n \leq 0.008856$ corresponds to low values of Y (very dark) and is not important.

In addition to this feature (or problem) of the human visual system, the CRT itself is the source of another, physical, problem. It turns out that the intensity I of the light emitted by a CRT depends nonlinearly on the voltage V that is fed to the electron gun. The relation is $I = V^\gamma$, where the voltage V is assumed to be in the range $[0, 1]$, and γ (gamma) is a real number, typically in the range $[2, 3]$. Thus, the light intensity is a function somewhere between the square and the cube of the voltage.

It is one of life's many lucky coincidences that these two features cancel each other out. The two relations $L^* = 116(Y/Y_n)^{1/3} - 16$ and $I = V^\gamma$ imply that the lightness perceived by the eye is proportional to the voltage fed to the CRT if γ is 3 or close to 3. Figure H.8 shows the plots of x^2 and $x^{(1/2)}$ (and also x^3 and $x^{(1/3)}$). It is easy to see how adding two such plots produces a straight line (i.e., linear dependence).

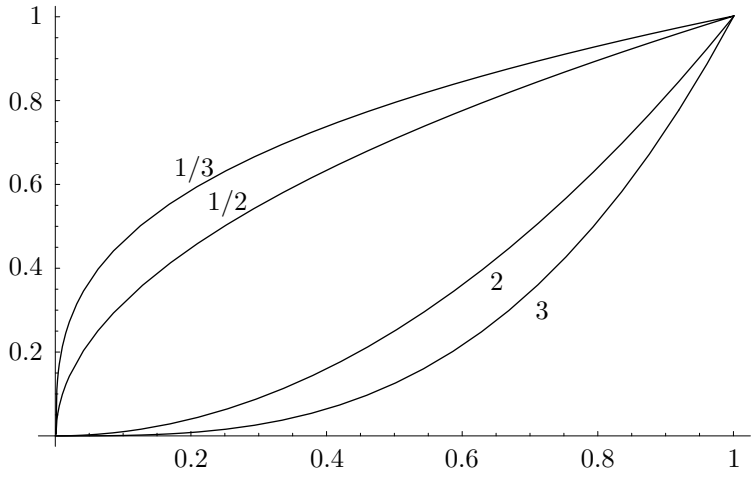


Figure H.8: Plots of x^γ and $x^{1/\gamma}$ For $\gamma = 2, 3$.

It should be stressed that this nonlinear behavior of the CRT is not caused by the phosphor but stems from the properties of the electron gun (the cathode). Since most CRTs are similar electrically, the value of gamma for a particular CRT depends mainly on its brightness and contrast settings. CRT users should therefore adjust their monitor's gamma with the intensity and contrast controls, using special test patterns. Figure H.9 is such a pattern where the goal is to find the bar where the brightnesses of the top and bottom parts match. This should be done from a distance of 6–10 feet. Some people find it more comfortable to look for the bar where the line in the middle disappears, or almost so. Once that bar is found, the gamma of the display is known and experts recommend to adjust it to between 2.2 and 2.5.

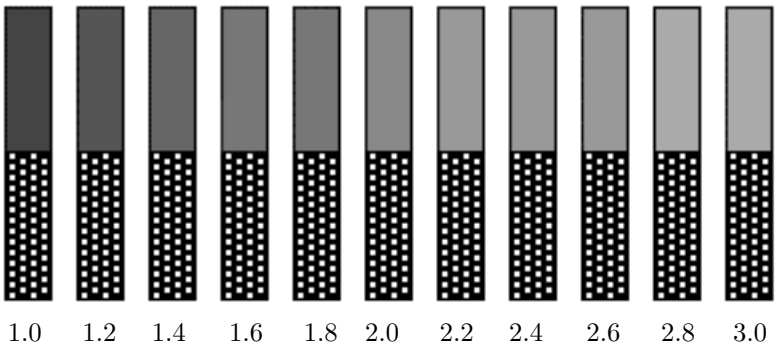


Figure H.9: A Gamma Test Pattern.

We turn now to the important (and confusing) concept of *gamma correction*. We have just stated that the lightness perceived by the eye is proportional to the

voltage fed to the CRT. Obviously we have to make sure that the voltage fed to the CRT is proportional to the light intensity seen by the video camera. The light sensor in the video camera (typically CCD) also has a gamma value and outputs a voltage V of the form $V = I^{\gamma}$ where I is the intensity "seen" by the sensor. This is why an extra circuit is built into the camera, that corrects for the nonlinear behavior of the sensor by changing $V \leftarrow V^{1/\gamma}$ and outputs a voltage V that is proportional to the intensity of the light falling on the sensor. Figure H.10 shows a block diagram of a video camera and a television receiver.

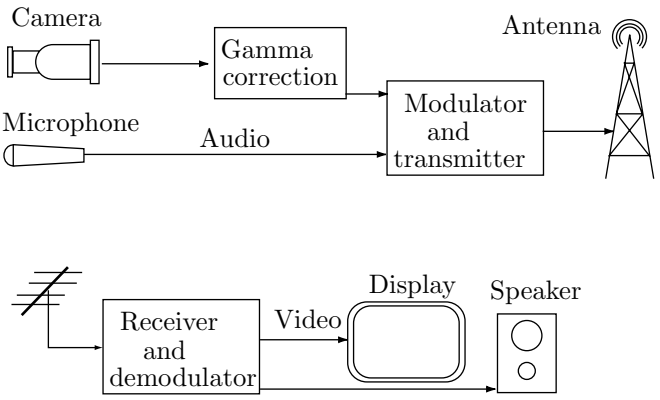


Figure H.10: Television Transmission With Gamma Correction.

In computer graphics, gamma correction is implemented in the color lookup table. To get an idea of how this is done, the reader should consider another feature of the human visual system. The eye-brain system can detect differences in light intensity, but this ability is not uniform from black to white. It is about 1% of the intensity itself. When looking at dark images, we can detect small variations in intensity. When seeing a bright image, we can only detect larger changes. With this in mind, imagine a color lookup table with 8-bit luminance values, where 0 represents black and 255 represents white. If two entries in the table have values of 25 and 26, then the difference between them is 1, but this one unit of difference is 4% of 25. Changing the value of a pixel from 25 to 26 would increase the brightness of the pixel by 4%, a large, noticeable intensity difference. Similarly, if the value of a pixel is changed from 100 to 101, then this one unit of change represents 1% of 100, and is, therefore, just barely noticeable to the eye. Changing a pixel from 200 to 201 increases the brightness of the pixel by 0.5% and would be unnoticeable. The value 201 is, in effect, wasted, and on the other hand we would like to have one or two values between 25 and 26. This suggests a gamma correction that assigns small intensity differences to small pixel values and large intensity differences to large pixel values. Ideally, adjacent values should correspond to luminance values that differ by about 1%, and one way of achieving this is to interpret pixel values as the logarithm of luminance.

Readers usually find gamma correction to be a confusing topic! Our aim, however, is to understand luminance and chrominance, and we start by stating

that the RGB components of a color are gamma corrected (in the video camera or in the lookup table), and the new, corrected values (which are also normalized in the range $[0,1]$) are denoted R', G', and B'. Each of these three components contributes differently to the brightness perceived by the eye (because of the three types of cones in the retina), so a new quantity Y', is defined as the weighted sum

$$Y' = 0.299R' + 0.587G' + 0.114B'. \tag{H.1}$$

Y' is called *luma*, but most authors call it luminance. Since the three corrected RGB components are normalized, Y' is also normalized in the range $[0,1]$. The two chrominance components U' and V' are defined as the differences $U' = B' - Y'$ and $V' = R' - Y'$. They can be interpreted as the presence or absence of blue and red in the color, respectively. Since R', G', B', and Y' all vary in the range $[0,1]$, it is easy to verify that U' varies in the range $[-0.886, +0.886]$ and V' varies in the range $[-0.701, +0.701]$, which suggests that they be normalized. Before we do that let's try to get a better understanding of the relations between the RGB and YUV color representations (the primes are omitted for convenience). Figure H.11 shows the RGB cube in dashed and the three planes $Y = 0.3$, $U = 0$, and $V = 0$. The latter two planes intersect at points that satisfy $R = G = B = Y$, so the intersection is the line that corresponds to all the gray colors.

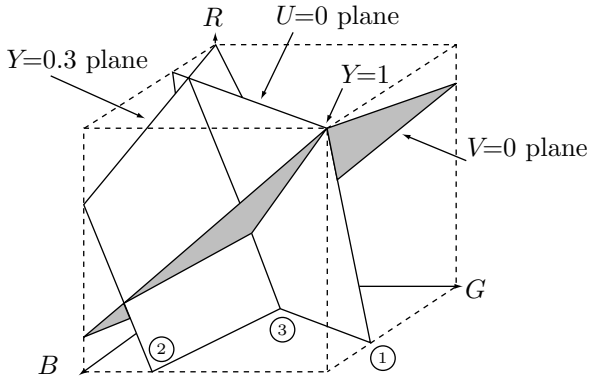


Figure H.11: Relationship Between the RGB and YUV Color Representations.

One point on the $U = 0$ plane is $(0, 0.755, .5)$. One point on the $V = 0$ plane is $(0.5, 0.5, 0.5)$. The reader should try to locate these points on the diagram.

◇ **Exercise H.4:** What are the RGB coordinates of the three points labeled ①, ②, and ③ in Figure H.11?

Next, we get to the chrominance components. Because of the ranges of variation of U' and V' they are easy to normalize. The two normalized quantities are denoted by Cb and Cr (for chrominance blue and chrominance red) and are defined by

$$Cb = (U'/2) + 0.5, \qquad Cr = (V'/1.6) + 0.5.$$

The $YCbCr$ color representation is commonly used in video and computer graphics, so image and video compression methods often assume that the images to be compressed are represented this way.

The three color representations RGB, YUV and $YCbCr$ are mathematically equivalent, since it is possible to transform each of them into the other ones. The advantage of $YCbCr$ is that the eye is more sensitive to small spatial luminance variations than to small color variations. A compression method that deals with an image in $YCbCr$ format should keep the Y component lossless (or close to lossless), but can afford to lose information in the chrominance components.

Transforming $R'G'B'$ into $Y'CbCr$ is done by

$$\begin{pmatrix} Y' \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} + \begin{pmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112 \\ 112 & -93.786 & -18.214 \end{pmatrix} \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}.$$

The inverse transformation is

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \begin{pmatrix} 0.00456621 & 0 & 0.00625893 \\ 0.00456621 & -0.00153632 & -0.00318811 \\ 0.00456621 & 0.00791061 & 0 \end{pmatrix} \left[\begin{pmatrix} Y' \\ Cb \\ Cr \end{pmatrix} - \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} \right].$$

Note: The luminance (or luma) defined by Equation (H.1) is part of the ITU-R recommendation BT.601-4. Strictly speaking it should be denoted Y_{601} and should be called nonlinear video luma. There is also a quantity called linear luma, which is defined by means of the linear (uncorrected) RGB. It is defined by the ITU-R recommendation 709 as

$$Y_{709} = 0.2125R + 0.7154G + 0.0721B.$$

H.9 Spectral Density

A laser is capable of emitting "pure" light, i.e., just one wavelength. Most light sources, however, emit "dirty" light that is a mixture of many wavelengths, normally with one dominating. For each light source, the graph of light intensity as a function of the wavelength λ is called the *spectral density* of the light source. Figure H.12 shows the spectral densities of several typical light sources.

These simple diagrams illustrate one problem in attempting to specify color systematically and unambiguously. Several different spectral densities may be perceived by us as identical. When the colors created by these spectral densities are placed side by side, we find it impossible to distinguish between them. The first step in solving the problem is color matching. Suppose that we use a color model defined by the three primaries $A(\lambda)$, $B(\lambda)$, and $C(\lambda)$ and we have a color described by the spectral density $S(\lambda)$. How can we express $S(\lambda)$ in terms of our three primaries? One way to do this is to shine a spot of $S(\lambda)$ on a white screen and, right next to it, a spot of light $P(\lambda) = \alpha A(\lambda) + \beta B(\lambda) + \gamma C(\lambda)$ created by mixing the three primaries (where $0 \le \alpha, \beta, \gamma \le 1$). Now change the amounts of α, β, and γ until a

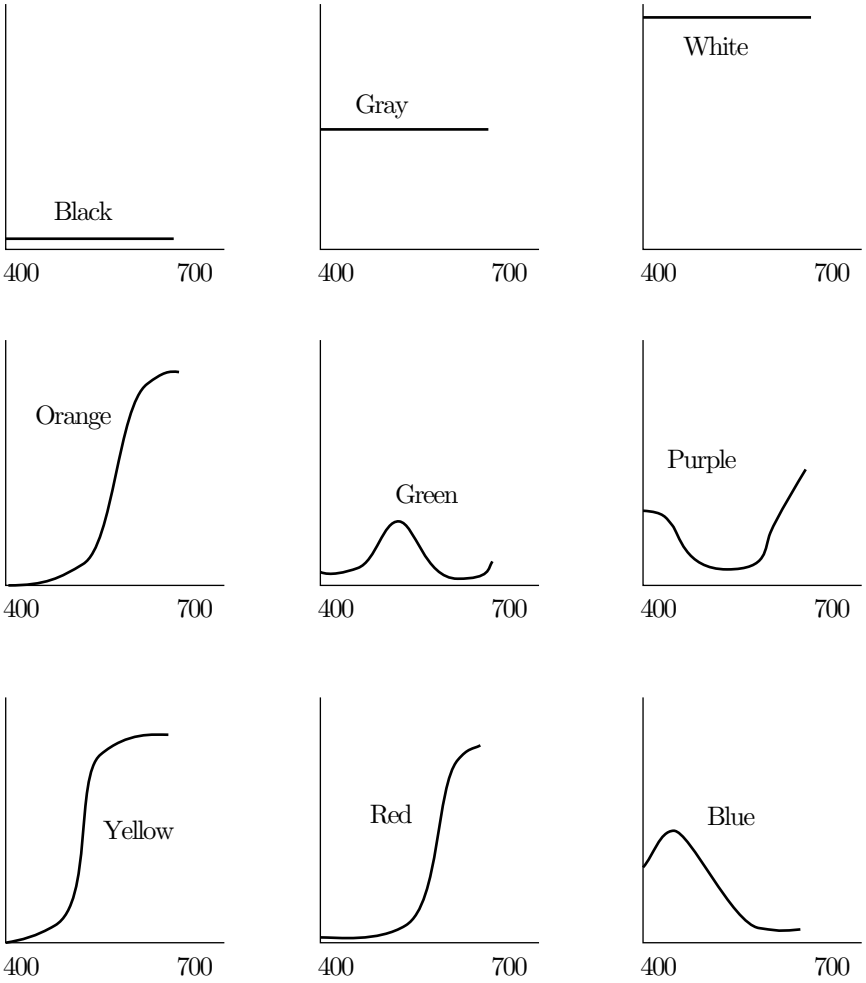


Figure H.12: Some Spectral Densities.

trained observer agrees that the spots are indistinguishable. We can now say that, in some sense, $S(\lambda)$ and $P(\lambda)$ are identical, and write $S = P$.

In what sense is the preceding true? It turns out that the above statement is meaningful because of a remarkable property of colors. Suppose that two spectral densities $S(\lambda)$ and $P(\lambda)$ have the same perceived color, so we write $S = P$. We now select another color Q and shine it on both spots S and P. We know from experience that the two new spots would also be indistinguishable. This means that we can use the symbol + for adding lights and we can describe the two spots by $S(\lambda)+Q(\lambda)$ and $P(\lambda)+Q(\lambda)$. In short, we can say "if $S = P$, then $S+Q = P+Q$." The same is true for changing intensities. If $S = P$, then $\alpha S = \alpha P$ for any intensity α. We, therefore, end up with a vector algebra for colors, where a color can be treated as a three-dimensional vector, with the usual vector operations.

Given the above, we can select a color model based on three primaries, A, B, and C, and can represent any color S as a linear combination of the primaries $S = \alpha A + \beta B + \gamma C$. We can say that the vector (α, β, γ) is the representation of S in the basis (A, B, C). Equivalently, we can say that S is represented as the point (α, β, γ) in the three-dimensional space defined by the vectors $A = (1, 0, 0)$, $B = (0, 1, 0)$, and $C = (0, 0, 1)$.

Since three-dimensional graphs are hard to draw on paper, we would like to artificially reduce the representation from three to two dimensions. This is done by realizing that the vector $(2\alpha, 2\beta, 2\gamma)$ represents the same color as (α, β, γ), only twice as bright. We, therefore, restrict ourselves to vectors (α, β, γ), where $\alpha + \beta + \gamma = 1$. These are vectors normalized to unit brightness. All vectors of unit brightness lie in the $\alpha + \beta + \gamma = 1$ plane and it is this plane (which, of course, is two-dimensional) that we plot on paper. We can specify a color by using two numbers, say, α and β and calculate γ as $1 - \alpha - \beta$.

For the RGB color model, we now select the pure spectral colors using trained human experts. The idea is to shine a spot of a pure color, say, 500 nm, on a screen and, right next to it, a spot that is a mixture $(r, g, b = 1 - r - g)$ of the three primaries of the RGB model. The values of r and g are varied until the observer judges the two spots to be indistinguishable. The point (r, g, b) is then plotted in the three-dimensional RGB color space. When all the pure colors have been plotted in this way, the points are connected to form a smooth curve, $\mathbf{P}(\lambda) = (r(\lambda), g(\lambda), b(\lambda))$. This is the *pure spectral color curve* of the RGB model (Figure H.13).

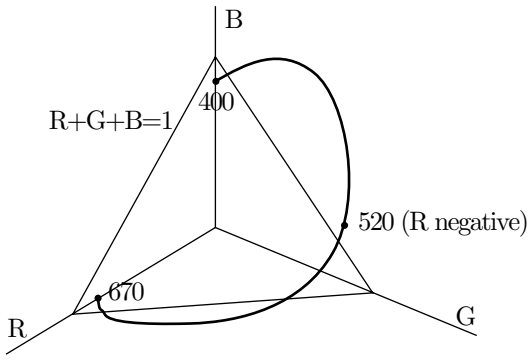


Figure H.13: Pure RGB Spectral Color Curve.

An important property of the curve is that some of r, g, and b may sometimes have to be negative. An example is $\lambda \approx 520\,\text{nm}$, where r turns out to be negative. What is the meaning of adding a negative quantity of green in a color defined by, for example, $S = 0.8R - 0.1G + 0.3B$? The way to understand this is to write the equation as $S + 0.1G = 0.8R + 0.3B$. It now means that color S cannot be constructed from the RGB primaries, but color $S + 0.1G$ can. The important conclusion is that not every color can be created in the RGB model! This is illustrated in Figure H.14. For some colors we can only create an approximation. This is true for all color models that can be created in practice.

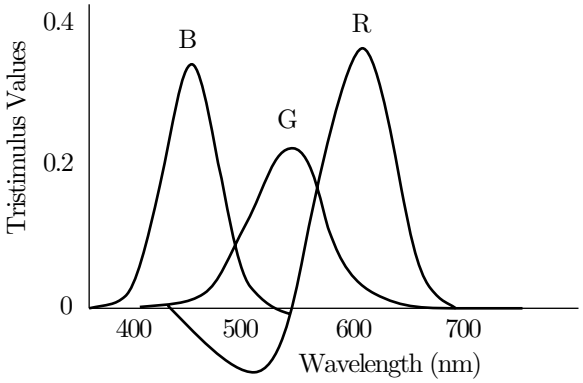


Figure H.14: RGB Color Combinations.

> Did I ever tell you my favorite color is blue?
> — Jürgen Prochnow (as Sutter Cane) in *In the Mouth of Madness*.

H.10 The CIE Standard

This standard was created in 1931 by the CIE. It is based on three carefully chosen artificial color primaries X, Y, and Z. They don't correspond to any real colors, but they have the important property that any real color can be represented as a linear combination $xX + yY + zZ$, where $x + y + z = 1$ and none of x, y, and z are negative (Figure H.15).

The plane $x + y + z = 1$ in the XYZ space is called the CIE chromaticity diagram (Figure H.16). The curve of pure spectral color in the CIE diagram covers all the pure colors, from 420 nm to 660 nm. It is shaped like a horseshoe.

Point $w = (0.310, 0.316)$ in the CIE diagram is special and is called "illuminant white." It is assumed to be the fully unsaturated white and is used in practice to match to colors that should be pure white.

◇ **Exercise H.5:** If illuminant white is pure white, why isn't it on the curve of pure spectral color in the CIE diagram?

The CIE diagram provides a standard for describing colors. There are instruments that, given a color sample, calculate the (x, y) coordinates of the color in the diagram. Also, given the CIE coordinates of a color, those instruments can generate a sample of the color. The diagram can also be used for useful color calculations. Here are some examples:

1. Given two points a and b in the diagram (Figure H.16), the line connecting them has the form $(1 - \alpha)a + \alpha b$ for $0 \leq \alpha \leq 1$. This line shows all the colors that can be created by adding varying amounts of colors a and b.

2. Imagine two points, such as c and d in Figure H.16. They are on the opposite sides of illuminant white and, therefore, correspond to complementary colors.

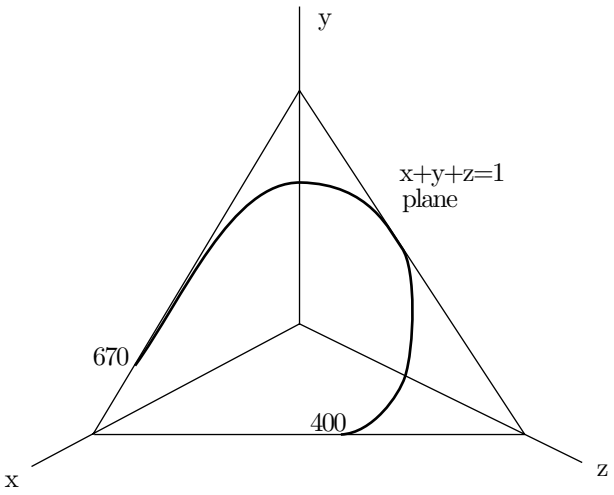


Figure H.15: Pure Spectral Color Curve.

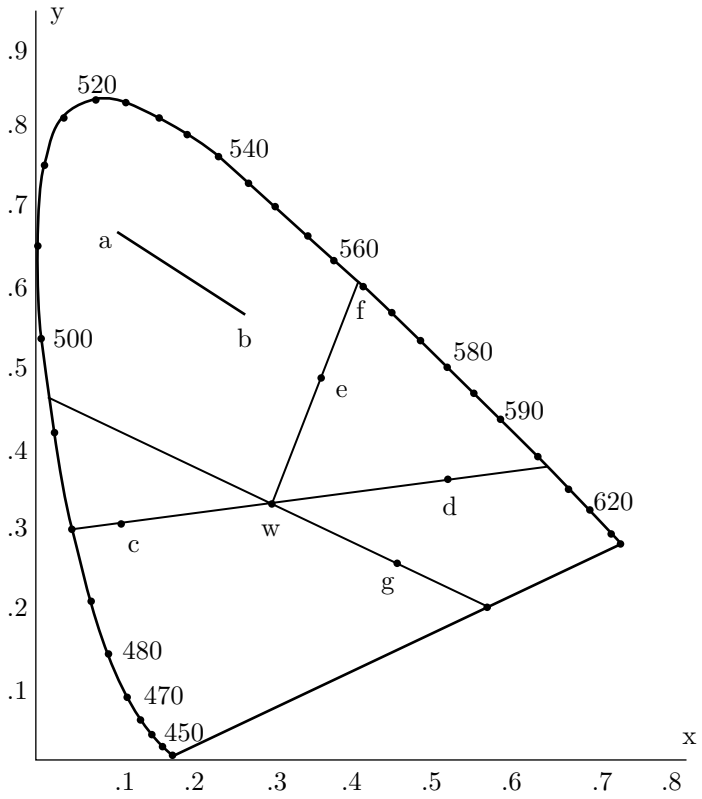


Figure H.16: The CIE Chromaticity Diagram.

◇ **Exercise H.6:** Why is that true?

3. The dominant wavelength of any color, such as e in Figure H.16 can be measured in the diagram. Just draw a straight line from illuminant white w to e and continue it until it intercepts the curve of pure spectral color. Then read the wavelength at the interception point f (564 nm in our example).

◇ **Exercise H.7:** How can the saturation of color e be calculated from the diagram?

◇ **Exercise H.8:** What is the dominant wavelength of point g in the CIE diagram?

> Have you lost your mind? What color is this bill?
> — Lori Petty (as Georgia "George" Sanders) in *Lush Life* (1996).

4. The *color gamut* of a device is the range of colors that can be displayed by the device. This can also be calculated with the CIE diagram. An RGB monitor, for example, can display combinations of red, green, and blue, but what colors are included in those combinations? To find the color gamut of an RGB monitor, we first have to find the locations of pure red, green, and blue in the diagram [these are points $(0.628, 0.330)$, $(0.285, 0.590)$, and $(0.1507, 0.060)$], then connect them with straight lines. The color gamut consists of all the colors within the resulting triangle. Each can be expressed as a linear combination of red, green, and blue, with non-negative coefficients (a convex combination).

Interestingly, because of the shape of the horseshoe, no three colors on or inside it can serve as ideal primaries. No matter what three points we select, some colors will be outside the triangle defined by them. This means that no set of three primaries can be used to create all the colors. The RGB set has the advantage that the triangle created by it is large, and thus contains many colors. The CMY triangle, for example, is much smaller by comparison. This is another reason for using red, green, and blue as the RGB primaries.

H.11 Halftoning

Color inkjet printers are common nowadays, but the inks are still expensive. Most laser printers are black and white. Halftoning is a method that makes it possible to print or display images with shades of gray on a black and white (i.e., bi-level) output device. The trade-off is loss of resolution. Instead of small, individual pixels, halftoning uses groups of pixels where only some of the pixels in a group are black. Halftoning is important, since it makes it possible to print pictures consisting of more than just black and white on a black and white printer. It is commonly used in newspapers and books. A classic reference is [Ulichney 87].

The human eye can resolve details as small as 1 minute of arc under normal light conditions. This is called the *visual acuity*. If we view a very small area from a normal viewing distance, our eyes cannot see the details in the area and end up integrating them, such that we only see an average intensity coming from the area. This property is called *spatial integration* and is very nicely demonstrated by Figure H.17. The figure consists of black circles and dots on a white background, but spatial integration creates the effect of a gray background.

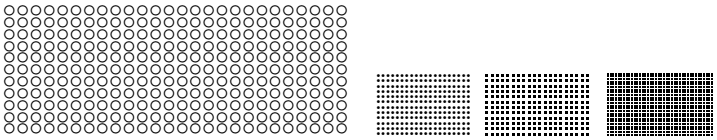


Figure H.17: Gray Backgrounds Obtained by Spatial Integration.

The principle of halftoning is to use groups of $n \times n$ pixels (with n usually in the range 2–4) and to set some of the pixels in a group to black. Depending on the black-to-white percentage of pixels in a group, the group appears to have a certain shade of gray. An $n \times n$ group of pixels contains n^2 pixels and can therefore provide $n^2 + 1$ levels of gray. The only practical problem is to find the best pattern for each of those levels. The $n^2 + 1$ pixel patterns selected must satisfy the following conditions:

1. Areas covered by copies of the same pattern should not show any textures.

2. Any pixel set to black for pattern k must also be black in all patterns of intensity levels $> k$. This is considered a good *growth sequence* and it minimizes the differences between patterns of successive intensities.

3. The patterns should grow from the center of the $n \times n$ area, to create the effect of a growing dot.

4. All the black pixels of a pattern must be adjacent to each other. This property is called *clustered dot halftoning* and is important if the output is intended for a printer (laser printers cannot always fully reproduce small isolated dots). If the output is intended for CRT only, then *dispersed dot halftoning* can be used, where the black pixels of a pattern are not adjacent.

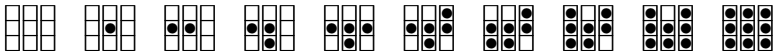
As a simple example of condition 1, a pattern such as

must be avoided, since large areas with level-3 groups would produce long horizontal lines. Other patterns may result in similar, annoying textures. With a 2×2 group, such effects may be impossible to avoid. The best that can be done is to use the patterns  .

A 3×3 group provides for more possibilities. The 10 patterns below ($= 3^2 + 1$) are the best ones possible (reflections and rotations of these patterns are considered identical) and usually avoid the problem above. They were produced by the matrix

$$\begin{bmatrix} 7 & 9 & 5 \\ 2 & 1 & 4 \\ 6 & 3 & 8 \end{bmatrix}$$

using the following rule: To create a group with intensity n, only cells with values $\leq n$ in the above matrix should be black.



The halftone method is not limited to a monochromatic display. Imagine a display with four levels of gray per pixel (2-bit pixels). Each pixel is either black or can be in one of three other levels. A 2×2 group consists of four pixels, each of which can be in one of three levels of gray or in black. The total number of levels is therefore $4\times3+1=13$. One possible set of the 13 patterns is shown below.

Bibliography

Ulichney, Robert (1987) *Digital Halftoning*, Cambridge, MA, MIT Press.

H.12 Dithering

The downside of halftoning is loss of resolution. It is also possible to display continuous-tone images (i.e., images with different shades of gray) on a bi-level device *without* losing resolution. Such methods are sometimes called *dithering* and their trade-off is loss of image detail. If the device resolution is high enough and if the image is watched from a suitable distance, then our eyes perceive an image in grayscale, but with fewer details than in the original.

The dithering problem can be phrased as follows: given an $m\times n$ array A of pixels in grayscale, calculate an array B of the same size with zeros and ones (corresponding to white and black pixels, respectively) such that for every pixel $B[i,j]$ the average value of the pixel and a group of its near neighbors will approximately equal the normalized value of $A[i,j]$. (Assume that pixel $A[i,j]$ has an integer value I in the interval $[0,a]$; then its normalized value is the fraction I/a. It is in the interval $[0,1]$.)

The simplest dithering method uses a threshold and the single test: Set $B[i,j]$ to white (0) if $A[i,j]$ is bright enough (i.e., less than the value of the threshold); otherwise, set $B[i,j]$ to black (1). This method is fast and simple, but generates very poor results, as the next example shows, so it is never used in practice. As an example, imagine a human head. The hair is generally darker than the face below it, so the simple threshold method may quantize the entire hair area to black and the entire face area to white, a very poor, unrecognizable, and unacceptable result. (It should be noted, however, that some images are instantly recognizable even in just black and white, as Figure H.18 aptly demonstrates.) This method can be improved a little by using a different, random threshold for each pixel, but even this produces low-quality results.

Four approaches to dithering, namely ordered dither, constrained average dithering, diffusion dither, and dot diffusion, are presented in this section. Another approach, called ARIES, is discussed in [Roetling 76] and [Roetling 77].

H.12.1 Ordered Dither

The principle of this method is to paint a pixel $B[i,j]$ black or leave it white, depending on the intensity of pixel $A[i,j]$ **and** on its position in the picture [i.e., on its coordinates (i,j)]. If $A[i,j]$ is a dark shade of gray, then $B[i,j]$ should ideally be dark thus, it is painted black most of the time, but sometimes it is left white. The decision whether to paint it black or white depends on its coordinates i and j. The opposite is true for a bright pixel. This method is described in [Jarvis et al. 76].

Figure H.18: A Familiar Black-and-White Image.

The giant panda resembles a bear, although anatomically it is more like a raccoon. It lives in the high bamboo forests of central China. Its body is mostly white, with black limbs, ears, and eye patches. Adults weigh 200 to 300 lb (90 to 140 kg). Low birth rates and human encroachment on its habitat have seriously endangered this species.

The method starts with an $m \times n$ dithering matrix D_{mn} which is used to determine the color (black = 1 or white = 0) of all the B pixels. In the example below we assume that the A pixels have 16 gray levels, with 0 as white and 15 as black. The dithering matrices for $n = 2$ and $n = 4$ are shown below. The idea in these matrices is to minimize the amount of texture in areas with a uniform gray level.

$$D_{22} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}, \qquad D_{44} = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}.$$

The rule is as follows: Given a pixel $A[x, y]$ calculate $i = x \bmod m$, $j = y \bmod n$, then select black (i.e., set $B[x, y]$ to 1) if $A[x, y] \geq D_{mn}[i, j]$ and select white otherwise.

To see how the dithering matrix is used, imagine a large, uniform area in the image A where all the pixels have a gray level of 4. Since 4 is the fifth of 16 levels, we would like to end up with 5/16 of the pixels in the area set to black (ideally they should be randomly distributed in this area). When a row of pixels is scanned in this area, y is incremented, but x does not change. Since i depends on x, and j depends on y, the pixels scanned are compared to one of the rows of matrix D_{44}. If this happens to be the first row, then we end up with the sequence 10101010 . . .

in bitmap B.

When the next line of pixels is scanned, x and, as a result, i have been incremented, so we look at the next row of D_{44}, that produces the pattern 01000100... in B. The final result is an area in B that looks like

$$\begin{array}{l} 10101010\ldots \\ 01000100\ldots \\ 10101010\ldots \\ 00000000\ldots \end{array}$$

Ten out of the 32 pixels are black, but $10/32 = 5/16$. The black pixels are not randomly distributed in the area, but their distribution does not create annoying patterns either.

◇ **Exercise H.9:** Assume that image A has three large uniform areas with gray levels 0, 1, and 15 and calculate the pixels that go into bitmap B for these areas.

Ordered dither is easy to understand if we visualize copies of the dither matrix laid next to each other on top of the bitmap. Figure H.19 shows a 6×12 bitmap with six copies of a 4×4 dither matrix laid on top of it. The threshold for dithering a pixel $A[i,j]$ is that element of the dither matrix that happens to lie on top of $A[i,j]$.

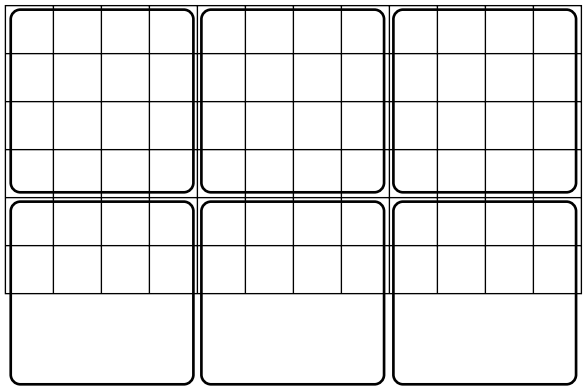


Figure H.19: Ordered Dither.

Matrix D_{44} above was created from D_{22} by the recursive rule

$$D_{nn} = \begin{pmatrix} 4D_{n/2,n/2} & 4D_{n/2,n/2} + 2U_{n/2,n/2} \\ 4D_{n/2,n/2} + 3U_{n/2,n/2} & 4D_{n/2,n/2} + U_{n/2,n/2}, \end{pmatrix}, \tag{H.2}$$

where U_{nn} is an $n\times n$ matrix with all ones. Other matrices are easy to generate with this rule.

◇ **Exercise H.10:** Use the rule of Equation (H.2) to construct D_{88}.

The basic rule of ordered dither can be generalized as follows: Given a pixel $A[x, y]$, calculate $i = x \bmod m$ and $j = y \bmod n$, then select black (i.e., assign $B[x, y] \leftarrow 1$) if $Ave[x, y] \geq D_{mn}[i, j]$, where $Ave[x, y]$ is the average of the 3×3 group of pixels centered on $A[x, y]$. This is computationally more intensive, but tends to produce better results in most cases, since it considers the average brightness of a group of pixels.

Ordered dither is a simple, fast method, but it tends to create images that have been described by various people as "computerized," "cold," or "artificial." The reason for that is probably the recursive nature of the dithering matrix.

H.12.2 Constrained Average Dithering

In cases where high speed is not important, this method [Jarvis and Roberts 76] gives good results, although it involves more computations than ordered dither. The idea is to compute, for each pixel $A[i, j]$, the average $\bar{A}[i, j]$ of the pixel and its eight near neighbors. The pixel is then compared to a threshold of the form

$$\gamma + \left(1 - \frac{2\gamma}{M}\right) \bar{A}[i, j],$$

where γ is a user-selected parameter and M is the maximum value of $A[i, j]$. Notice that the threshold can have values in the range $[\gamma, M - \gamma]$. The final step is to compare $A[i, j]$ to the threshold and set $B[i, j]$ to 1 if $A[i, j] \geq$ threshold and to 0 otherwise.

The main advantage of this method is edge enhancement. An example is Figure H.20 that shows a 6×8 bitmap A, where pixels have 4-bit values. Most pixels have a value of 0, but the bitmap also contains a thick slanted line indicated in the figure. It separates the 0 (white) pixels in the upper-left and bottom-right corners from the darker pixels in the middle.

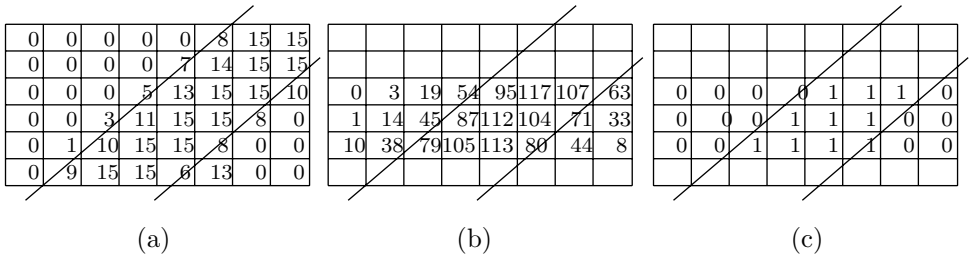


Figure H.20: Constrained Average Dithering.

Figure H.20a shows how the pixels around the line have values approaching the maximum (which is 15). Figure H.20b shows (for some pixels) the average of the pixel and its eight near neighbors (the averages are shown as integers, so an average of 54 really indicates 54/9). The result of comparing these averages to the

threshold (which in this example is 75) is shown in Figure H.20c. It is easy to see how the thick line is sharply defined in the bi-level image.

> So I'm a ditherer? Well, I'm jolly well going to dither, then!
> — Roland Young (as Cosmo Topper) in *Topper* (1937).

H.12.3 Diffusion Dither

Imagine a photograph, rich in color, being digitized by a scanner that can distinguish millions of colors. The result may be an image file where each pixel $A[i,j]$ is represented by, say, 24 bits. The pixel may have one of 2^{24} colors. Now, imagine that we want to display this file on a computer that can only display 256 colors simultaneously on the screen. A good example is a computer using a color lookup table whose size is 3×256 bytes.

We begin by loading a palette of 256 colors into the lookup table. Each pixel $A[i,j]$ of the original image will now have to be displayed on the screen as a pixel $B[i,j]$ in 1 of the 256 palette colors. The diagram below shows a pixel $A[i,j]$ with original color $(255,52,80)$. If we decide to assign pixel $B[i,j]$ the palette color $(207,62,86)$, then we are left with a difference of $A[i,j]-B[i,j]=(48,-10,-6)$. This difference is called the *color error* of the pixel.

$$\boxed{\begin{array}{l}\text{R=255}\\ \text{G=52}\\ \text{B=80}\end{array}} - \boxed{\begin{array}{l}\text{R=207}\\ \text{G=62}\\ \text{B=86}\end{array}} = \boxed{\begin{array}{l}\text{R=48}\\ \text{G=}-10\\ \text{B=}-6\end{array}}$$

Large color errors degrade the quality of the displayed image, so an algorithm is needed to minimize the total color error of the image. Diffusion dither does this by distributing the color errors among all the pixels such that the total color error for the entire image is zero (or very close to zero).

The algorithm is very simple. Pixels $A[i,j]$ are scanned line by line from top to bottom. In each line, they are scanned from left to right. For each pixel, the algorithm performs the following:

1. Pick up the palette color that is nearest the original pixel's color. This palette color is stored in the destination bitmap $B[i,j]$.
2. Calculate the color error $A[i,j]-B[i,j]$ for the pixel.
3. Distribute this error to four of $A[i,j]$'s nearest neighbors that haven't been scanned yet (the one on the right and the three ones centered below) according to the *Floyd-Steinberg filter* [Floyd and Steinberg 75] (where the X represents the current pixel):

	X	7/16
3/16	5/16	1/16

Consider the example of Figure H.21a. The current pixel is $(255,52,80)$ and we (arbitrarily) assume that the nearest palette color is $(207,62,86)$. The color error is $(48,-10,-6)$ and is distributed as shown in Figure H.21c. The five nearest

neighbors are assigned new colors as shown in Figure H.21b. The algorithm is shown in Figure H.22a where the weights $p1$, $p2$, $p3$, and $p4$ can be assigned either the Floyd-Steinberg values 7/16, 3/16, 5/16, and 1/16 or any other values.

The total color error may not be exactly zero because the method does not work well for the leftmost column and for the bottom row of pixels. However, the results can be quite good if the palette colors are carefully selected.

This method can easily be applied to the case of a monochromatic display (or any *bi-level* output device), as shown by the pseudo-code of Figure H.22b, where $p1$, $p2$, $p3$, and $p4$ are the four error diffusion parameters. They can be the ones already given (i.e., 7/16, 3/16, 5/16, and 1/16) or different ones, but their sum should be 1.

◇ **Exercise H.11:** Consider an all-gray image, where $A[i, j]$ is a real number in the range $[0, 1]$ and it equals 0.5 for all pixels. What image B would be generated by diffusion dither in this case?

◇ **Exercise H.12:** Imagine a grayscale image consisting of a single row of pixels where pixels have real values in the range $[0, 1]$. The value of each pixel p is compared to the threshold value of 0.5 and the error is propagated to the neighbor on the right. Show the result of dithering a row of pixels all with values 0.5.

Error diffusion can also be used for color printing. A typical low-end ink-jet color printer has four ink cartridges for cyan, magenta, yellow, and black ink. The printer places dots of ink on the page such that each dot has one of the four colors. If a certain area on the page should have color L, where L isn't any of CMYK, then L can be simulated by dithering. This is done by printing adjacent dots in the area with CMYK colors such that the eye (which integrates colors in a small area) will perceive color L. This can be done with error diffusion where the palette consists of the four colors cyan $(255, 0, 0)$, magenta $(0, 255, 0)$, yellow $(0, 0, 255)$, and black $(255, 255, 255)$ and the error for a pixel is the difference between the pixel color and the nearest palette color.

A slightly simpler version of error diffusion is the *minimized average error* method. The errors are not propagated but rather calculated and saved in a separate table. When a pixel $A[x, y]$ is examined, the error table is used to look up the errors $E[x + i, y + j]$ already computed for some previously seen neighbors $A[i, j]$ of the pixel. Pixel $B[x, y]$ is assigned a value of 0 or 1 depending on the corrected intensity:

$$A[x, y] + \frac{1}{\sum_{ij} \alpha_{ij}} \sum_{ij} \alpha_{ij} \mathrm{E}[x + i, y + j].$$

The new error, $E[x, y] = A[x, y] - B[x, y]$, is then added to the error table to be used for future pixels. The quantities α_{ij} are weights assigned to the near neighbors of $A[x, y]$. They can be assigned in many different ways, but they should assign more weight to nearby neighbors, so the following is a typical example:

$$\alpha = \begin{pmatrix} 1 & 3 & 5 & 3 & 1 \\ 3 & 5 & 7 & 5 & 3 \\ 5 & 7 & x & - & - \end{pmatrix},$$

Before	R=255 G=52 B=80	R=178 G=20 B=60
R=192 G=45 B=75	R=250 G=49 B=83	R=191 G=31 B=72

(a)

After	R=207 G=62 B=86	R=199 G=16 B=57
R=201 G=43 B=74	R=265 G=46 B=81	R=194 G=30 B=72

(b)

$\frac{7}{16} \times 48 = 21, \quad \frac{1}{16} \times 48 = 3, \quad \frac{5}{16} \times 48 = 15, \quad \frac{3}{16} \times 48 = 9,$
$\frac{7}{16} \times (-10) = -4, \quad \frac{1}{16} \times (-10) = -1, \quad \frac{5}{16} \times (-10) = -3, \quad \frac{3}{16} \times (-10) = -2,$
$\frac{7}{16} \times (-6) = -3, \quad \frac{1}{16} \times (-6) = 0, \quad \frac{5}{16} \times (-6) = -2, \quad \frac{3}{16} \times (-6) = -1.$

(c)

Figure H.21: Diffusion Dither.

```
for i := 1 to m do
 for j := 1 to n do
  begin
  B[i,j]:=SearchPalette(A[i,j]);

  err := A[i,j] - B[i,j];
  A[i,j+1] := A[i,j+1] + err * p1;
  A[i+1,j-1] := A[i+1,j-1] + err * p2;
  A[i+1,j] := A[i+1,j] + err * p3;
  A[i+1,j+1] := A[i+1,j+1] + err * p4;
  end.
```

(a)

```
for i := 1 to m do
 for j := 1 to n do
  begin
  if A[i,j] < 0.5 then B[i,j] := 0
    else B[i,j] := 1;
  err := A[i,j] - B[i,j];
  A[i,j+1] := A[i,j+1] + err * p1;
  A[i+1,j-1] := A[i+1,j-1] + err * p2;
  A[i+1,j] := A[i+1,j] + err * p3;
  A[i+1,j+1] := A[i+1,j+1] + err * p4;
  end.
```

(b)

Figure H.22: Diffusion Dither Algorithm. (a) For Color. (b) For Bi-level.

where x is the current pixel $A[x, y]$ and the weights are defined for some previously seen neighbors above and to the left of $A[x, y]$. If the weights add up to 1, then the corrected intensity above is simplified and becomes

$$A[x, y] + \sum_{ij} \alpha_{ij} E[x + i, y + j].$$

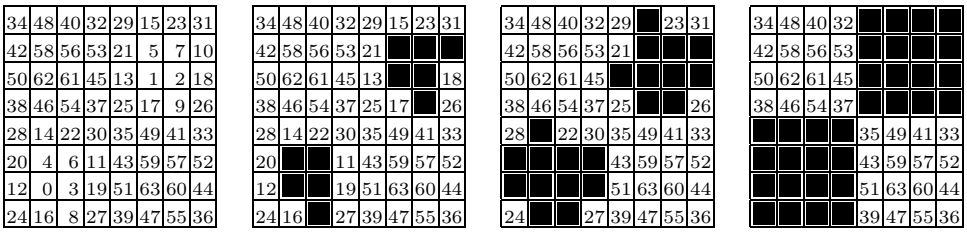
Floyd-Steinberg error diffusion generally produces better results than ordered dither, but has two drawbacks, namely it is serial in nature and it sometimes produces annoying "ghosts." Diffusion dither is serial, since the near neighbors of $B[i, j]$ cannot be calculated until the calculation of $B[i, j]$ is complete and the error $A[i, j] - B[i, j]$ has been distributed to the four near neighbors of $A[i, j]$. To understand why ghosts are created, imagine a dark area positioned above a bright area (for example, a dark sky above a bright sea). When the algorithm works on the last dark A pixels, a lot of error is distributed below, to the first bright A pixels (and also to the right). When the algorithm gets to the first bright pixels, they have collected so much error from above that they may no longer be bright, creating perhaps several rows of dark B pixels. It has been found experimentally that ghosts can be "exorcised" by scaling the A pixels before the algorithm starts. For example, each $A[i, j]$ pixel can be replaced by $0.1 + 0.8A[i, j]$, which "softens" the differences in brightness (the contrast) between the dark and bright pixels, thereby reducing the ghosts. This solution also changes all the pixel intensities, but the eye is less sensitive to absolute intensities than to changes in contrast, so changing intensities may be acceptable in many practical situations.

H.12.4 Dot Diffusion

This section is based on [Knuth 87], a very detailed article containing thorough analysis and actual images dithered using several different methods. The dot diffusion algorithm is somewhat similar to diffusion dither, it also produces good quality, sharp bi-level images, but it is not serial in nature and may be easier to implement on a parallel computer.

We start with the 8×8 *class matrix* of Figure H.23a. The way this matrix was constructed will be discussed later. For now, we simply consider it a permutation of the integers $(0, 1, \ldots, 63)$, which we call *classes*. The class number k of a pixel $A[i, j]$ is found at position (i, j) of the class matrix. The main algorithm is shown in Figure H.24.

The algorithm computes all the pixels of class 0 first, then those of class 1, and so on. Procedure `Distribute` is called for every class k and diffuses the error err to those near neighbors of $A[i, j]$ whose class numbers exceed k. The algorithm distinguishes between the four orthogonal neighbors and the four diagonal neighbors of $A[i, j]$. If a neighbor is $A[u, v]$, then the former type satisfies $(u-i)^2+(v-j)^2 = 1$, while the latter type is identified by $(u - i)^2 + (v - j)^2 = 2$. It is reasonable to distribute more of the error to the orthogonal neighbors than to the diagonal ones, so a possible weight function is $\text{weight}(x, y) = 3 - x^2 - y^2$. For an orthogonal neighbor, either $(u - i)$ or $(v - j)$ equals 1, so $\text{weight}(u - i, v - j) = 2$, while for a diagonal neighbor, both $(u - i)$ and $(v - j)$ equal 1, so $\text{weight}(u - i, v - j) = 1$. Procedure `Distribute` is listed in pseudo-code in Figure H.24.



(a) (b) (c) (d)

Figure H.23: 8×8 Matrices for Dot Diffusion.

```
for k := 0 to 63 do
 for all (i,j) of class k do
  begin
  if A[i,j] < .5 then B[i,j] := 0 else B[i,j] := 1;
  err := A[i,j] − B[i,j];
  Distribute(err,i,j,k);
  end.

procedure Distribute(err,i,j,k);
 w := 0;
 for all neighbors A[u,v] of A[i,j] do
  if class(u,v) > k then w := w+weight(u − i,v − j);
 if w > 0 then for all neighbors A[u,v] of A[i,j] do
  if class(u,v) > k then A[u,v] := A[u,v] + err×weight(u − i,v − j)/w;
end;
```

Figure H.24: The Dot Diffusion Algorithm.

Once the coordinates (i,j) of a pixel $A[i,j]$ are known, the class matrix gives the pixel's class number k that is independent of the color (or brightness) of the pixel. The class matrix also gives the classes of the eight near neighbors of $A[i,j]$, so those neighbors whose classes exceed k can be selected and linked in a list. It is a good idea to construct those lists once and for all, since this speeds up the algorithm considerably.

It remains to show how the class matrix, Figure H.23a, was constructed. The main consideration is the relative positioning of small and large classes. Imagine a large class surrounded, in the class matrix, by smaller classes. An example is class 63, which is surrounded by the "lower classes" 43, 59, 57, 51, 60, 39, 47, and 55. A little thinking shows that as the algorithm iterates toward 63, more and more error is absorbed into pixels that belong to this class, regardless of their brightness. A large class surrounded by lower classes is therefore undesirable and may be called a "baron." The class matrix of Figure H.23a has just two barons. Similarly, "near-baron" positions, which have only one higher-class neighbor, are undesirable and should be avoided. Our class matrix has just two of them.

◇ **Exercise H.13:** What are the barons and near-barons of our class matrix?

◇ **Exercise H.14:** Consider an all-gray image where $A[i, j] = 0.5$ for all pixels. What image B would be generated by dot diffusion in this case?

Another important consideration is the positions of consecutive classes in the class matrix. Figure H.23b,c,d show the class matrix after 10, 21, and 32 of its lowest classes have been blackened. It is easy to see how the black areas form 45° grids that grow and eventually form a 2×2 checkerboard. This helps create diagonal, rather than rectilinear dot patterns in the bi-level array B, and we know from experience that such patterns are less noticeable to the eye. Figure H.25a shows a class matrix with just one baron and one near-baron, but it is easy to see how the lower classes are mostly concentrated at the bottom-left corner of the matrix.

25	21	13	39	47	57	53	45
48	32	29	43	55	63	61	56
40	30	35	51	59	62	60	52
36	14	22	26	46	54	58	44
16	6	10	18	38	42	50	24
8	0	2	7	15	31	34	20
4	1	3	11	23	33	28	12
17	9	5	19	27	49	41	37

(a)

14	13	1	2
4	6	11	9
0	3	15	12
10	8	5	7

(b)

Figure H.25: Two Class Matrices for Dot Diffusion.

A close examination of the class matrix shows that the class numbers in positions (i, j) and $(i, j + 4)$ always add up to 63. This means that the grid pattern of $63 - k$ white pixels after k steps is identical to the grid pattern of $63 - k$ black pixels after $63 - k$ steps, shifted right four positions. This relation between the dot pattern and the diffusion pattern is the reason for the name *dot diffusion*.

◇ **Exercise H.15:** Figure H.25b shows a 4×4 class matrix. Identify the barons, near-barons, and grid patterns.

Experiments with the four methods described in this section seem to indicate that the dot diffusion method produces best results for printing because it tends to generate contiguous areas of black pixels, rather than "checkerboard" areas of alternating black and white. Modern laser and ink-jet printers have resolutions of 600 dpi or more, but they generally cannot produce a high-quality checkerboard of 300 black and 300 white alternating pixels per inch.

Bibliography

Floyd, R., and L. Steinberg (1975) "An Adaptive Algorithm for Spatial Gray Scale," in *Society for Information Display 1975 Symposium Digest of Technical Papers*, p. 36.

Jarvis, J. F., C. N. Judice, and W. H. Ninke (1976) "A Survey of Techniques for the Image Display of Continuous Tone Pictures on Bilevel Displays" *Computer Graphics and Image Processing* **5**(1):13–40.

Jarvis, J. F. and C. S. Roberts (1976) "A New Technique for Displaying Continuous Tone Images on a Bilevel Display" *IEEE Transactions on Communications* **24**(8):891–898, August.

Knuth, Donald E., (1987) "Digital Halftones by Dot Diffusion," *ACM Transactions on Graphics* **6**(4):245–273.

Roetling, P. G. (1976) "Halftone Method with Edge Enhancement and Moiré Suppression," *Journal of the Optical Society of America*, **66**:985–989.

Roetling, P. G. (1977) "Binary Approximation of Continuous Tone Images," *Photography Science and Engineering*, **21**:60–65.

> While wondering what he should do in this emergency he came upon a girl sitting by the roadside. She wore a costume that struck the boy as being remarkably brilliant: her silken waist being of emerald green and her skirt of four distinct colors – blue in front, yellow at the left side, red at the back and purple at the right side. Fastening the waist in front were four buttons—the top one blue, the next yellow, a third red and the last purple.
>
> L. Frank Baum, *The Marvelous Land of Oz*

I
Introductory Mathematics

> Do not worry too much about your difficulties in mathematics, I can assure you that mine are still greater.
>
> — Albert Einstein.

I.1 Useful Sums

1. The sum of a geometric series is

$$\sum_{i=0}^{n} a^i = \begin{cases} 0, & \text{if } a = 0, \\ n+1, & \text{if } a = 1, \\ \frac{1-a^{n+1}}{1-a}, & \text{otherwise.} \end{cases} \tag{I.1}$$

A simple corollary is

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \quad \text{for } |a| < 1. \tag{I.2}$$

Differentiating Equation (I.2) yields

$$\sum_{i=0}^{\infty} i a^i = \frac{a}{(1-a)^2} \quad \text{for } |a| < 1.$$

2. The binomial theorem

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}.$$

3. Series expansion of an exponential (Taylor series)

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

4. The sum of the first n integers

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

5. The sum of the first n integers squared

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

6. Sum of powers of 2

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$

I.2 Matrices

A matrix $\mathbf{T}$ is a rectangular array of numbers, where each element a_{ij} is identified by its row and column. Matrix $\mathbf{T}_1$ below is "generic," with m rows and n columns. Notice how elements a_{ii} constitute the main diagonal of the matrix. Matrix $\mathbf{T}_2$ is diagonal ($a_{ij} = 0$ for $i \neq j$), matrix $\mathbf{T}_3$, is symmetric ($a_{ij} = a_{ji}$), and $\mathbf{T}_4$ is an identity matrix.

$$\mathbf{T}_1 = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \quad \mathbf{T}_2 = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{pmatrix},$$

$$\mathbf{T}_3 = \begin{pmatrix} 33 & -17 & 201 & -5 \\ -17 & 66 & 26 & -68 \\ 201 & 26 & 21 & -9 \\ -5 & -68 & -9 & 0 \end{pmatrix}, \quad \mathbf{T}_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The transpose of matrix $\mathbf{A}$ (denoted by $\mathbf{A}^T$) is obtained from $\mathbf{A}$ by reflecting all the elements with respect to the main diagonal. A symmetric matrix equals its transpose.

> All problems in computer graphics can be solved with a matrix inversion.
>
> James F. Blinn, 1993.

I.2.1 Matrix Operations

The rule for matrix addition/subtraction is $c_{ij} = a_{ij} \pm b_{ij}$, where $\mathbf{C} = \mathbf{A} \pm \mathbf{B}$. The rule for matrix multiplication is slightly more complex: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Each element of $\mathbf{C}$ is the *dot product* of a row of $\mathbf{A}$ and a column of $\mathbf{B}$. In the dot product, corresponding elements from $\mathbf{A}$ and $\mathbf{B}$ are multiplied, and the products summed. In order for the multiplication to be well defined, each row of $\mathbf{A}$ must have the same size as a column of $\mathbf{B}$. Matrices $\mathbf{A}$ and $\mathbf{B}$ can therefore be multiplied only if the number of columns of $\mathbf{A}$ equals the number of rows of $\mathbf{B}$. Note that matrix multiplication is not commutative, i.e., $\mathbf{AB} \neq \mathbf{BA}$ in general.

An example of matrix multiplication is the product of the 1×3 and 3×1 matrices

$$(1, -1, 5) \begin{pmatrix} 4 \\ -2 \\ 3 \end{pmatrix},$$

which yields the 1×1 matrix 21.

Tensor products. This is a special case of matrix multiplication. If $\mathbf{A}$ is a column vector and $\mathbf{B}$ is a row vector (each with n elements), then their tensor product $\mathbf{C}$ is defined by $\mathbf{C}_{ij} = \mathbf{A}_i \mathbf{B}_j$. Example:

$$\begin{pmatrix} 4 \\ -2 \\ 3 \end{pmatrix} (1, -1, 5) = \begin{pmatrix} 4 & -4 & 20 \\ -2 & 2 & -10 \\ 3 & -3 & 15 \end{pmatrix}.$$

A square matrix has a determinant, denoted by either "$\det \mathbf{A}$" or $|\mathbf{A}|$, that is a number. The determinant of the 2×2 matrix $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ is defined as $ad - bc$. The determinant of a larger matrix can be calculated by the rule (note the alternating signs):

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}.$$

Matrix division is not defined, but certain matrices have an *inverse*. The inverse of $\mathbf{A}$ is denoted $\mathbf{A^{-1}}$, and has the property that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, where $\mathbf{I}$ is the *identity matrix* (with ones in the diagonal and zeros elsewhere). The inverse of a matrix is used, e.g., to solve systems of linear algebraic equations. Such a system can be denoted $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is the matrix of coefficients, $\mathbf{x}$ is the column of unknowns, and $\mathbf{b}$ is the column of the right-hand side coefficients. The solution is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Example: The following system of three equations with three unknowns x, y, and z

$$\begin{aligned} x - y &= 1, \\ -x + y &= 2, \\ 25x + 2y + z &= 3, \end{aligned} \tag{I.3}$$

can be written

$$\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 25 & 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

The inverse of the 3×3 transformation matrix (used in Section 4.32.1)

$$\mathbf{T} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ m & n & 1 \end{pmatrix} \quad \text{is} \quad \mathbf{T}^{-1} = \frac{1}{ad-bc} \begin{pmatrix} d & -b & 0 \\ -c & a & 0 \\ cn-dm & bm-an & 1 \end{pmatrix}. \tag{I.4}$$

In general, however, the calculation of the inverse is not trivial and can be found in any text on Linear Algebra, and also in [Press et al. 88]. Page 227 has an interesting example of the inverse of a matrix.

Here is a summary of the properties of matrix operations:

$$\begin{gathered} \mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}, \quad \mathbf{A}+(\mathbf{B}+\mathbf{C})=(\mathbf{A}+\mathbf{B})+\mathbf{C}, \\ k(\mathbf{A}+\mathbf{B})=k\mathbf{A}+k\mathbf{B}, \quad (k+m)\mathbf{A}=k\mathbf{A}+m\mathbf{A}, \quad k(m\mathbf{A})=(km)\mathbf{A}=m(k\mathbf{A}), \\ \mathbf{A}(\mathbf{BC})=(\mathbf{AB})\mathbf{C}, \quad \mathbf{A}(\mathbf{B}+\mathbf{C})=\mathbf{AB}+\mathbf{AC}, \\ (\mathbf{A}+\mathbf{B})\mathbf{C}=\mathbf{AB}+\mathbf{AC}, \quad \mathbf{A}(k\mathbf{B})=k(\mathbf{AB})=(k\mathbf{A})\mathbf{B}, \\ (\mathbf{A}+\mathbf{B})^T=\mathbf{A}^T+\mathbf{B}^T, \quad (k\mathbf{A})^T=k^T\mathbf{A}^T, \quad (\mathbf{AB})^T=\mathbf{B}^T\mathbf{A}^T. \end{gathered}$$

Information on the history of matrices and determinants can be found at URL `http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/`, file `Matrices_and_determinants.html`.

◇ **Exercise I.1:** Add, subtract, and multiply the two matrices

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix}.$$

◇ **Exercise I.2:** Calculate the inverse of

$$\mathbf{T} = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 25 & 2 & 1 \end{pmatrix}.$$

A matrix is *orthogonal* if the dot product of any two different rows is zero (and the same for columns. A matrix is *orthonormal* if it is orthogonal and the dot product of a row with itself is one (and the same for columns). Imagine a square matrix $\mathbf{A}$. When it is transposed, its rows and columns change roles. A general element (i,j) of the product $\mathbf{AA}^T$ is thus the dot product of row i of $\mathbf{A}$ and row j of the same $\mathbf{A}$. Therefore, if $\mathbf{A}$ is orthonormal, then $\mathbf{AA}^T$ is the identity matrix $\mathbf{I}$. However, the product $\mathbf{BB}^{-1}$ for any matrix $\mathbf{B}$ is $\mathbf{I}$ (if $\mathbf{B}$ has an inverse), so we conclude that the transpose $\mathbf{A}^T$ of an orthonormal matrix $\mathbf{A}$ equals its inverse $\mathbf{A}^{-1}$.

The opposite is also true. If $\mathbf{A}^T = \mathbf{A}^{-1}$ for some matrix $\mathbf{A}$, then $\mathbf{A}$ is orthonormal. It can be shown that an orthonormal matrix is always a rotation matrix, and that any rotation matrix [Equation (4.48)] is orthonormal.

Eigenvalues and eigenvectors (from the German word for "own") are useful mathematical quantities associated with matrices. They are defined as follows: If $\mathbf{A}$ is an $n\times n$ matrix and if there exist vectors $\mathbf{x}$ and scalars λ such that $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ [or $(\mathbf{A} - \lambda I)\mathbf{x} = 0$], then λ is called an *eigenvalue* of $\mathbf{A}$, and $\mathbf{x}$ is the *eigenvector* associated with λ. The eigenvectors of a symmetric matrix are orthogonal.

In principle, calculating the eigenvalues of an $n\times n$ matrix involves solving an nth-degree polynomial equation. Therefore, for $n \geq 5$, the results cannot in general be expressed purely in terms of explicit radicals. Even for the simple matrix

$$\begin{pmatrix} a & b \\ -b & 2a \end{pmatrix},$$

the eigenvalues have the two complicated expressions

$$\frac{1}{2}\left(3a - \sqrt{a^2 - 4b^2}\right), \text{ and } \frac{1}{2}\left(3a + \sqrt{a^2 - 4b^2}\right).$$

This is why mathematical software is used in practice to obtain approximate values (real and complex) of the eigenvalues and eigenvectors of a given matrix.

Eigenvalues and eigenvectors are mentioned in Section 4.4.8.

Bibliography

Press, W. H., B. P. Flannery, et al. (1988) *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press.
(Also available on-line from `http://www.nr.com/`.)

> We [he and Halmos] share a philosophy about linear algebra: we think basis-free, we write basis-free, but when the chips are down we close the office door and compute with matrices like fury.
>
> Irving Kaplansky.

I.3 Trigonometric Identities

Many of the identities listed here can be derived with the help of DeMoivre's theorem [Equation (Ans.3)].

Basic Identities

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha}, \quad \cot\alpha = \frac{\cos\alpha}{\sin\alpha} = \frac{1}{\tan\alpha}, \quad \csc\alpha = \frac{1}{\sin\alpha}, \quad \sec\alpha = \frac{1}{\cos\alpha}.$$

$$\sin(-\alpha) = -\sin\alpha, \quad \cos(-\alpha) = \cos\alpha, \quad \tan(-\alpha) = -\tan\alpha.$$

$$\sin^2\alpha + \cos^2\alpha = 1, \quad \tan^2\alpha + 1 = \sec^2\alpha, \quad \cot^2\alpha + 1 = \csc^2\alpha.$$

Sum and Difference Identities

$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta, \quad \sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta,$$
$$\tan(\alpha \pm \beta) = \frac{\tan\alpha \pm \tan\beta}{1 \mp \tan\alpha\tan\beta}.$$

Cofunction Identities

$$\sin(\pi/2 - \alpha) = \cos\alpha, \quad \cos(\pi/2 - \alpha) = \sin\alpha, \quad \tan(\pi/2 - \alpha) = \cot\alpha.$$

Multiple Angle and Half Angle Identities

$$\cos 2\alpha = \cos^2\alpha - \sin^2\alpha = 1 - 2\sin^2\alpha = 2\cos^2\alpha - 1, \quad \sin 2\alpha = 2\sin\alpha\cos\alpha,$$
$$\tan 2\alpha = \frac{2\tan\alpha}{1 - \tan^2\alpha}.$$
$$\cos(\alpha/2) = \pm\sqrt{(1+\cos\alpha)/2}, \quad \sin(\alpha/2) = \pm\sqrt{(1+\cos\alpha)/2}.$$
$$\tan(\alpha/2) = \pm\sqrt{\frac{1-\cos\alpha}{1+\cos\alpha}} = \frac{\sin\alpha}{1+\cos\alpha} = \frac{1-\cos\alpha}{\sin\alpha}.$$

Sum and Product Identities

$$\sin\alpha + \sin\beta = 2\sin\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right),$$
$$\sin\alpha - \sin\beta = 2\cos\left(\frac{\alpha+\beta}{2}\right)\sin\left(\frac{\alpha-\beta}{2}\right).$$
$$\cos\alpha + \cos\beta = 2\cos\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right),$$
$$\cos\alpha - \cos\beta = -2\sin\left(\frac{\alpha+\beta}{2}\right)\sin\left(\frac{\alpha-\beta}{2}\right).$$
$$\sin\alpha\cos\beta = \frac{1}{2}\left[\sin(\alpha+\beta) + \sin(\alpha-\beta)\right],$$
$$\cos\alpha\sin\beta = \frac{1}{2}\left[\sin(\alpha+\beta) - \sin(\alpha-\beta)\right].$$
$$\cos\alpha\cos\beta = \frac{1}{2}\left[\cos(\alpha+\beta) + \cos(\alpha-\beta)\right],$$
$$\sin\alpha\sin\beta = -\frac{1}{2}\left[\cos(\alpha+\beta) - \cos(\alpha-\beta)\right].$$

Note that the line above also implies

$$\cos^2\alpha = \frac{1}{2}\left(\cos(2\alpha) + 1\right), \quad \sin^2\alpha = \frac{1}{2}\left(1 - \cos(2\alpha)\right).$$

Laws of Sines and Cosines: any triangle with sides a, b, c and angles α, β, γ satisfies the law of sines $a/\sin\alpha = b/\sin\beta = c/\sin\gamma$ and the law of cosines

$$a^2 = b^2 + c^2 - 2bc\cos\alpha, \quad b^2 = a^2 + c^2 - 2ac\cos\beta, \quad c^2 = a^2 + b^2 - 2ab\cos\gamma.$$

Mathematics is the only universal language there is, senator!
— Jodie Foster (as Ellie Arroway) in *Contact* (1977).

I.4 Vector Algebra

A vector is a mathematical entity with two attributes, direction and magnitude (notice that a vector has no spatial position). The magnitude of vector $\mathbf{P} = (x, y, z)$ (also called its *absolute value*) is $|\mathbf{P}| = \sqrt{x^2 + y^2 + z^2}$. The direction of a vector can be expressed by the cosines of the angles it makes with the coordinate axes $x/|\mathbf{P}|$, $y/|\mathbf{P}|$, and $z/|\mathbf{P}|$. Note that the vector $(x/|\mathbf{P}|, y/|\mathbf{P}|, z/|\mathbf{P}|)$ has a magnitude of 1 (it is a *unit vector*).

The three unit vectors in the directions of the coordinate axes are traditionally denoted $\mathbf{i} = (1, 0, 0)$, $\mathbf{j} = (0, 1, 0)$, and $\mathbf{k} = (0, 0, 1)$.

I.4.1 Operations on Vectors

Vector addition is defined by adding the individual elements of the vectors being added. Thus, $\mathbf{P} + \mathbf{Q} = (P_x, P_y, P_z) + (Q_x, Q_y, Q_z) = (P_x + Q_x, P_y + Q_y, P_z + Q_z)$. This operation is both commutative ($\mathbf{P}+\mathbf{Q} = \mathbf{Q}+\mathbf{P}$) and associative $\mathbf{P}+(\mathbf{Q}+\mathbf{T}) = (\mathbf{P} + \mathbf{Q}) + \mathbf{T}$. Subtraction of vectors $\mathbf{P} - \mathbf{Q}$ is done similarly and results in the vector from $\mathbf{Q}$ to $\mathbf{P}$.

Vectors can be multiplied in three different ways:

1. The multiplication of a scalar by a vector is defined by $\alpha\mathbf{P} = (\alpha x, \alpha y, \alpha z)$. It changes the magnitude of the vector (by a factor α), but not its direction. This operation is distributive with respect to vector addition or subtraction, $\alpha(\mathbf{P} \pm \mathbf{Q}) = \alpha\mathbf{P} \pm \alpha\mathbf{Q}$.

2. The dot product of two vectors is denoted by $\mathbf{P} \bullet \mathbf{Q}$ and is defined as the scalar

$$(P_x, P_y, P_z)(Q_x, Q_y, Q_z)^T = \mathbf{P}\mathbf{Q}^T = P_xQ_x + P_yQ_y + P_zQ_z.$$

This also equals $|\mathbf{P}|\,|\mathbf{Q}|\cos\theta$, where θ is the angle between the vectors. The dot product of perpendicular vectors (also called *orthogonal vectors*) is thus zero. The dot product is commutative, $\mathbf{P} \bullet \mathbf{Q} = \mathbf{Q} \bullet \mathbf{P}$ and is also distributive with respect to vector addition or subtraction $\mathbf{P} \bullet (\mathbf{Q} \pm \mathbf{T}) = \mathbf{P} \bullet \mathbf{Q} \pm \mathbf{P} \bullet \mathbf{T}$.

The triple product $(\mathbf{P} \bullet \mathbf{Q})\mathbf{R}$ is sometimes useful. It can be represented as

$$\begin{aligned}
&(\mathbf{P} \bullet \mathbf{Q})\mathbf{R} \\
&= (P_xQ_x + P_yQ_y + P_zQ_z)(R_x, R_y, R_z) \\
&= \big((P_xQ_x + P_yQ_y + P_zQ_z)R_x, (P_xQ_x + P_yQ_y + P_zQ_z)R_y, (P_xQ_x + P_yQ_y + P_zQ_z)\big)R_z \\
&= (Q_x, Q_y, Q_z) \begin{pmatrix} P_xR_x & P_yR_x & P_zR_x \\ P_xR_y & P_yR_y & P_zR_y \\ P_xR_z & P_yR_z & P_zR_z \end{pmatrix} \\
&= \mathbf{Q}(\mathbf{P}\mathbf{R}), \qquad \text{(I.5)}
\end{aligned}$$

where the notation $(\mathbf{P}\mathbf{R})$ stands for the 3×3 matrix above.

3. The cross product of two vectors (also called the *vector product*) is denoted $\mathbf{P}\times\mathbf{Q}$ and is defined as the vector

$$(P_2Q_3 - P_3Q_2, -P_1Q_3 + P_3Q_1, P_1Q_2 - P_2Q_1). \tag{I.6}$$

It is easy to show that $\mathbf{P}\times\mathbf{Q}$ is perpendicular to both $\mathbf{P}$ and $\mathbf{Q}$.

◇ **Exercise I.3:** Prove it!

The following expressions show how $\mathbf{P}\times\mathbf{Q}$ can be expressed by means of a determinant.

$$\begin{aligned}\mathbf{P}\times\mathbf{Q} &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ P_1 & P_2 & P_3 \\ Q_1 & Q_2 & Q_3 \end{vmatrix} = \mathbf{i}\begin{vmatrix} P_2 & P_3 \\ Q_2 & Q_3 \end{vmatrix} - \mathbf{j}\begin{vmatrix} P_1 & P_3 \\ Q_1 & Q_3 \end{vmatrix} + \mathbf{k}\begin{vmatrix} P_1 & P_2 \\ Q_1 & Q_2 \end{vmatrix} \\ &= (P_2Q_3 - P_3Q_2, -P_1Q_3 + P_3Q_1, P_1Q_2 - P_2Q_1),\end{aligned}$$

or, alternatively, by means of a matrix

$$= (Q_1, Q_2, Q_3)\begin{pmatrix} 0 & P_3 & -P_2 \\ -P_3 & 0 & P_1 \\ P_2 & -P_1 & 0 \end{pmatrix}. \tag{I.7}$$

◇ **Exercise I.4:** The cross product $\mathbf{P}\times\mathbf{Q}$ is perpendicular to both $\mathbf{P}$ and $\mathbf{Q}$. In what direction does it point?

The cross product is not commutative and is not associative. It is, however, distributive with respect to addition or subtraction of vectors. Hence $\mathbf{P}\times(\mathbf{Q}\pm\mathbf{T}) = \mathbf{P}\times\mathbf{Q} \pm \mathbf{P}\times\mathbf{T}$.

The magnitude of $\mathbf{P}\times\mathbf{Q}$ equals $|\mathbf{P}|\,|\mathbf{Q}|\sin\theta$, where θ is the angle between the two vectors. The cross product therefore has a simple geometric interpretation. Its magnitude equals the area of the parallelogram defined by the two vectors.

◇ **Exercise I.5:** Given that $\mathbf{P}\times\mathbf{Q} = 0$, what does it tell us about the vectors involved?

As an example, the vector equation of a straight line is shown below for the case where the direction of the line and one point on the line are known. Assume that $\mathbf{d}$ is a unit vector in the direction of the line and $\mathbf{P}_1$ is a given point on the line. The equation of the entire line is

$$\mathbf{P}(t) = \mathbf{P}_1 + t\mathbf{d}, \text{ for any real } t. \tag{I.8}$$

◇ **Exercise I.6:** Derive the vector line equation for the straight segment between two given points $\mathbf{P}_1$ and $\mathbf{P}_2$.

> What if angry vectors veer
> Round your sleeping head, and form.
> There's never need to fear
> Violence of the poor world's abstract storm.
>
> — Robert Penn Warren, *Lullaby in Encounter*, 1957.

I.4.2 The Scalar Triple Product

The scalar triple product of three vectors $\mathbf{P}$, $\mathbf{Q}$, and $\mathbf{R}$ is defined as

$$\begin{aligned} S = \mathbf{P} \bullet (\mathbf{Q} \times \mathbf{R}) &= P_1(Q_2R_3 - Q_3R_2) + P_2(Q_3R_1 - Q_1R_3) + P_3(Q_1R_2 - Q_2R_1) \\ &= \begin{vmatrix} P_1 & P_2 & P_3 \\ Q_1 & Q_2 & Q_3 \\ R_1 & R_2 & R_3 \end{vmatrix}. \end{aligned} \tag{I.9}$$

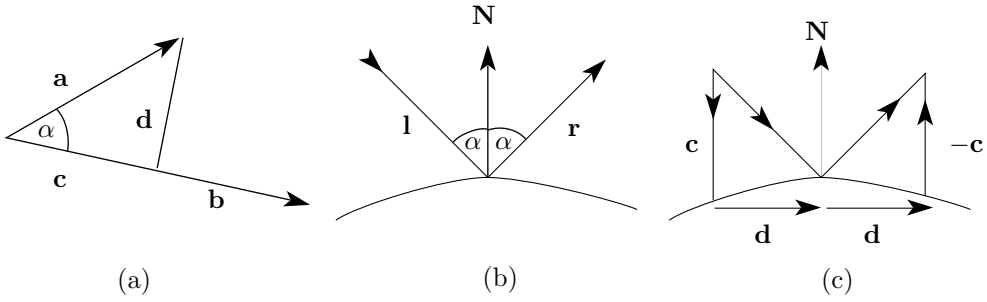
Interchanging two rows in a determinant changes its sign, so interchanging rows twice leaves the determinant unchanged. This is why the triple product is not affected by a cyclic permutation of its three components. We can therefore write

$$S = \mathbf{P} \bullet (\mathbf{Q} \times \mathbf{R}) = \mathbf{Q} \bullet (\mathbf{R} \times \mathbf{P}) = \mathbf{R} \bullet (\mathbf{P} \times \mathbf{Q}).$$

The triple product has a simple geometric interpretation. It equals the volume of the parallelepiped defined by the three vectors. An important corollary is: If the three vectors are coplanar, then the parallelepiped defined by them has volume zero, implying that their scalar triple product is zero.

I.4.3 Projecting a Vector

A common and useful operation on vectors is projecting a vector $\mathbf{a}$ on another vector $\mathbf{b}$. The idea is to break vector $\mathbf{a}$ up into two perpendicular components $\mathbf{c}$ and $\mathbf{d}$, such that $\mathbf{c}$ is in the direction of $\mathbf{b}$.



(a) (b) (c)

Figure I.1: Projecting a Vector.

Figure I.1a shows that $\mathbf{a} = \mathbf{c} + \mathbf{d}$ and $|\mathbf{c}| = |\mathbf{a}| \cos\alpha$. On the other hand $\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos\alpha$, yielding the magnitude of $\mathbf{c}$

$$|\mathbf{c}| = |\mathbf{a}| \frac{(\mathbf{a} \bullet \mathbf{b})}{|\mathbf{a}|\,|\mathbf{b}|} = \frac{(\mathbf{a} \bullet \mathbf{b})}{|\mathbf{b}|}. \tag{I.10}$$

The direction of $\mathbf{c}$ is identical to the direction of $\mathbf{b}$, so we can write vector $\mathbf{c}$ as

$$\mathbf{c} = |\mathbf{c}| \frac{\mathbf{b}}{|\mathbf{b}|} = \frac{(\mathbf{a} \bullet \mathbf{b})}{|\mathbf{b}|^2} \mathbf{b}. \tag{I.11}$$

Example: Given vectors $\mathbf{a} = (2, 1)$ and $\mathbf{b} = (1, 0)$ it is easy to calculate

$$\mathbf{c} = \frac{(\mathbf{a} \bullet \mathbf{b})}{|\mathbf{b}|^2}\mathbf{b} = \frac{2\times 1 + 1\times 0}{1^2 + 0^2}(2, 0) = (4, 0), \qquad \mathbf{d} = \mathbf{a} - \mathbf{c} = (-2, 1).$$

◇ **Exercise I.7:** The projection method above works also for three-dimensional vectors. Given vectors $\mathbf{a} = (2, 1, 3)$ and $\mathbf{b} = (1, 0, -1)$, calculate the projection of $\mathbf{a}$ on $\mathbf{b}$.

◇ **Exercise I.8:** Vectors and their operations have been known for a long time. Explain why they have become important in the last few decades, since the introduction of the digital computer.

I.5 Complex Numbers

Complex numbers are expressed in terms of the special number i that is defined as $\sqrt{-1}$ and, hence, satisfies $i\times i = i^2 = -1$. Any complex number z can be represented either as the sum $a+bi$ or as the pair (a, b), where a and b are real. The *conjugate* of z is denoted by z^* and is defined as $a - bi$. Complex conjugates roughly correspond to negative real numbers. The sum of the real numbers a and $-a$ is zero and the sum $z+z^*$ is $2a$, which is real. The *magnitude* or *absolute value* of a complex number is denoted by $|z|$ and is defined as $\sqrt{z \cdot z^*} = \sqrt{a^2 + b^2}$. The sum and the difference of the complex numbers $a+bi$, $c+di$ are the obvious $(a+b)\pm(c+d)i$. The product makes use of the relation $i^2 = -1$ and is $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$. The inverse, z^{-1}, of z is defined as $z^*/|z|$. It corresponds to the reciprocal $1/a$ of a real number a. The division z_1/z_2 is easy to perform for $|z_2| \neq 0$:

$$\frac{z_1}{z_2} = \frac{z_1 z_2^*}{z_2 z_2^*} = \frac{(a + bi)(c - di)}{c^2 + d^2} = \left(\frac{ac + bd}{c^2 + d^2}, \frac{bc - ad}{c^2 + d^2}\right).$$

The multiplication rule of complex numbers can be interpreted as a rotation in two dimensions. This is easy to see if we consider the product of the two complex numbers (x, y) and $(\cos\theta, \sin\theta)$.

$$\begin{aligned}(x, y) \cdot (\cos\theta, \sin\theta) &= (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta) \\ &= (x, y)\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.\end{aligned} \tag{I.12}$$

This product rotates the two-dimensional point (x, y) through an angle θ about the origin.

◇ **Exercise I.9:** Use the rule of complex number multiplication to multiply the complex number $(0, 1)$ by itself.

◇ **Exercise I.10:** Use the rule of complex number multiplication to multiply the complex number (a, b) by the number $(-a, -b)$.

> The shortest path between two truths in the real domain passes through the complex domain.
>
> —Jacques Hadamard.

Complex numbers can be represented graphically as two-dimensional points where the real part is the x coordinate and the imaginary part is the y coordinate. Such a representation is called an *Argand diagram*.

We normally use the *Cartesian coordinates* (a, b) of a point $\mathbf{P}$. The *polar coordinates* of $\mathbf{P}$ are (r, θ), where $r = \sqrt{a^2 + b^2}$ is the distance of the point from the origin and $\theta = \arctan(b/a)$ is the angle between the x axis and vector r. Given a complex number $z = (x, y)$, r is its absolute value and θ is called its *argument* (`arg` for short). The polar coordinates can be obtained from the Cartesian ones by $(a, b) = (r\cos\theta, r\sin\theta)$. Since the complex number $z = (a, b)$ can be interpreted as a two-dimensional point, it has the polar representation $z = (r\cos\theta, r\sin\theta) = r(\cos\theta + i\sin\theta)$. This representation is useful in many applications.

The famous Euler formula

$$e^{i\theta} = \cos\theta + i\sin\theta$$

allows us to write $z = re^{i\theta}$, a representation that makes it easy to multiply and divide complex numbers

$$z_1 z_2 = r_1 r_2 e^{i(\theta_1+\theta_2)}, \quad \frac{z_1}{z_2} = r_1 r_2 e^{i(\theta_1-\theta_2)},$$

and even extract roots

$$\sqrt[n]{z} = \sqrt[n]{r}\left[\cos\left(\frac{\theta + 2k\pi}{n}\right) + i\sin\left(\frac{\theta + 2k\pi}{n}\right)\right], \quad k = 0, 1, \ldots, n-1.$$

The n roots of z can be visualized as equally-spaced points lying on the circumference of a circle of radius $\sqrt[n]{r}$ whose center is at the origin. Connecting them produces an n-sided regular polygon.

◇ **Exercise I.11:** (Mathematical.) We know that $i = \sqrt{-1}$. What is $\sqrt{i}$?

◇ **Exercise I.12:** While we are at it, what are i^i and $\ln i$?

Bibliography

Nahin, Paul J., (1998) *An Imaginary Tale: The Story of* $\sqrt{-1}$, Princeton, NJ, Princeton University Press.

The Complex Number Song

(Tune: John Brown's Body)

Mine eyes have seen the glory of the Argand diagram
They have seen the i's and thetas of De Moivre's mighty plan
Now I can find the complex roots with consummate elan
With the root of minus one

Complex numbers are so easy
Complex numbers are so easy
Complex numbers are so easy
With the root of minus one

In Cartesian co-ordinates the complex plane is fine,
But the grandeur of the polar form this beauty doth outshine
You be raising i+40 to the power of 99
With the root of minus one

You'll realise your understanding was just second rate
When you see the power and magic of the complex conjugate
Drawing vectors corresponding to the roots of minus eight
With the root of minus one

(Attributed to Mrs P. E. Perella.)

I.6 Convolution

This is an important quantity that has several practical applications. It is used in Sections 5.6.1 and 5.8. We start with the simple, intuitive concept of a *system*. This is anything that receives input and generates output in response. The input and output can be one-dimensional (a function of the time), two-dimensional (a function of two spatial variables), or can have any number of dimensions. We will be concerned with the relation of the output to the input, not with the internal operation of the system. We will also concentrate on *linear systems*, since they are both simple and important. A linear system is defined as follows: If input $x_1(t)$ produces output $y_1(t)$ [we denote this by $x_1(t) \to y_1(t)$] and if $x_2(t) \to y_2(t)$, then $x_1(t) + x_2(t) \to y_1(t) + y_2(t)$. Any system that does not satisfy this condition is considered nonlinear.

This definition implies that $2x_1(t) = x_1(t) + x_1(t) \to y_1(t) + y_1(t) = 2y_1(t)$ or, in general, $a\,x_1(t) \to a\,y_1(t)$ for any real a.

Some linear systems are *shift invariant*. If such a linear system satisfies $x(t) \to y(t)$, then $x(t-T) \to y(t-T)$, i.e., shifting the input by an amount T shifts the output by the same amount, but does not otherwise affect the output. In the discussion of convolution, we assume that the systems in question are linear and shift-invariant. This is true (or true to a very good approximation) for electrical networks and optical systems, the main pieces of hardware used in image processing and compression.

It is useful to have a general relation between the input and output of a linear,

System. Frequently used without need.	
Dayton has adopted the commission system of government.	Dayton has adopted government by commission.
The dormitory system	Dormitories

— Strunk and White, *The Elements of Style*.

shift-invariant system. It turns out that the expression

$$y(t) = \int_{-\infty}^{+\infty} f(t,\tau)x(\tau)\,d\tau, \tag{I.13}$$

is general enough for this purpose. In other words, there is always a two-parameter function $f(t,\tau)$ that can be used to predict the output $y(t)$ if the input $x(\tau)$ is known. However, we want to express this relation with a one-parameter function, and we use the shift-invariance of the system for this purpose. For a linear, shift-invariant system we can write

$$y(t-T) = \int_{-\infty}^{+\infty} f(t,\tau)x(\tau-T)\,d\tau.$$

If we change variables by adding T to both t and τ, we get

$$y(t) = \int_{-\infty}^{+\infty} f(t+T,\tau+T)x(\tau)\,d\tau. \tag{I.14}$$

Comparing Equations (I.13) and (I.14) shows that $f(t,\tau) = f(t+T,\tau+T)$. Thus, function f has the property that if we add T to both its parameters, it does not change. The function is constant as long as the difference between its parameters is constant. Function f depends only on the difference of its parameters, so it is essentially a single parameter function. We can therefore write $g(t-\tau) = f(t,\tau)$, which changes Equation (I.13) to

$$y(t) = \int_{-\infty}^{+\infty} g(t-\tau)x(\tau)\,d\tau. \tag{I.15}$$

This is the *convolution integral*, an important relation between $x(t)$ and $y(t)$ or between $x(t)$ and $g(t)$. This relation is denoted $y = g \star x$ and it says that the output of a linear, shift-invariant system is given by the convolution of its input with a certain function $g(t)$ (or by *convolving* x with g). Function g, which is characteristic of the system, is called the *impulse response* of the system. Figure I.2 shows a graphical description of a convolution, where the final result (the integral) is the gray area under the curve.

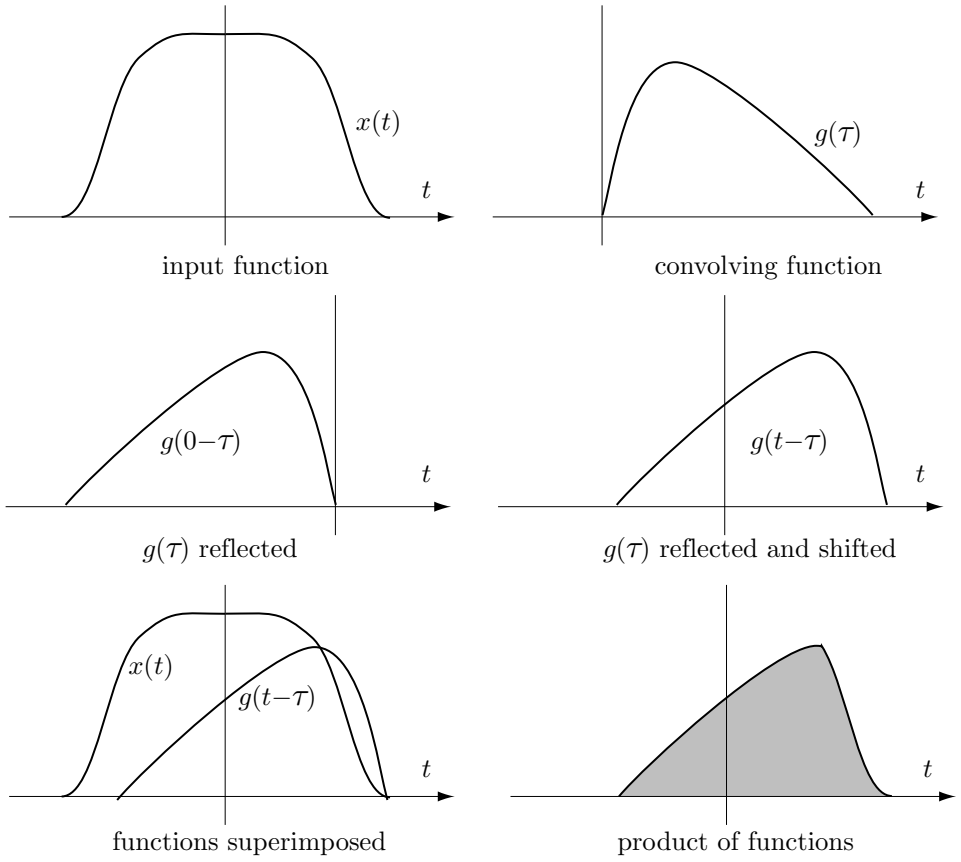


Figure I.2: The Convolution of $x(t)$ and $g(t)$.

"Oh no," George said. "It was more than money."

He leaned his forehead in his hand and tried to remember what else more than money. The darkness inside his head was full of convolutions. His eardrums were too tight. Only the higher registers of sound were getting through.

—Paul Scott, *The Bender*

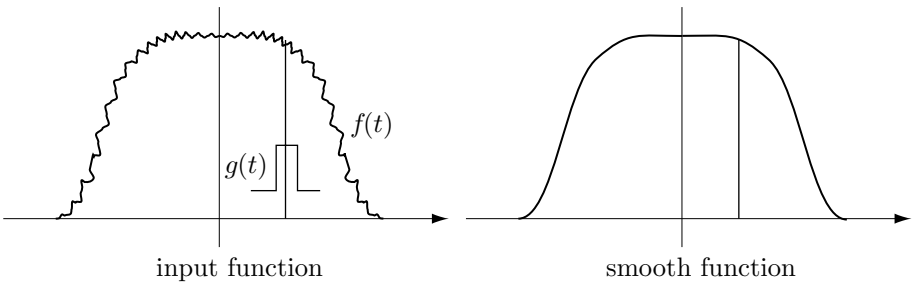


Figure I.3: Applying Convolution to Denoising a Function.

The convolution has a number of important properties. It is commutative, associative, and distributive over addition. These properties are shown in Equation (I.16)

$$\begin{aligned} f \star g &= g \star f, \\ f \star (g \star h) &= (f \star g) \star h, \\ f \star (g + h) &= f \star g + f \star h. \end{aligned} \tag{I.16}$$

Practical problems normally involve discrete sequences of numbers, rather than continuous functions, so the *discrete convolution* is useful. The discrete convolution of the two sequences $f(i)$ and $g(i)$ is defined as

$$h(i) = f(i) \star g(i) = \sum_j f(j)\, g(i-j). \tag{I.17}$$

If the lengths of $f(i)$ and $g(i)$ are m and n, respectively, then $h(i)$ has length $m + n - 1$.

Example: Given the two sequences $f = \big(f(0), f(1), \ldots, f(5)\big)$ (six elements) and $g = \big(g(0), g(1), \ldots, g(4)\big)$ (five elements), Equation (I.17) yields the ten elements of the convolution $h = f \star g$

$$h(0) = \sum_{j=0}^{0} f(j)g(0-j) = f(0)g(0)$$

$$h(1) = \sum_{j=0}^{1} f(j)g(1-j) = f(0)g(1) + f(1)g(0)$$

$$h(2) = \sum_{j=0}^{2} f(j)g(2-j) = f(0)g(2) + f(1)g(1) + f(2)g(0)$$

$$h(3) = \sum_{j=0}^{3} f(j)g(3-j) = f(0)g(3) + f(1)g(2) + f(2)g(1) + f(3)g(0)$$

$$h(4) = \sum_{j=0}^{4} f(j)g(4-j) = f(0)g(4) + f(1)g(3) + f(2)g(2) + f(3)g(1) + f(4)g(0)$$

$$h(5) = \sum_{j=1}^{5} f(j)g(5-j) = f(1)g(4) + f(2)g(3) + f(3)g(2) + f(4)g(1) + f(5)g(0)$$

$$h(6) = \sum_{j=2}^{5} f(j)g(6-j) = f(2)g(4) + f(3)g(3) + f(4)g(2) + f(5)g(1)$$

$$h(7) = \sum_{j=3}^{5} f(j)g(7-j) = f(3)g(4) + f(4)g(3) + f(5)g(2)$$

$$h(8) = \sum_{j=4}^{5} f(j)g(8-j) = f(4)g(4) + f(5)g(3)$$
$$h(9) = \sum_{j=5}^{5} f(j)g(9-j) = f(5)g(4)$$

A simple example of the use of a convolution is smoothing (or denoising). This shows how convolution can be used as a filter. Given a noisy function $f(t)$ (Figure (I.16)), we select a rectangular pulse as the convolving function $g(t)$. It is defined as

$$g(t) = \begin{cases} 1, & -a/2 < t < a/2, \\ \frac{1}{2}, & t = \pm a/2, \\ 0, & \text{elsewhere}, \end{cases}$$

where a is a suitably small value (typically 1, but could be anything). As the convolution proceeds, the pulse is moved from left to right and is multiplied by $f(t)$. The result of the product is a local average of $f(t)$ over an interval of width a. This has the effect of suppressing the high frequency fluctuations of $f(t)$.

From the Dictionary
convolution: coiling together
convolve: roll together

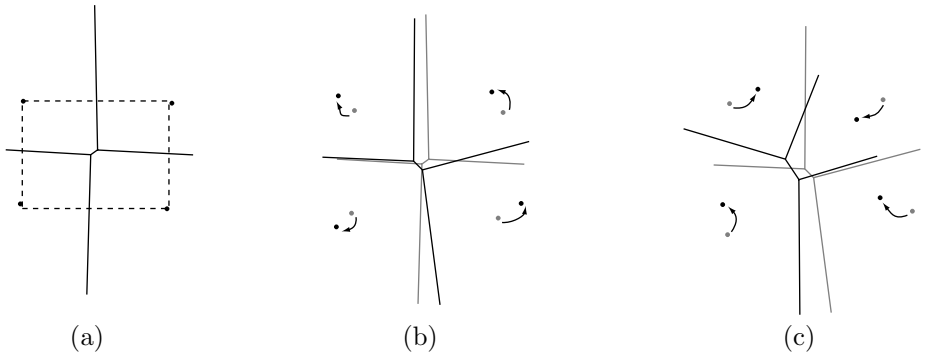
I.7 Voronoi Diagrams

Imagine a petri dish ready for growing bacteria. Four bacteria of different types are simultaneously placed in it at different points and immediately start multiplying. We assume that their colonies grow at the same rate. Initially, each colony consists of a growing circle around one of the starting points. After a while some of them meet and stop growing in the meeting area due to lack of food. The final result is that the entire dish gets divided into four areas, one around each of the four starting points, such that all the points within area i are closer to starting point i than to any other start point. Such areas are called *Voronoi regions* or *Dirichlet tessellations*. Figure I.4a shows the Voronoi regions for four points placed approximately at the four corners of a dashed rectangle. The regions are close to the four quadrants of the rectangle. Figure I.4b,c shows how the regions change when the points are moved.

At the time of writing there are on the web several Java applets that demonstrate the concepts discussed here. A typical example is [Zhao 98].

Bibliography

Zhao, Zhiyuan (1998) is an applet at
`http://ra.cfm.ohio-state.edu/~zhao/algorithms/algorithms.html`.



(a) (b) (c)

Figure I.4: Three Voronoi Diagrams of Four Points.

I.8 L Systems

Lindenmayer Systems (or L-systems for short) were developed by the biologist Aristid Lindenmayer in 1968 as a tool [Lindenmayer 68] to describe the morphology of plants. They were initially used in computer science, in the 1970s, as a tool to define formal languages, but have become really popular only after 1984, when Alvy Ray Smith pointed out [Smith 84] that L-systems can be used to draw many types of fractals, in addition to their use in botany. Today L-systems are also used to generate tilings, geometric art, and even musical scores.

The main idea of L-systems is to define a complex object by (1) defining an initial simple object, called the *axiom*, and (2) giving rules that show how to replace parts of the axiom.

> The following true story is an example of Aristid's modesty. At one of the American conferences somebody asked him what the L in "L-systems" stands for. Aristid's answer was "Languages."
>
> —Grzegorz Rozenberg.

The rules are applied successively, creating parts that get more and more complex, thereby transforming the simple axiom closer to the final, complex goal. The rules are called *rewriting* or *production* rules, and are an extension of Chomsky's work on formal grammars, and also of the BNF notation. N. Chomsky showed, in the 1950s, how to describe the syntax of a natural language by means of production rules. At about the same time Backus and Naur developed the BNF notation, which is based on rewriting rules, specifically to provide a formal definition [Naur 60] of the syntax of ALGOL 60.

Figure I.5 shows how a fractal, the Koch snowflake curve, is constructed, in several steps, out of an axiom that is a simple triangle (I.5a) and a rewriting rule that says: Replace each straight segment with the curve of I.5b. Figure I.5c is the

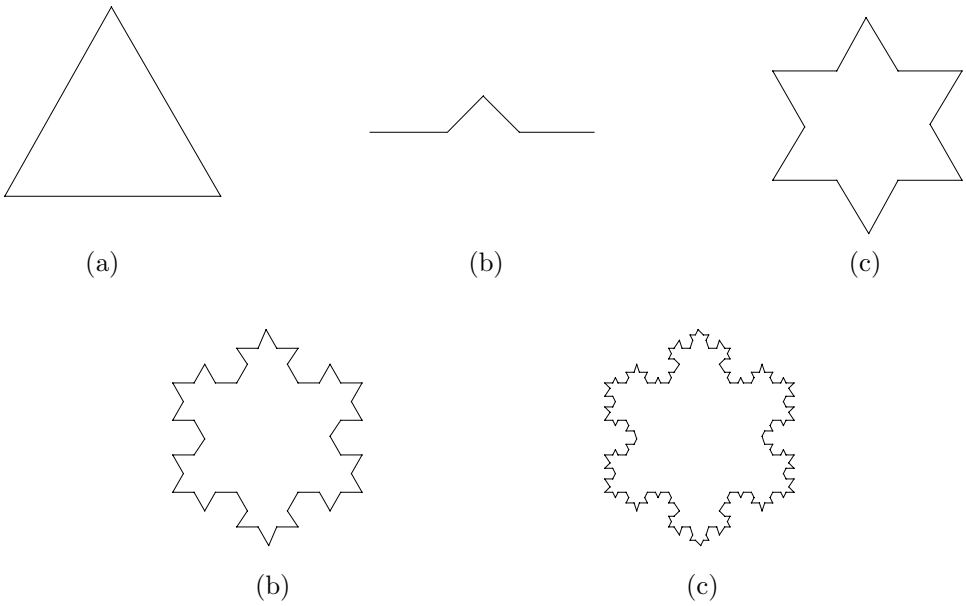


Figure I.5: Successive Generations of the Koch Snowflake.

result of applying the rule on **all** three triangle sides. Figure I.5d is the result of applying the same rule on all 12 sides of I.5c, and so on.

Notice that in order to construct iteration $i+1$ of an object, the rule has to be applied to **all** parts of iteration i of the object. This is the main difference between L-systems and Chomsky grammars, and this is also one reason why L-systems are so powerful. Another reason is the notation used in modern L-systems, a notation introduced in 1979 by A. Szilard and R. E. Quinton and improved by P. Prusinkiewicz in 1986. It is based on the LOGO language [Abelson 82] and the concept of turtle moves. These two differences are illustrated below.

Example: An L-system dealing with the two letters x and y. The axiom is y and the two rewriting rules are: $x \to xy$ (every occurrence of x should be replaced by xy) and $y \to x$ (every occurrence of y should be replaced by x). The first iteration starts with the axiom y, and applies **both** rules to it. The first rule does not apply, and the second yields x. The result of the first iteration is thus x. Iteration 2 applies both rules to x. The first rule replaces x by xy and the second rule does not apply, since the original string x did not have any y in it. Iteration 3 replaces the x of xy by xy and the y of xy by x. The result is xyx. Successive iterations produce the strings

$$y \to x \to xy \to xyx \to xyxxy \to xyxxyxyx.$$

(This does not seem useful, but wait until this method is applied to geometric shapes.) The two parts on the left and right of a production rule are called its *predecessor* and *successor*, respectively. An L-system such as the one above is called

a D0L-system. (D0L stands for Deterministic, Context-Free L-system. Notice that most texts on L-systems corrupt this name and spell it "DOL" instead of "D0L.")

Turtle Moves: It is possible to define geometric shapes by imagining a turtle moving in the two-dimensional plane, sometimes leaving marks behind. The LOGO programming language supports drawing commands that "move" the turtle from point to point and cause it to turn at an angle when it reaches a point. The production rules of L-systems also use this notation. Mathematically, the state of the turtle is represented by a triplet (x, y, α) where (x, y) are the present coordinates of the turtle and α is its heading. The basic notation used in such a rule employs the following characters:

F : The turtle moves forward a distance d, drawing a straight line of a given thickness W. The state of the turtle changes from (x, y, α) to $(x + d\cos\alpha, y + d\sin\alpha, \alpha)$.

f : The turtle moves forward as above, but without drawing anything.

$+$: The turtle turns to the right (clockwise) by a given angle δ. Its new state is thus $(x, y, \alpha + \delta)$.

$-$: The turtle turns to the left (counterclockwise) by the same angle δ. Its new state is $(x, y, \alpha - \delta)$.

Table I.7 shows several more character commands that have traditionally been used in L-systems. As more research is done in this field, the number of turtle commands will grow, but the reader has to keep one important convention in mind: When a rewriting rule contains a command that the turtle (i.e., the computer implementation of L-systems) does not understand, *that command is ignored*; no error message is issued. This convention is useful and is commonly used in drawing complex shapes.

Table I.7 implies that several more parameters, such as C, sl, and Δ, are needed to completely specify the shape being drawn. These parameters should be supported by any computer implementation of L-systems; they should have default values, and should be easy for the user to modify. These parameters are listed in Table I.8.

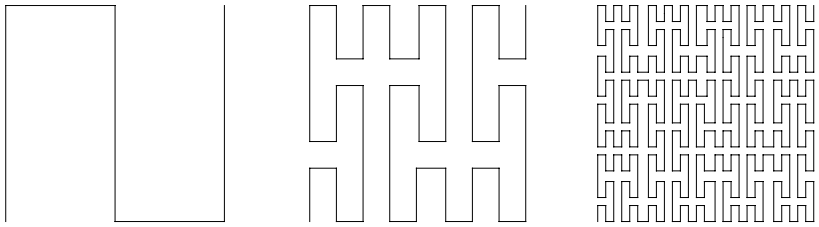
The string `F+F+F+F` is a command to move forward one line length, turn right, and repeat three more times. If the turn angle is 90°, the result is a square of size d. If the initial turtle heading is $\alpha = 90°$, then the start/end point is the bottom left corner of the square (Figure I.9a). The string `FFF+FF+F+F-F-F-FF+F+FFF` draws the shape of Figure I.9b.

The Koch snowflake of Figure I.5 was generated by an L-system with an axiom `F++F++F` and the single production rule `F->F-F++F-F`. The initial heading was 0° and the turn angle 60°. Figure I.6 shows three iterations of the Peano space-filling curve drawn with both an initial heading and a turn angle of 90°.

The L-system for this curve consists of the axiom `X` and the two production rules

`X->XFYFX+F+YFXFY-F-XFYFX` and `Y->YFXFY-F-XFYFX+F+YFXFY`.

The key to understanding this L-system is the rule that any unknown turtle commands (in this case the characters `X` and `Y`) should be ignored. The first iteration draws the axiom `X`, which is unknown, causing nothing to be drawn. The next iteration executes the two rewriting rules. The first rule replaces the axiom `X` with



(a) (b) (c)

Figure I.6: Three Iterations of the Peano Curve.

`XFYFX+F+YFXFY-F-XFYFX`, which is plotted (since `X` and `Y` are unknown) as `FF+F+FF-F-FF`. The second rule looks for a `Y` in the axiom, but finds none. The iteration thus draws `FF+F+FF-F-FF`, which results in Figure I.6a. The next iteration starts with `XFYFX+F+YFXFY-F-XFYFX`, replaces each `X` with the successor of rule 1, and replaces each `Y` with the successor of rule 2. The result is a very long string, that, when drawn, produces the curve of Figure I.6b.

The L-system for the Hilbert curve is similarly defined by the axiom `X` and the 2 production rules

`X->-YF+XFX+FY-` and `Y->+XF-YFY-FX+`.

◇ **Exercise I.13:** Show how to get the 4 orientations of the Hilbert curve out of the L-system above.

Abelson, H. and A. A. diSessa (1982) *Turtle Geometry*, Cambridge, MA, MIT Press.

Prusinkiewicz, Przemysław (1986) *Graphical Applications of L-systems*, in Proc. of Graphics Interface '86—Vision Interface '86, pp .247–253.

Prusinkiewicz, P., and A. Lindenmayer (1990) *The Algorithmic Beauty of Plants*, New York, Springer Verlag.

Prusinkiewicz, P., A. Lindenmayer, and F. D. Fracchia (1991) "Synthesis of Space-Filling Curves on the Square Grid," in *Fractals in the Fundamental and Applied Sciences*, edited by Peitgen, H.-O. et al., Amsterdam, Elsevier Science Publishers, pp. 341–366.

Smith, Alvy Ray (1984) "Plants, Fractals and Formal Languages," *Computer Graphics* **18**(3):1–10.

Szilard, A. L. and R. E. Quinton (1979) "An Interpretation for D0L Systems by Computer Graphics," *The Science Terrapin* **4**:8–13.

`F`	Move forward d units and draw a line.
`f`	Move forward d units without drawing.
`+`	Turn clockwise by an angle δ.
`-`	Turn counterclockwise by an angle δ.
`|`	Reverse direction (rotate by 180°).
`[`	Push current turtle state into the stack.
`]`	Pop current turtle state from the stack.
`#`	Increment the line width W by an amount w.
`!`	Decrement the line width W by an amount w.
`@`	Draw a dot with radius W.
`{`	Open a polygon.
`}`	Close a polygon and fill it with color C.
`<`	Divide line length d by scale factor sl.
`>`	Multiply line length d by scale factor sl.
`&`	Swap meaning of + and −.
`(`	Decrement turn angle δ by Δ.
`)`	Increment turn angle δ by Δ.
`*`	Match any character (used in context-sensitive L-systems only).
...	Ignore rule (used in context-sensitive L-systems only).

Table I.7: L-system Conventions for Turtle Commands.

d	The line length.
sl	Scale factor for line length d.
W	The line width.
w	The line width increment.
α	The initial turtle heading.
δ	Turn angle.
Δ	Increment/decrement the turn angle δ.
C	Default color for polygon fill.

Table I.8: Additional Turtle Parameters.

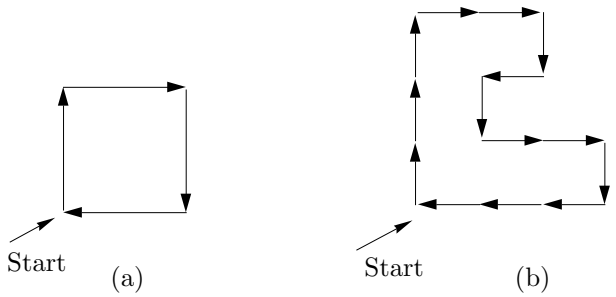


Figure I.9: Examples of Turtle Movements.

I.9 The Greek Alphabet

A	α	alpha	I	ι	iota	P	ρ	ϱ	rho
B	β	beta	K	κ	kappa	Σ	σ	ς	sigma
Γ	γ	gamma	Λ	λ	lambda	T	τ		tau
Δ	δ	delta	M	μ	mu	Y	υ		upsilon
E	ϵ	epsilon	N	ν	nu	Φ	ϕ	φ	phi
Z	ζ	zeta	Ξ	ξ	xi	X	ξ		xi
H	η	eta	O	o	omicron	Ψ	ψ		psi
Θ	θ	theta	Π	π	ϖ pi	Ω	ω		omega

I.10 Interpolating Polynomials

This section shows how to predict the value of a pixel from those of 16 of its near neighbors by means of a two-dimensional interpolating polynomial. The results are used in Table 4.118.

We start with an intuitive discussion of the term *interpolation*. Given two numbers a and b, their average $(a+b)/2$ is always located midway between them, so we can use the average to interpolate them. However, given four numbers a, b, c, and d, their average $(a+b+c+d)/4$ is not a good interpolation, since it is not located "midway" between the four. A simple example is the four numbers 1, 1, 1, and 100. Their average is close to 25, so it is nowhere "in the middle" of the four numbers. Interpolating four numbers is therefore done by (1) converting the numbers to two-dimensional points, (2) calculating a smooth curve that passes through the points, and (3) finding the midpoint of the curve.

Any numbers a, b, c, and d can be converted to the points $(1,a)$, $(2,b)$, $(3,c)$, and $(4,d)$. It is intuitively clear that the midpoint (x,y) of a smooth curve that passes through those points is a good candidate for the title "the interpolation of the four points." The y coordinate becomes the interpolation of the four numbers, and the x coordinate is ignored.

This method is called one-dimensional interpolation. It can be extended to more than four numbers, and also to pixels, where it becomes two-dimensional interpolation. As mentioned before, we want to use a group of 16 neighboring pixels to predict the value of a pixel at the center of the group. The main idea is to consider the 16 neighbor pixels a 4×4 equally-spaced points on a surface (where the value of a pixel is interpreted as the height of the surface) and to derive a polynomial function $\mathbf{P}(u,w)$ that passes through all 16 points. Graphically, $\mathbf{P}(u,w)$ can be thought of as a surface. The value of the pixel at the center of the 4×4 group can then be predicted by calculating the height of the center point $\mathbf{P}(.5,.5)$ of the surface. Mathematically, this surface is the two-dimensional polynomial interpolation of the 16 points.

I.10.1 One-Dimensional Interpolation

A surface can be viewed as an extension of a curve, so we start by deriving a one-dimensional polynomial (a curve) that interpolates four points, then extend it to a two-dimensional polynomial (a surface) that interpolates a grid of 4×4 points.

Given four points $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$, and $\mathbf{P}_4$ we look for a polynomial that will pass through them. In general, a polynomial of degree n in x is defined (Section 3.23)

as the function

$$P_n(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \tag{I.18}$$

where a_i are the $n+1$ coefficients of the polynomial and the parameter x is a real number. The one-dimensional interpolating polynomial that is of interest to us is special, and differs from the definition above in two respects

1. This polynomial goes from point $\mathbf{P}_1$ to point $\mathbf{P}_4$. Its length is finite, and it is therefore better to describe it as the function

$$P_n(t) = \sum_{i=0}^{n} a_i t^i = a_0 + a_1 t + a_2 t^2 + \cdots + a_n t^n; \text{ where } 0 \le t \le 1.$$

This is the *parametric representation* of a polynomial. We want this polynomial to go from $\mathbf{P}_1$ to $\mathbf{P}_4$ when the parameter t is varied from 0 to 1.

2. The only given data are the four points and we have to use them to calculate all $n+1$ coefficients of the polynomial. This suggests the value $n = 3$ (a polynomial of degree 3, a cubic polynomial; one which has four coefficients). The idea is to set up and solve four equations, with the four coefficients as the unknowns, and with the four points as known quantities. Thus, we use the notation (T indicates transpose)

$$\mathbf{P}(t) = \mathbf{a}t^3 + \mathbf{b}t^2 + \mathbf{c}t + \mathbf{d} = (t^3, t^2, t, 1)(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})^T = \mathbf{T}(t) \cdot \mathbf{A}. \tag{I.19}$$

The four coefficients $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$ are shown in boldface because they are not numbers. Keep in mind that the polynomial has to pass through the given points, so the value of $\mathbf{P}(t)$ for any t must be the three coordinates of a point. Each coefficient should therefore be a triplet. $\mathbf{T}(t)$ is the row vector $(t^3, t^2, t, 1)$, and $\mathbf{A}$ is the column vector $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})^T$. Calculating the curve therefore involves finding the values of the four unknowns $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$. $\mathbf{P}(t)$ is called a *parametric cubic* (or PC) polynomial.

It turns out that degree 3 is the smallest one that is still useful for an interpolating polynomial. A polynomial of degree 1 has the form $\mathbf{P}_1(t) = \mathbf{c}t + \mathbf{d}$ and is, therefore, a straight line, so it can only be used in special cases. A polynomial of degree two (quadratic) has the form $\mathbf{P}_2(t) = \mathbf{b}t^2 + \mathbf{c}t + \mathbf{d}$ and is a conic section, so it can only take a few different shapes. A polynomial of degree 3 (cubic) is thus the simplest one that can take on complex shapes, and can also be a space curve.

◇ **Exercise I.14:** Prove that a quadratic polynomial must be a plane curve.

Our ultimate problem is to interpolate pixels. Pixels are always equally-spaced, so we assume that the two interior points $\mathbf{P}_2$ and $\mathbf{P}_3$ are equally spaced between $\mathbf{P}_1$ and $\mathbf{P}_4$. The first point $\mathbf{P}_1$ is the start point $\mathbf{P}(0)$ of the polynomial, the last point, $\mathbf{P}_4$ is the endpoint $\mathbf{P}(1)$, and the two interior points $\mathbf{P}_2$ and $\mathbf{P}_3$ are the two equally-spaced interior points $\mathbf{P}(1/3)$ and $\mathbf{P}(2/3)$ of the polynomial.

We thus write $\mathbf{P}(0) = \mathbf{P}_1$, $\mathbf{P}(1/3) = \mathbf{P}_2$, $\mathbf{P}(2/3) = \mathbf{P}_3$, $\mathbf{P}(1) = \mathbf{P}_4$, or

$$\begin{aligned}
\mathbf{a}(0)^3 + \mathbf{b}(0)^2 + \mathbf{c}(0) + \mathbf{d} &= \mathbf{P}_1,\\
\mathbf{a}(1/3)^3 + \mathbf{b}(1/3)^2 + \mathbf{c}(1/3) + \mathbf{d} &= \mathbf{P}_2,\\
\mathbf{a}(2/3)^3 + \mathbf{b}(2/3)^2 + \mathbf{c}(2/3) + \mathbf{d} &= \mathbf{P}_3,\\
\mathbf{a}(1)^3 + \mathbf{b}(1)^2 + \mathbf{c}(1) + \mathbf{d} &= \mathbf{P}_4.
\end{aligned}$$

These equations are easy to solve and the solutions are:

$$\begin{aligned}
\mathbf{a} &= -9/2\mathbf{P}_1 + 27/2\mathbf{P}_2 - 27/2\mathbf{P}_3 + 9/2\mathbf{P}_4,\\
\mathbf{b} &= 9\mathbf{P}_1 - 45/2\mathbf{P}_2 + 18\mathbf{P}_3 - 9/2\mathbf{P}_4,\\
\mathbf{c} &= -11/2\mathbf{P}_1 + 9\mathbf{P}_2 - 9/2\mathbf{P}_3 + \mathbf{P}_4,\\
\mathbf{d} &= \mathbf{P}_1.
\end{aligned}$$

Substituting into Equation (I.19) gives

$$\begin{aligned}
\mathbf{P}(t) =&(-9/2\mathbf{P}_1 + 27/2\mathbf{P}_2 - 27/2\mathbf{P}_3 + 9/2\mathbf{P}_4)t^3\\
&+ (9\mathbf{P}_1 - 45/2\mathbf{P}_2 + 18\mathbf{P}_3 - 9/2\mathbf{P}_4)t^2\\
&+ (-11/2\mathbf{P}_1 + 9\mathbf{P}_2 - 9/2\mathbf{P}_3 + \mathbf{P}_4)t + \mathbf{P}_1.
\end{aligned}$$

Which, after rearranging, becomes

$$\begin{aligned}
\mathbf{P}(t) =&(-4.5t^3 + 9t^2 - 5.5t + 1)\mathbf{P}_1 + (13.5t^3 - 22.5t^2 + 9t)\mathbf{P}_2\\
&+ (-13.5t^3 + 18t^2 - 4.5t)\mathbf{P}_3 + (4.5t^3 - 4.5t^2 + t)\mathbf{P}_4\\
=&G_1(t)\mathbf{P}_1 + G_2(t)\mathbf{P}_2 + G_3(t)\mathbf{P}_3 + G_4(t)\mathbf{P}_4\\
=&\mathbf{G}(t)\cdot\mathbf{P},
\end{aligned} \tag{I.20}$$

where

$$\begin{aligned}
G_1(t) &= (-4.5t^3 + 9t^2 - 5.5t + 1), \qquad & G_2(t) &= (13.5t^3 - 22.5t^2 + 9t),\\
G_3(t) &= (-13.5t^3 + 18t^2 - 4.5t), & G_4(t) &= (4.5t^3 - 4.5t^2 + t);
\end{aligned} \tag{I.21}$$

$\mathbf{P}$ is the column $(\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \mathbf{P}_4)^T$, and $\mathbf{G}(t)$ is the row vector

$$\big(G_1(t), G_2(t), G_3(t), G_4(t)\big).$$

The functions $G_i(t)$ are called *blending functions*, since they create any point on the curve as a blend of the four given points. Note that they add up to 1 for any value of t. This property must be satisfied by any set of blending functions, and such functions are called *barycentric*. We can also write

$$G_1(t) = (t^3, t^2, t, 1)(-4.5, 9, -5.5, 1)^T$$

and, similarly, for $G_2(t)$, $G_3(t)$, and $G_4(t)$. In matrix notation this becomes

$$\mathbf{G}(t) = (t^3, t^2, t, 1) \begin{pmatrix} -4.5 & 13.5 & -13.5 & 4.5 \\ 9.0 & -22.5 & 18 & -4.5 \\ -5.5 & 9.0 & -4.5 & 1.0 \\ 1.0 & 0 & 0 & 0 \end{pmatrix} = \mathbf{T}(t) \cdot \mathbf{N}. \tag{I.22}$$

The curve can now be written $\mathbf{P}(t) = \mathbf{G}(t) \cdot \mathbf{P} = \mathbf{T}(t) \cdot \mathbf{N} \cdot \mathbf{P}$. $\mathbf{N}$ is called the basis matrix and $\mathbf{P}$ is the geometry vector. From Equation (I.19) we know that $\mathbf{P}(t) = \mathbf{T}(t) \cdot \mathbf{A}$, so we can write $\mathbf{A} = \mathbf{N} \cdot \mathbf{P}$. Equation (5.17) illustrates an application of this interpolating polynomial for image compression.

The word *barycentric* is derived from *barycenter*, meaning "center of gravity," because such weights are used to calculate the center of gravity of an object. Barycentric weights have many uses in geometry in general, and in curve and surface design in particular.

Given the four points, the interpolating polynomial can be calculated in two steps:

1. Set-up the equation $\mathbf{A} = \mathbf{N} \cdot \mathbf{P}$ and solve it for $\mathbf{A} = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})^T$.
2. The polynomial is $\mathbf{P}(t) = \mathbf{T}(t) \cdot \mathbf{A}$.

I.10.2 Example

(This example is in two dimensions, each of the four points $\mathbf{P}_i$ and each of the four coefficients $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ is a pair. For three-dimensional curves, the method is the same, except that triplets should be used, instead of pairs.) Given the four two-dimensional points $\mathbf{P}_1 = (0,0)$, $\mathbf{P}_2 = (1,0)$, $\mathbf{P}_3 = (1,1)$, and $\mathbf{P}_4 = (0,1)$, we set up the equation

$$\begin{pmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{pmatrix} = \mathbf{A} = \mathbf{N} \cdot \mathbf{P} = \begin{pmatrix} -4.5 & 13.5 & -13.5 & 4.5 \\ 9.0 & -22.5 & 18 & -4.5 \\ -5.5 & 9.0 & -4.5 & 1.0 \\ 1.0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} (0,0) \\ (1,0) \\ (1,1) \\ (0,1) \end{pmatrix},$$

which is easy to solve

$$\begin{aligned} \mathbf{a} &= -4.5(0,0) + 13.5(1,0) - 13.5(1,1) + 4.5(0,1) = (0,-9), \\ \mathbf{b} &= 19(0,0) - 22.5(1,0) + 18(1,1) - 4.5(0,1) = (-4.5, 13.5), \\ \mathbf{c} &= -5.5(0,0) + 9(1,0) - 4.5(1,1) + 1(0,1) = (4.5, -3.5), \\ \mathbf{d} &= 1(0,0) - 0(1,0) + 0(1,1) - 0(0,1) = (0,0). \end{aligned}$$

$$\text{Thus} \qquad \mathbf{P}(t) = \mathbf{T} \cdot \mathbf{A} = (0,-9)t^3 + (-4.5, 13.5)t^2 + (4.5, -3.5)t.$$

It is now easy to calculate and verify that $\mathbf{P}(0) = (0,0) = \mathbf{P}_1$, and

$$\mathbf{P}(1/3) = (0,-9)1/27 + (-4.5,13.5)1/9 + (4.5,-3.5)1/3 = (1,0) = \mathbf{P}_2,$$

$$\mathbf{P}(1) = (0,-9)1^3 + (-4.5,13.5)1^2 + (4.5,-3.5)1 = (0,1) = \mathbf{P}_4.$$

◇ **Exercise I.15:** Calculate $\mathbf{P}(2/3)$ and verify that it is equal to $\mathbf{P}_3$.

◇ **Exercise I.16:** Imagine the circular arc of radius one in the first quadrant (a quarter circle). Write the coordinates of the four points that are equally spaced on this arc. Use the coordinates to calculate a PC interpolating polynomial approximating this arc. Calculate point $\mathbf{P}(1/2)$. How far does it deviate from the midpoint of the true quarter circle?

The main advantage of this method is its simplicity. Given the four points, it is easy to calculate the PC polynomial that passes through them.

◇ **Exercise I.17:** This method makes sense if the four points are (at least approximately) equally spaced along the curve. If they are not, the following may be done: Instead of using 1/3 and 2/3 as the intermediate values, the user may specify values α, β such that $\mathbf{P}_2 = \mathbf{P}(\alpha)$ and $\mathbf{P}_3 = \mathbf{P}(\beta)$. Generalize Equation (I.22) such that it depends on α and β.

I.10.3 Two-Dimensional Interpolation

The PC polynomial, Equation (I.19), can easily be extended to two dimensions by means of a technique called *Cartesian product*. The polynomial is generalized from a cubic curve to a *bicubic* surface.

A one-dimensional PC polynomial has the form $\mathbf{P}(t) = \sum_{i=0}^{3} \mathbf{a}_i t^i$. Two such curves, $\mathbf{P}(u)$ and $\mathbf{P}(w)$, can be combined by means of this technique to form the surface:

$$\begin{aligned}
\mathbf{P}(u,w) &= \sum_{i=0}^{3}\sum_{j=0}^{3} \mathbf{a}_{ij} u^i w^j \\
&= \mathbf{a}_{33}u^3w^3 + \mathbf{a}_{32}u^3w^2 + \mathbf{a}_{31}u^3w + \mathbf{a}_{30}u^3 + \mathbf{a}_{23}u^2w^3 + \mathbf{a}_{22}u^2w^2 + \mathbf{a}_{21}u^2w + \mathbf{a}_{20}u^2 \\
&+ \mathbf{a}_{13}uw^3 + \mathbf{a}_{12}uw^2 + \mathbf{a}_{11}uw + \mathbf{a}_{10}u + \mathbf{a}_{03}w^3 + \mathbf{a}_{02}w^2 + \mathbf{a}_{01}w + \mathbf{a}_{00} \\
&= (u^3,u^2,u,1)\begin{pmatrix} \mathbf{a}_{33} & \mathbf{a}_{32} & \mathbf{a}_{31} & \mathbf{a}_{30} \\ \mathbf{a}_{23} & \mathbf{a}_{22} & \mathbf{a}_{21} & \mathbf{a}_{20} \\ \mathbf{a}_{13} & \mathbf{a}_{12} & \mathbf{a}_{11} & \mathbf{a}_{10} \\ \mathbf{a}_{03} & \mathbf{a}_{02} & \mathbf{a}_{01} & \mathbf{a}_{00} \end{pmatrix}\begin{pmatrix} w^3 \\ w^2 \\ w \\ 1 \end{pmatrix}, \quad \text{where } 0 \le u,w \le 1. \qquad \text{(I.23)}
\end{aligned}$$

This is a double cubic polynomial (hence the name *bicubic*) with 16 terms, where each of the 16 coefficients $\mathbf{a}_{ij}$ is a triplet. Note that the surface depends on all 16 coefficients. Any change in any of them produces a different surface. Equation (I.23) is the *algebraic representation* of a bicubic surface. In order to use it in practice, the 16 unknown coefficients have to be expressed in terms of the 16

known, equally-spaced points. We denote these points

$$\begin{array}{cccc} \mathbf{P}_{03} & \mathbf{P}_{13} & \mathbf{P}_{23} & \mathbf{P}_{33} \\ \mathbf{P}_{02} & \mathbf{P}_{12} & \mathbf{P}_{22} & \mathbf{P}_{32} \\ \mathbf{P}_{01} & \mathbf{P}_{11} & \mathbf{P}_{21} & \mathbf{P}_{31} \\ \mathbf{P}_{00} & \mathbf{P}_{10} & \mathbf{P}_{20} & \mathbf{P}_{30}. \end{array}$$

To calculate the 16 unknown coefficients, we write 16 equations, each based on one of the given points:

$$\begin{array}{cccc} \mathbf{P}(0,0)=\mathbf{P}_{00} & \mathbf{P}(0,1/3)=\mathbf{P}_{01} & \mathbf{P}(0,2/3)=\mathbf{P}_{02} & \mathbf{P}(0,1)=\mathbf{P}_{03} \\ \mathbf{P}(1/3,0)=\mathbf{P}_{10} & \mathbf{P}(1/3,1/3)=\mathbf{P}_{11} & \mathbf{P}(1/3,2/3)=\mathbf{P}_{12} & \mathbf{P}(1/3,1)=\mathbf{P}_{13} \\ \mathbf{P}(2/3,0)=\mathbf{P}_{20} & \mathbf{P}(2/3,1/3)=\mathbf{P}_{21} & \mathbf{P}(2/3,2/3)=\mathbf{P}_{22} & \mathbf{P}(2/3,1)=\mathbf{P}_{23} \\ \mathbf{P}(1,0)=\mathbf{P}_{30} & \mathbf{P}(1,1/3)=\mathbf{P}_{31} & \mathbf{P}(1,2/3)=\mathbf{P}_{32} & \mathbf{P}(1,1)=\mathbf{P}_{33}. \end{array}$$

Solving, substituting the solutions in Equation (I.23), and simplifying produces the *geometric representation* of the bicubic surface

$$\mathbf{P}(u,w)=(u^3,u^2,u,1)\mathbf{N}\begin{pmatrix} \mathbf{P}_{33} & \mathbf{P}_{32} & \mathbf{P}_{31} & \mathbf{P}_{30} \\ \mathbf{P}_{23} & \mathbf{P}_{22} & \mathbf{P}_{21} & \mathbf{P}_{20} \\ \mathbf{P}_{13} & \mathbf{P}_{12} & \mathbf{P}_{11} & \mathbf{P}_{10} \\ \mathbf{P}_{03} & \mathbf{P}_{02} & \mathbf{P}_{01} & \mathbf{P}_{00} \end{pmatrix}\mathbf{N}^T\begin{pmatrix} w^3 \\ w^2 \\ w \\ 1 \end{pmatrix}, \qquad \text{(I.24)}$$

where $\mathbf{N}$ is the Hermite matrix of Equation (I.22).

The surface of Equation (I.24) can now be used to predict the value of a pixel as a polynomial interpolation of 16 of its near neighbors. All that is necessary is to substitute $u=0.5$ and $w=0.5$. The following *Mathematica* code

```
Clear[Nh,P,U,W];
Nh={{-4.5,13.5,-13.5,4.5},{9,-22.5,18,-4.5},
 {-5.5,9,-4.5,1},{1,0,0,0}};
P={{p33,p32,p31,p30},{p23,p22,p21,p20},
 {p13,p12,p11,p10},{p03,p02,p01,p00}};
U={u^3,u^2,u,1};
W={w^3,w^2,w,1};
u:=0.5;
w:=0.5;
Expand[U.Nh.P.Transpose[Nh].Transpose[W]]
```

does that and produces

$$\begin{aligned} &\mathbf{P}(.5,.5) \\ &\quad = 0.00390625\mathbf{P}_{00} - 0.0351563\mathbf{P}_{01} - 0.0351563\mathbf{P}_{02} + 0.00390625\mathbf{P}_{03} \\ &\quad - 0.0351563\mathbf{P}_{10} + 0.316406\mathbf{P}_{11} + 0.316406\mathbf{P}_{12} - 0.0351563\mathbf{P}_{13} \\ &\quad - 0.0351563\mathbf{P}_{20} + 0.316406\mathbf{P}_{21} + 0.316406\mathbf{P}_{22} - 0.0351563\mathbf{P}_{23} \\ &\quad + 0.00390625\mathbf{P}_{30} - 0.0351563\mathbf{P}_{31} - 0.0351563\mathbf{P}_{32} + 0.00390625\mathbf{P}_{33}, \end{aligned}$$

where the 16 coefficients are the ones used in Table 4.118.

◇ **Exercise I.18:** How can this method be used in cases where not all 16 points are known?

◇ **Exercise I.19:** The center point of the surface is calculated as a weighted sum of the 16 equally-spaced data points. It makes sense to assign small weights to points located away from the center, but our result assigns *negative* weights to eight of the 16 points. Explain the meaning of negative weights and show what role they play in interpolating the center of the surface.

Readers who find it hard to follow the details above should compare the way two-dimensional polynomial interpolation is presented here to the way it is discussed by [Press et al. 88]. The following quotation is from page 125: "... The formulas that obtain the c's from the function and derivative values are just a complicated linear transformation, with coefficients which, having been determined once, in the mists of numerical history, can be tabulated and forgotten."

Seated at his disorderly desk, caressed by a counterpane of drifting tobacco haze, he would pore over the manuscript, crossing out, interpolating, re-arguing, and then referring to volumes on his shelves.

Christopher Morley, *The Haunted Bookshop*

Glossary

ACB

A very efficient text compression method by G. Buyanovsky (Section 8.3). It uses a dictionary with unbounded contexts and contents to select the context that best matches the search buffer and the content that best matches the look-ahead buffer.

Adaptive Compression

A compression method that modifies its operations and/or its parameters according to new data read from the input stream. Examples are the adaptive Huffman method of Section 2.9 and the dictionary-based methods of Chapter 3. (See also Semiadaptive Compression, Locally Adaptive Compression.)

Affine Transformations

Two-dimensional or three-dimensional geometric transformations, such as scaling, reflection, rotation, and translation, that preserve parallel lines (Section 4.32.1).

Alphabet

The set of all possible symbols in the input stream. In text compression the alphabet is normally the set of 128 ASCII codes. In image compression it is the set of values a pixel can take (2, 16, 256, or anything else). (See also Symbol.)

Archive

A set of one or more files combined into one file (Section 3.20). The individual members of an archive may be compressed. An archive provides a convenient way of transferring or storing groups of related files. (See also ARC, ARJ.)

ARC

A compression/archival/cataloging program written by Robert A. Freed in the mid 1980s (Section 3.20). It offers good compression and the ability to combine several files into an archive. (See also Archive, ARJ.)

Arithmetic Coding

A statistical compression method (Section 2.14) that assigns one (normally long) code to the entire input stream, instead of assigning codes to the individual symbols. The method reads the input stream symbol by symbol and appends more bits to the code each time a symbol is input and processed. Arithmetic coding is slow, but it compresses at or close to the entropy, even when the symbol probabilities are skewed. (See also Model of Compression, Statistical Methods, QM Coder.)

ARJ

A free compression/archiving utility for MS/DOS (Section 3.21), written by Robert K. Jung to compete with ARC and the various PK utilities. (See also Archive, ARC.)

ASCII Code

The standard character code on all modern computers (although Unicode is becoming a competitor). ASCII stands for American Standard Code for Information Interchange. It is a (1+7)-bit code, meaning 1 parity bit and 7 data bits per symbol. As a result, 128 symbols can be coded (see appendix in book's web page). They include the upper- and lowercase letters, the ten digits, some punctuation marks, and control characters. (See also Unicode.)

Bark

Unit of critical band rate. Named after Heinrich Georg Barkhausen and used in audio applications. The Bark scale is a nonlinear mapping of the frequency scale over the audio range, a mapping that matches the frequency selectivity of the human ear.

Bayesian Statistics

(See Conditional Probability.)

Bi-level Image

An image whose pixels have two different colors. The colors are normally referred to as black and white, "foreground" and "background," or 1 and 0. (See also Bitplane.)

BinHex

A file format for reliable file transfers, designed by Yves Lempereur for use on the Macintosh computer (Section 1.4.3).

Bintrees

A method, somewhat similar to quadtrees, for partitioning an image into nonoverlapping parts. The image is (horizontally) divided into two halves, each half is divided (vertically) into smaller halves, and the process continues recursively, alternating between horizontal and vertical splits. The result is a binary tree where any uniform part of the image becomes a leaf. (See also Prefix Compression, Quadtrees.)

Bitplane

Each pixel in a digital image is represented by several bits. The set of all the kth bits of all the pixels in the image is the kth bitplane of the image. A bi-level image, for example, consists of two bitplanes. (See also Bi-level Image.)

Bitrate

In general, the term "bitrate" refers to both bpb and bpc. In MPEG audio, however, this term is used to indicate the rate at which the compressed stream is read by the decoder. This rate depends on where the stream comes from (such as disk, communications channel, memory). If the bitrate of an MPEG audio file is, e.g., 128Kbps. then the encoder will convert each second of audio into 128K bits of compressed data, and the decoder will convert each group of 128K bits of compressed data into one second of sound. Lower bitrates mean smaller file sizes. However, as the bitrate decreases, the encoder must compress more audio data into fewer bits, eventually resulting in a noticeable loss of audio quality. For CD-quality audio, experience indicates that the best bitrates are in the range of 112Kbps to 160Kbps. (See also Bits/Char.)

Bits/Char

Bits per character (bpc). A measure of the performance in text compression. Also a measure of entropy. (See also Bitrate, Entropy.)

Bits/Symbol

Bits per symbol. A general measure of compression performance.

Block Coding

A general term for image compression methods that work by breaking the image into small blocks of pixels, and encoding each block separately. JPEG (Section 4.6) is a good example, since it processes blocks of 8×8 pixels.

Block Decomposition

A method for lossless compression of discrete-tone images. The method works by searching for, and locating, identical blocks of pixels. A copy B of a block A is compressed by preparing the height, width, and location (image coordinates) of A, and compressing those four numbers by means of Huffman codes. (See also Discrete-Tone Image.)

Block Matching

A lossless image compression method based on the LZ77 sliding window method originally developed for text compression. (See also LZ Methods.)

Block Truncation Coding

BTC is a lossy image compression method that quantizes pixels in an image while preserving the first two or three *statistical moments*. (See also Vector Quantization.)

Burrows-Wheeler Method

This method (Section 8.1) prepares a string of data for later compression. The compression itself is done using the move-to-front method (Section 1.5), perhaps in combination with RLE. The BW method converts a string S to another string L that satisfies two conditions:

1. Any region of L will tend to have a concentration of just a few symbols.
2. It is possible to reconstruct the original string S from L (a little more data may be needed for the reconstruction, in addition to L, but not much).

CALIC

A context-based, lossless image compression method (Section 4.21) whose two main features are (1) the use of three passes in order to achieve symmetric contexts and (2) context quantization, to significantly reduce the number of possible contexts without degrading compression performance.

CCITT

The International Telegraph and Telephone Consultative Committee (Comité Consultatif International Télégraphique et Téléphonique), the old name of the ITU, the International Telecommunications Union, a United Nations organization responsible for developing and recommending standards for data communications (not just compression). (See also ITU.)

Cell Encoding

An image compression method where the entire bitmap is divided into cells of, say, 8×8 pixels each and is scanned cell by cell. The first cell is stored in entry 0 of a table and is encoded (i.e., written on the compressed file) as the pointer 0. Each subsequent cell is searched in the table. If found, its index in the table becomes its code and is written on the compressed file. Otherwise, it is added to the table. In the case of an image made of just straight segments, it can be shown that the table size is just 108 entries.

CIE

CIE is an abbreviation for Commission Internationale de l'Éclairage (International Committee on Illumination). This is the main international organization devoted to light and color. It is responsible for developing standards and definitions in this area. (See Luminance.)

Circular Queue

A basic data structure (Section 3.2.1) that moves data along an array in circular fashion, updating two pointers to point to the start and end of the data in the array.

Codec

A term used to refer to both encoder and decoder.

Codes

A code is a symbol that stands for another symbol. In computer and telecommunications applications, codes are virtually always binary numbers. The ASCII code is the defacto standard, although the new Unicode is used on several new computers and the older EBCDIC is still used on some old IBM computers. (See also ASCII, Unicode.)

Composite and Difference Values

A progressive image method that separates the image into layers using the method of *bintrees*. Early layers consist of a few large, low-resolution blocks, followed by later layers with smaller, higher-resolution blocks. The main principle is to transform a pair of pixels into two values, a composite and a differentiator. (See also Bintrees, Progressive Image Compression.)

Compress

In the large UNIX world, `compress` is commonly used to compress data. This utility uses LZW with a growing dictionary. It starts with a small dictionary of just 512 entries and doubles its size each time it fills up, until it reaches 64K bytes (Section 3.16).

Compression Factor

The inverse of compression ratio. It is defined as

$$\text{compression factor} = \frac{\text{size of the input stream}}{\text{size of the output stream}}.$$

Values greater than 1 mean compression, and values less than 1 imply expansion. (See also Compression Ratio.)

Compression Gain

This measure is defined as

$$100 \log_e \frac{\text{reference size}}{\text{compressed size}},$$

where the reference size is either the size of the input stream or the size of the compressed stream produced by some standard lossless compression method.

Compression Ratio

One of several measures that are commonly used to express the efficiency of a compression method. It is the ratio

$$\text{compression ratio} = \frac{\text{size of the output stream}}{\text{size of the input stream}}.$$

A value of 0.6 means that the data occupies 60% of its original size after compression. Values greater than 1 mean an output stream bigger than the input stream (negative compression).

Sometimes the quantity $100 \times (1 - \text{compression ratio})$ is used to express the quality of compression. A value of 60 means that the output stream occupies 40% of its original size (or that the compression has resulted in a savings of 60%). (See also Compression Factor.)

Conditional Image RLE

A compression method for grayscale images with n shades of gray. The method starts by assigning an n-bit code to each pixel depending on its near neighbors. It then concatenates the n-bit codes into a long string, and calculates run lengths. The run lengths are encoded by prefix codes. (See also RLE, Relative Encoding.)

Conditional Probability

We tend to think of probability as something that is built into an experiment. A true die, for example, has probability of 1/6 of falling on any side, and we tend to consider this an intrinsic feature of the die. Conditional probability is a different way of looking at probability. It says that knowledge affects probability. The main task of this field is to calculate the probability of an event A given that another event, B, is known to have occurred. This is the conditional probability of A (more precisely, the probability of A conditioned on B), and it is denoted by $P(A|B)$. The field of conditional probability is sometimes called *Bayesian statistics*, since it was first developed by the Reverend Thomas Bayes [1702–1761], who came up with the basic formula of conditional probability.

Context

The N symbols preceding the next symbol. A context-based model uses context to assign probabilities to symbols.

Context-Free Grammars

A formal language uses a small number of symbols (called *terminal symbols*) from which valid sequences can be constructed. Any valid sequence is finite, the number of valid sequences is normally unlimited, and the sequences are constructed according to certain rules (sometimes called *production rules*). The rules can be used to construct valid sequences and also to determine whether a given sequence is valid. A production rule consists of a nonterminal symbol on the left and a string of terminal and nonterminal symbols on the right. The nonterminal symbol on the left becomes the name of the string on the right. The set of production rules constitutes the grammar of the formal language. If the production rules do not depend on the context of a symbol, the grammar is context-free. There are also context-sensitive grammars. The sequitur method of Section 8.10 uses context-free grammars.

Context-Tree Weighting

A method for the compression of bit strings. It can be applied to text and images, but they have to be carefully converted to bit strings. The method constructs a context tree where bits input in the immediate past (context) are used to estimate the probability of the current bit. The current bit and its estimated probability are then sent to an arithmetic encoder, and the tree is updated to include the current bit in the context. (See also KT Probability Estimator.)

Continuous-Tone Image

A digital image with a large number of colors, such that adjacent image areas with colors that differ by just one unit appear to the eye as having continuously varying colors. An example is an image with 256 grayscale values. When adjacent pixels in such an image have consecutive gray levels, they appear to the eye as a continuous variation of the gray level. (See also Bi-level image, Discrete-Tone Image, Grayscale Image.)

Continuous Wavelet Transform

An important modern method for analyzing the time and frequency contents of a function $f(t)$ by means of a wavelet. The wavelet is itself a function (which has to satisfy certain conditions), and the transform is done by multiplying the wavelet and $f(t)$ and calculating the integral of the product. The wavelet is then translated and the process is repeated. When done, the wavelet is scaled, and the entire process is carried out again in order to analyze $f(t)$ at a different scale. (See also Discrete Wavelet Transform, Lifting Scheme, Multiresolution Decomposition, Taps.)

Convolution

A way to describe the output of a linear, shift-invariant system by means of its input.

Correlation

A statistical measure of the linear relation between two paired variables. The values of R range from -1 (perfect negative relation), to 0 (no relation), to $+1$ (perfect positive relation).

CRC

CRC stands for *Cyclical Redundancy Check* (or *Cyclical Redundancy Code*). It is a rule that shows how to obtain vertical check bits from all the bits of a data stream (Section 3.23). The idea is to generate a code that depends on all the bits of the data stream, and use it to detect errors (bad bits) when the data is transmitted (or when it is stored and retrieved).

CRT

A CRT (cathode ray tube) is a glass tube with a familiar shape. In the back it has an electron gun (the cathode) that emits a stream of electrons. Its front surface is positively charged, so it attracts the electrons (which have a negative electric charge). The front is coated with a phosphor compound that converts the kinetic energy of the electrons hitting it to light. The flash of light lasts only a fraction of a second, so in order to get a constant display, the picture has to be refreshed several times a second.

Data Compression Conference

A meeting of researchers and developers in the area of data compression. The DCC takes place every year in Snowbird, Utah, USA. It lasts three days and the next few meetings are scheduled for late March.

Data Structure

A set of data items used by a program and stored in memory such that certain operations (for example, finding, adding, modifying, and deleting items) can be performed on the data items fast and easily. The most common data structures are the array, stack, queue, linked list, tree, graph, and hash table. (See also Circular Queue.)

Decibel

A logarithmic measure that can be used to measure any quantity that takes values over a very wide range. A common example is sound intensity. The intensity (amplitude) of sound can vary over a range of 11–12 orders of magnitude. Instead of using a linear measure, where numbers as small as 1 and as large as 10^{11} would be needed, a logarithmic scale is used, where the range of values is $[0, 11]$.

Decoder

A decompression program (or algorithm).

Dictionary-Based Compression

Compression methods (Chapter 3) that save pieces of the data in a "dictionary" data structure (normally a tree). If a string of new data is identical to a piece already saved in the dictionary, a pointer to that piece is output to the compressed stream. (See also LZ Methods.)

Differential Image Compression

A lossless image compression method where each pixel p is compared to a *reference pixel*, which is one of its immediate neighbors, and is then encoded in two parts: a prefix, which is the number of most significant bits of p that are identical to those of the reference pixel, and a suffix, which is (almost all) the remaining least significant bits of p. (See also DPCM.)

Digital Video

Digital video is a form of video in which the original image is generated, in the camera, in the form of pixels. (See also High-Definition Television.)

Digram

A pair of consecutive symbols.

Discrete Cosine Transform

A variant of the discrete Fourier transform (DFT) that produces just real numbers. The DCT (Sections 4.4.3 and 4.6.2) transforms a set of numbers by combining n numbers to become an n-dimensional point and rotating it in n-dimensions such that the first coordinate becomes dominant. The DCT and its inverse, the IDCT, are used in JPEG (Section 4.6) to compress an image with acceptable loss, by isolating the high-frequency components of an image, so that they can later be quantized. (See also Fourier Transform, Transform.)

Discrete-Tone Image

A discrete-tone image may be bi-level, grayscale, or color. Such images are (with few exceptions) artificial, having been obtained by scanning a document, or grabbing a computer screen. The pixel colors of such an image do not vary continuously or smoothly, but have a small set of values, such that adjacent pixels may differ much in intensity or color. Figure G.5 is an example of such an image. (See also Block Decomposition, Continuous-Tone Image.)

Discrete Wavelet Transform

The discrete version of the continuous wavelet transform. A wavelet is represented by means of several filter coefficients, and the transform is carried out by matrix multiplication (or a simpler version thereof) instead of by calculating an integral. (See also Continuous Wavelet Transform, Multiresolution Decomposition.)

Dithering

A technique for printing or displaying a grayscale image on a bi-level output device, such as a monochromatic screen or a black and white printer. The tradeoff is loss of image detail. (See also Halftoning.)

DjVu

Certain images combine the properties of all three image types (bi-level, discrete-tone, and continuous-tone). An important example of such an image is a scanned document containing text, line drawings, and regions with continuous-tone pictures, such as paintings or photographs. DjVu (pronounced "déjà vu"), is designed for high compression and fast decompression of such documents.

It starts by decomposing the document into three components: mask, foreground, and background. The background component contains the pixels that constitute the pictures and the paper background. The mask contains the text and the lines in bi-level form (i.e., one bit per pixel). The foreground contains the color of the mask pixels. The background is a continuous-tone image and can be compressed at the low resolution of 100 dpi. The foreground normally contains large uniform areas and is also compressed as a continuous-tone image at the same low resolution. The mask is left at 300 dpi but can be efficiently compressed, since it is bi-level. The background and foreground are compressed with a wavelet-based method called IW44, while the mask is compressed with JB2, a version of JBIG2 (Section 4.10) developed at AT&T.

DPCM

DPCM compression is a member of the family of differential encoding compression methods, which itself is a generalization of the simple concept of relative encoding (Section 1.3.1). It is based on the fact that neighboring pixels in an image (and also adjacent samples in digitized sound) are correlated. (See also Differential Image Compression, Relative Encoding.)

Embedded Coding

This feature is defined as follows: Imagine that an image encoder is applied twice to the same image, with different amounts of loss. It produces two files, a large one of size M and a small one of size m. If the encoder uses embedded coding, the smaller file is identical to the first m bits of the larger file.

The following example aptly illustrates the meaning of this definition. Suppose that three users wait for you to send them a certain compressed image, but they need different image qualities. The first one needs the quality contained in a 10 Kb file. The image qualities required by the second and third users are contained in files of sizes 20 Kb and 50 Kb, respectively. Most lossy image compression methods would have to compress the same image three times, at different qualities, to generate three files with the right sizes. An embedded encoder, on the other hand, produces one file, and then three chunks—of lengths 10 Kb, 20 Kb, and 50 Kb, all starting at the beginning of the file—can be sent to the three users, satisfying their needs. (See also SPIHT, EZW.)

Encoder

A compression program (or algorithm).

Entropy

The entropy of a single symbol a_i is defined (in Section 2.1) as $-P_i \log_2 P_i$, where P_i is the probability of occurrence of a_i in the data. The entropy of a_i is the smallest number of bits needed, on average, to represent symbol a_i. Claude Shannon, the creator of information theory, coined the term *entropy* in 1948, since this term is used in thermodynamics to indicate the amount of disorder in a physical system. (See also Entropy Encoding, Information Theory.)

Entropy Encoding

A lossless compression method where data can be compressed such that the average number of bits/symbol approaches the entropy of the input symbols. (See also Entropy.)

Error-Correcting Codes

The opposite of data compression, these codes (appendix in the book's web page) detect and correct errors in digital data by increasing the redundancy of the data. They use check bits or parity bits, and are sometimes designed with the help of generating polynomials.

EXE Compressor

A compression program for compressing EXE files on the PC. Such a compressed file can be decompressed and executed with one command. The original EXE compressor is LZEXE, by Fabrice Bellard (Section 3.22).

EZW

A progressive, embedded image coding method based on the zerotree data structure. It has largely been superseded by the more efficient SPIHT method. (See also SPIHT, Progressive Image Compression, Embedded Coding.)

Facsimile Compression

Transferring a typical page between two fax machines can take up to 10–11 minutes without compression, This is why the ITU has developed several standards for compression of facsimile data. The current standards (Section 2.13) are T4 and T6, also called Group 3 and Group 4, respectively. (See also ITU.)

FELICS

A Fast, Efficient, Lossless Image Compression method designed for grayscale images that competes with the lossless mode of JPEG. The principle is to code each pixel with a variable-size code based on the values of two of its previously seen neighbor pixels. Both the unary code and the Golomb code are used. There is also a progressive version of FELICS (Section 4.18). (See also Progressive FELICS.)

FHM Curve Compression

A method for compressing curves. The acronym FHM stands for Fibonacci, Huffman, and Markov. (See also Fibonacci Numbers.)

Fibonacci Numbers

A sequence of numbers defined by

$$F_1 = 1, \quad F_2 = 1, \quad F_i = F_{i-1} + F_{i-2}, \quad i = 3, 4, \ldots.$$

The first few numbers in the sequence are 1, 1, 2, 3, 5, 8, 13, and 21. These numbers have many applications in mathematics and in various sciences. They are also found in nature, and are related to the golden ratio. (See also FHM Curve Compression.)

Fourier Transform

A mathematical transformation that produces the frequency components of a function (Section 5.1). The Fourier transform shows how a periodic function can be written as the sum of sines and cosines, thereby showing explicitly the frequencies "hidden" in the original representation of the function. (See also Discrete Cosine Transform, Transform.)

Gaussian Distribution

(See Normal Distribution.)

GFA

A compression method originally developed for bi-level images that can also be used for color images. GFA uses the fact that most images of interest have a certain amount of self similarity (i.e., parts of the image are similar, up to size, orientation, or brightness, to the entire image or to other parts). GFA partitions the image into subsquares using a quadtree, and expresses relations between parts of the image in a graph. The graph is similar to graphs used to describe finite-state automata. The method is lossy, since parts of a real image may be very similar to other parts. (See also Quadtrees, Resolution Independent Compression, WFA.)

GIF

An acronym that stands for Graphics Interchange Format. This format (Section 3.17) was developed by Compuserve Information Services in 1987 as an efficient, compressed graphics file format that allows for images to be sent between computers. The original version of GIF is known as GIF 87a. The current standard is GIF 89a. (See also Patents.)

Golomb Code

A way to generate a variable-size code for integers n (Section 2.4). It depends on the choice of a parameter b and it is created in two steps:

1. Compute the two quantities

$$q = \left\lfloor \frac{n-1}{b} \right\rfloor, \qquad r = n - qb - 1.$$

2. Construct the Golomb code of n in two parts; the first is the value of $q+1$, coded in unary (exercise 2.5), and the second, the binary value of r coded in either $\lfloor \log_2 b \rfloor$ bits (for the small remainders) or in $\lceil \log_2 b \rceil$ bits (for the large ones). (See also Unary Code.)

Gray Codes

These are binary codes for the integers, where the codes of consecutive integers differ by one bit only. Such codes are used when a grayscale image is separated into bitplanes, each a bi-level image. (See also Grayscale Image,)

Grayscale Image

A continuous-tone image with shades of a single color. (See also Continuous-Tone Image.)

Growth Geometry Coding

A method for progressive lossless compression of bi-level images. The method selects some *seed* pixels and applies geometric rules to grow each seed pixel into a pattern of pixels. (See also Progressive Image Compression.)

GZip

Popular software that implements the so-called "deflation" algorithm (Section 3.19) that uses a variation of LZ77 combined with static Huffman coding. It uses a 32 Kb-long sliding dictionary, and a look-ahead buffer of 258 bytes. When a string is not found in the dictionary, it is emitted as a sequence of literal bytes. (See also Zip.)

H.261

In late 1984, the CCITT (currently the ITU-T) organized an expert group to develop a standard for visual telephony for ISDN services. The idea was to send images and sound between special terminals, so that users could talk and see each other. This type of application requires sending large amounts of data, so compression became an important consideration. The group eventually came up with a

number of standards, known as the H series (for video) and the G series (for audio) recommendations, all operating at speeds of $p\times64$ Kbit/s for $p = 1, 2, \ldots, 30$. These standards are known today under the umbrella name of $p\times64$.

Halftoning

A method for the display of gray scales in a bi-level image. By placing groups of black and white pixels in carefully designed patterns, it is possible to create the effect of a gray area. The tradeoff of halftoning is loss of resolution. (See also Bi-level Image, Dithering.)

Hamming Codes

A type of error-correcting code for 1-bit errors, where it is easy to generate the required parity bits.

Hierarchical Progressive Image Compression

An image compression method (or an optional part of such a method) where the encoder writes the compressed image in layers of increasing resolution. The decoder decompresses the lowest-resolution layer first, displays this crude image, and continues with higher-resolution layers. Each layer in the compressed stream uses data from the preceding layer. (See also Progressive Image Compression.)

High-Definition Television

A general name for several standards that are currently replacing traditional television. HDTV uses digital video, high-resolution images, and aspect ratios different from the traditional 3:4. (See also Digital Video.)

Huffman Coding

A popular method for data compression (Section 2.8). It assigns a set of "best" variable-size codes to a set of symbols based on their probabilities. It serves as the basis for several popular programs used on personal computers. Some of them use just the Huffman method, while others use it as one step in a multistep compression process. The Huffman method is somewhat similar to the Shannon-Fano method. It generally produces better codes, and like the Shannon-Fano method, it produces best code when the probabilities of the symbols are negative powers of 2. The main difference between the two methods is that Shannon-Fano constructs its codes top to bottom (from the leftmost to the rightmost bits), while Huffman constructs a code tree from the bottom up (builds the codes from right to left). (See also Shannon-Fano Coding, Statistical Methods.)

Information Theory

A mathematical theory that quantifies information. It shows how to measure information, so that one can answer the question; How much information is included in this piece of data? with a precise number! Information theory is the creation, in 1948, of Claude Shannon, of Bell labs. (See also Entropy.)

Interpolating Polynomials

Given two numbers a and b we know that $m = 0.5a + 0.5b$ is their average, since it is located midway between a and b. We say that the average is an *interpolation* of the two numbers. Similarly, the weighted sum $0.1a+0.9b$ represents an interpolated value located 10% away from b and 90% away from a. Extending this concept to points (in two or three dimensions) is done by means of *interpolating polynomials* (see the book's web page). Given a set of points we start by fitting a parametric polynomial $\mathbf{P}(t)$ or $\mathbf{P}(u, w)$ through them. Once the polynomial is known, it can be used to calculate interpolated points by computing $\mathbf{P}(0.5)$, $\mathbf{P}(0.1)$, or other values.

ISO

The International Standards Organization. This is one of the organizations responsible for developing standards. Among other things it is responsible (together with the ITU) for the JPEG and MPEG compression standards. (See also ITU, CCITT, MPEG.)

Iterated Function Systems (IFS)

An image compressed by IFS is uniquely defined by a few affine transformations (Section 4.32.1). The only rule is that the scale factors of these transformations must be less than 1 (shrinking). The image is saved in the output stream by writing the sets of six numbers that define each transformation. (See also Affine Transformations, Resolution Independent Compression.)

ITU

The International Telecommunications Union, the new name of the CCITT, is a United Nations organization responsible for developing and recommending standards for data communications (not just compression). (See also CCITT.)

JBIG

A special-purpose compression method (Section 4.9) developed specifically for progressive compression of bi-level images. The name JBIG stands for Joint Bi-Level Image Processing Group. This is a group of experts from several international organizations, formed in 1988 to recommend such a standard. JBIG uses multiple arithmetic coding and a resolution-reduction technique to achieve its goals. (See also Bi-level Image, JBIG2.)

JBIG2

A recent international standard for the compression of bi-level images. It is intended to replace the original JBIG. Its main features are

1. Large increases in compression performance (typically 3–5 times better than Group 4/MMR, and 2–4 times better than JBIG).
2. Special compression methods for text, halftones, and other bi-level image parts.
3. Lossy and lossless compression.
4. Two modes of progressive compression. Mode 1 is quality-progressive compression, where the decoded image progresses from low to high quality. Mode 2

is content-progressive coding, where important image parts (such as text) are decoded first, followed by less important parts (such as halftone patterns).
5. Multipage document compression.
6. Flexible format, designed for easy embedding in other image file formats, such as TIFF.
7. Fast decompression. Using some coding modes, images can be decompressed at over 250 million pixels/second in software.

(See also Bi-level Image, JBIG.)

JFIF

The full name of this method (Section 4.6.9) is JPEG File Interchange Format. It is a graphics file format that makes it possible to exchange JPEG-compressed images between different computers. The main features of JFIF are the use of the YCbCr triple-component color space for color images (only one component for grayscale images) and the use of a *marker* to specify features missing from JPEG, such as image resolution, aspect ratio, and features that are application-specific.

JPEG

A sophisticated lossy compression method (Section 4.6) for color or grayscale still images (not movies). It also works best on continuous-tone images, where adjacent pixels have similar colors. One advantage of JPEG is the use of many parameters, allowing the user to adjust the amount of data loss (and thus also the compression ratio) over a very wide range. There are two main modes: lossy (also called baseline) and lossless (which typically gives a 2:1 compression ratio). Most implementations support just the lossy mode. This mode includes progressive and hierarchical coding.

The main idea behind JPEG is that an image exists for people to look at, so when the image is compressed, it is acceptable to lose image features to which the human eye is not sensitive.

The name JPEG is an acronym that stands for Joint Photographic Experts Group. This was a joint effort by the CCITT and the ISO that started in June 1987. The JPEG standard has proved successful and has become widely used for image presentation, especially in Web pages. (See also JPEG-LS, MPEG.)

JPEG-LS

The lossless mode of JPEG is inefficient and often is not even implemented. As a result, the ISO decided to develop a new standard for the lossless (or near-lossless) compression of continuous-tone images. The result became popularly known as JPEG-LS. This method is not simply an extension or a modification of JPEG. It is a new method, designed to be simple and fast. It does not use the DCT, does not use arithmetic coding, and uses quantization in a limited way, and only in its near-lossless option. JPEG-LS examines several of the previously seen neighbors of the current pixel, uses them as the *context* of the pixel, uses the context to predict the pixel and to select a probability distribution out of several such distributions, and uses that distribution to encode the prediction error with a special Golomb code. There is also a run mode, where the length of a run of identical pixels is encoded. (See also Golomb Code, JPEG.)

Kraft-MacMillan Inequality

A relation (Section 2.5) that says something about unambiguous variable-size codes. Its first part states; given an unambiguous variable-size code, with n codes of sizes L_i, then

$$\sum_{i=1}^{n} 2^{-L_i} \le 1.$$

(This is Equation(2.1).) The second part states the opposite, namely, given a set of n positive integers $(L_1, L_2, \ldots, L_n)$ that satisfy Equation (2.1), there exists an unambiguous variable-size code such that L_i are the sizes of its individual codes. Together, both parts say that a code is unambiguous if and only if it satisfies relation (2.1).

KT Probability Estimator

A method to estimate the probability of a bit string containing a zeros and b ones. It is due to Krichevsky and Trofimov. (See also Context-Tree Weighting.)

L Systems

Lindenmayer systems (or L-systems for short) were developed by the biologist Aristid Lindenmayer in 1968 as a tool [Lindenmayer 68] to describe the morphology of plants. They were initially used in computer science, in the 1970s, as a tool to define formal languages, but have become really popular only after 1984, when it became apparent that they can be used to draw many types of fractals, in addition to their use in botany (see the book's web page).

Laplace Distribution

A probability distribution similar to the normal (Gaussian) distribution, but narrower and sharply peaked. The general Laplace distribution with variance V and mean m is given by

$$L(V, x) = \frac{1}{\sqrt{2V}} \exp\left(-\sqrt{\frac{2}{V}}|x - m|\right).$$

Experience seems to suggest that the values of pixels in many images are Laplace distributed, which is why this distribution is used in some image compression methods, such as MLP. (See also Normal Distribution.)

Laplacian Pyramid

A progressive image compression technique where the original image is transformed to a set of difference images that can later be decompressed and displayed as a small, blurred image that becomes increasingly sharper. (See also Progressive Image Compression.)

LHArc

This method (Section 3.21) is by Haruyasu Yoshizaki. Its predecessor is LHA, designed jointly by Haruyasu Yoshizaki and Haruhiko Okumura. These methods use adaptive Huffman coding with features drawn from LZSS.

Lifting Scheme

A method for calculating the discrete wavelet transform in place, so no extra memory is required. (See also Discrete Wavelet Transform.)

Locally Adaptive Compression

A compression method that adapts itself to local conditions in the input stream, and changes this adaptation as it moves from area to area in the input. An example is the move-to-front method of Section 1.5. (See also Adaptive Compression, Semiadaptive Compression.)

Luminance

This quantity is defined by the CIE (Section 4.6.1) as radiant power weighted by a spectral sensitivity function that is characteristic of vision. (See also CIE.)

Lossless Compression

A compression method where the output of the decoder is identical to the original data compressed by the encoder. (See also Lossy Compression.)

Lossy Compression

A compression method where the output of the decoder is different from the original data compressed by the encoder, but is nevertheless acceptable to a user. Such methods are common in image and audio compression, but not in text compression, where the loss of even one character may result in ambiguous or incomprehensible text. (See also Lossless Compression, Subsampling.)

LZ Methods

All dictionary-based compression methods are based on the work of J. Ziv and A. Lempel, published in 1977 and 1978. Today, these are called LZ77 and LZ78 methods, respectively. Their ideas have been a source of inspiration to many researchers, who generalized, improved, and combined them with RLE and statistical methods to form many commonly used adaptive compression methods, for text, images, and audio. (See also Block Matching, Dictionary-Based Compression, Sliding-Window Compression.)

LZAP

The LZAP method (Section 3.12) is an LZW variant based on the following idea: Instead of just concatenating the last two phrases and placing the result in the dictionary, place all prefixes of the concatenation in the dictionary. The suffix AP stands for All Prefixes.

LZFG

This is the name of several related methods (Section 3.7) that are hybrids of LZ77 and LZ78. They were developed by Edward Fiala and Daniel Greene. All these methods are based on the following scheme. The encoder produces a compressed file with tokens and literals (raw ASCII codes) intermixed. There are two types of tokens, a *literal* and a *copy*. A literal token indicates that a string of literals follow, a copy token points to a string previously seen in the data. (See also LZ Methods, Patents.)

LZMW

A variant of LZW, the LZMW method (Section 3.11) works as follows: Instead of adding I plus one character of the next phrase to the dictionary, add I plus the entire next phrase to the dictionary. (See also LZW.)

LZP

An LZ77 variant developed by C. Bloom (Section 3.14). It is based on the principle of context prediction that says "if a certain string '`abcde`' has appeared in the input stream in the past and was followed by '`fg...`', then when '`abcde`' appears again in the input stream, there is a good chance that it will be followed by the same '`fg...`'." (See also Context.)

LZSS

This version of LZ77 (Section 3.3) was developed by Storer and Szymanski in 1982 [Storer 82]. It improves on the basic LZ77 in three ways: (1) it holds the look-ahead buffer in a circular queue, (2) It holds the search buffer (the dictionary) in a binary search tree, and (3) it creates tokens with two fields instead of three. (See also LZ Methods.)

LZW

This is a popular variant (Section 3.10) of LZ78, developed by Terry Welch in 1984. Its main feature is eliminating the second field of a token. An LZW token consists of just a pointer to the dictionary. As a result, such a token always encodes a string of more than one symbol. (See also Patents.)

LZY

LZY (Section 3.13) is an LZW variant that adds one dictionary string per input character and increments strings by one character at a time.

MLP

A progressive compression method for grayscale images. An image is compressed in levels. A pixel is predicted by a symmetric pattern of its neighbors from preceding levels, and the prediction error is arithmetically encoded. The Laplace distribution is used to estimate the probability of the error. (See also Laplace Distribution, Progressive FELICS.)

MNP5, MNP7

These have been developed by Microcom, Inc., a maker of modems, for use in its modems. MNP5 (Section 2.10) is a two-stage process that starts with run length encoding, followed by adaptive frequency encoding. MNP7 (Section 2.11) combines run length encoding with a two-dimensional variant of adaptive Huffman coding.

Model of Compression

A model is a method to "predict" (to assign probabilities to) the data to be compressed. This concept is important in statistical data compression. When a statistical method is used, a model for the data has to be constructed before compression

can begin. A simple model can be built by reading the entire input stream, counting the number of times each symbol appears (its frequency of occurrence), and computing the probability of occurrence of each symbol. The data stream is then input again, symbol by symbol, and is compressed using the information in the probability model. (See also Statistical Methods, Statistical Model.)

One feature of arithmetic coding is that it is easy to separate the statistical model (the table with frequencies and probabilities) from the encoding and decoding operations. It is easy to encode, for example, the first half of a data stream using one model, and the second half using another model.

Move-to-Front Coding

The basic idea behind this method (Section 1.5) is to maintain the alphabet A of symbols as a list where frequently occurring symbols are located near the front. A symbol `s` is encoded as the number of symbols that precede it in this list. After symbol `s` is encoded, it is moved to the front of list A.

MPEG

This acronym stands for Moving Pictures Experts Group. The MPEG standard consists of several methods for the compression of movies, including the compression of digital images and digital sound, as well as synchronization of the two. There currently are several MPEG standards. MPEG-1 is intended for intermediate data rates, on the order of 1.5 Mbit/s. MPEG-2 is intended for high data rates of at least 10 Mbit/s. MPEG-3 was intended for HDTV compression but was found to be redundant and was merged with MPEG-2. MPEG-4 is intended for very low data rates of less than 64 Kbit/s. A third international body, the ITU-T, has been involved in the design of both MPEG-2 and MPEG-4. A working group of the ISO is still at work on MPEG. (See also ISO, JPEG.)

Multiresolution Decomposition

This method groups all the discrete wavelet transform coefficients for a given scale, displays their superposition, and repeats for all scales. (See also Continuous Wavelet Transform, Discrete Wavelet Transform.)

Multiresolution Image

A compressed image that may be decompressed at any resolution. (See also Resolution Independent Compression, Iterated Function Systems, WFA.)

Normal Distribution

A probability distribution with the typical bell shape. It is found in many places in both theoretical and real-life models. The normal distribution with mean m and standard deviation s is defined by

$$f(x) = \frac{1}{s\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right\}.$$

Patents

A mathematical algorithm can be patented if it is intimately associated with software or firmware implementing it. Several compression methods, most notably LZW, have been patented (Section 3.25), creating difficulties for software developers who work with GIF, UNIX `compress`, or any other system that uses LZW. (See also GIF, LZW, Compress.)

Pel

The smallest unit of a facsimile image; a dot. (See also Pixel.)

Phrase

A piece of data placed in a dictionary to be used in compressing future data. The concept of phrase is central in dictionary-based data compression methods since the success of such a method depends a lot on how it selects phrases to be saved in its dictionary. (See also Dictionary-Based Compression, LZ Methods.)

Pixel

The smallest unit of a digital image; a dot. (See also Pel.)

PKZip

A compression program for MS/DOS (Section 3.20) written by Phil Katz who has founded the PKWare company (`http://www.pkware.com`), which also markets the PKunzip, PKlite, and PKArc software.

Prediction

Assigning probabilities to symbols. (See also PPM.)

Prefix Compression

In an image of size $2^n \times 2^n$ each pixel can be assigned a $2n$-bit number using quadtree methods. The prefix compression method shows how to select a *prefix* value P. Once P has been selected, the method finds all the pixels whose numbers have the same P leftmost bits (same prefix). Those pixels are compressed by writing the prefix once on the compressed stream, followed by all the suffixes.

Prefix Property

One of the principles of variable-size codes. It states; Once a certain bit pattern has been assigned as the code of a symbol, no other codes should start with that pattern (the pattern cannot be the *prefix* of any other code). Once the string 1, for example, is assigned as the code of a_1, no other codes should start with 1 (i.e., they all have to start with 0). Once 01, for example, is assigned as the code of a_2, no other codes can start with 01 (they all should start with 00). (See also Variable-Size Codes, Statistical Methods.)

Progressive FELICS

A progressive version of FELICS where pixels are encoded in levels. Each level doubles the number of pixels encoded. To decide what pixels are included in a certain level, the preceding level can conceptually be rotated 45° and scaled by $\sqrt{2}$ in both dimensions. (See also FELICS, MLP, Progressive Image Compression.)

Progressive Image Compression

An image compression method where the compressed stream consists of "layers," where each layer contains more detail of the image. The decoder can very quickly display the entire image in a low-quality format, then improve the display quality as more and more layers are being read and decompressed. A user watching the decompressed image develop on the screen can normally recognize most of the image features after only 5–10% of it has been decompressed. Improving image quality over time can be done by (1) sharpening it, (2) adding colors, or (3) adding resolution. (See also Progressive FELICS, Hierarchical Progressive Image Compression, MLP, JBIG.)

PPM

A compression method that assigns probabilities to symbols based on the context (long or short) in which they appear. (See also Prediction, PPPM.)

PPPM

A lossless compression method for grayscale (and color) images that assigns probabilities to symbols based on the Laplace distribution, like MLP. Different contexts of a pixel are examined and their statistics used to select the mean and variance for a particular Laplace distribution. (See also Laplace Distribution, Prediction, PPM, MLP.)

Prefix Compression

A variant of quadtrees, designed for bi-level images with text or diagrams, where the number of black pixels is relatively small. Each pixel in a $2^n \times 2^n$ image is assigned an n-digit, or $2n$-bit, number based on the concept of quadtrees. Numbers of adjacent pixels tend to have the same prefix (most-significant bits), so the common prefix and different suffixes of a group of pixels are compressed separately. (See also Quadtrees.)

Psychoacoustic Model

A mathematical model of the sound masking properties of the human auditory (ear brain) system.

QIC-122 Compression

An LZ77 variant that has been developed by the QIC organization for text compression on 1/4-inch data cartridge tape drives.

QM Coder

This is the arithmetic coder of JPEG and JBIG. It is designed for simplicity and speed, so it is limited to input symbols that are single bits and it uses an approximation instead of a multiplication. It also uses fixed-precision integer arithmetic, so it has to resort to *renormalization* of the probability interval from time to time, in order for the approximation to remain close to the true multiplication. (See also Arithmetic Coding.)

Quadrisection

This is a relative of the quadtree method. It assumes that the original image is a $2^k \times 2^k$ square matrix M_0, and it constructs matrices M_1, M_2,...,M_{k+1} with fewer and fewer columns. These matrices naturally have more and more rows, and quadrisection achieves compression by searching for and removing duplicate rows. Two closely related variants of quadrisection are bisection and octasection (See also Quadtrees.)

Quadtrees

This is a data compression method for bitmap images. A quadtree (Section 4.27) is a tree where each leaf corresponds to a uniform part of the image (a quadrant, subquadrant, or a single pixel) and each interior node has exactly four children. (See also Bintrees, Prefix Compression, Quadrisection.)

Quaternary

A base-4 digit. It can be 0, 1, 2, or 3.

Relative Encoding

A variant of RLE, sometimes called *differencing* (Section 1.3.1). It is used in cases where the data to be compressed consists of a string of numbers that don't differ by much, or in cases where it consists of strings that are similar to each other. The principle of relative encoding is to send the first data item a_1 followed by the differences $a_{i+1} - a_i$. (See also DPCM, RLE.)

Reliability

Variable-size codes and other codes are vulnerable to errors. In cases where reliable transmission of codes is important, the codes can be made more reliable by adding check bits, parity bits, or CRC (Section 2.12 and Appendix in the book's web page). Notice that reliability is, in a sense, the opposite of data compression, since it is done by increasing redundancy. (See also CRC.)

Resolution Independent Compression

An image compression method that does not depend on the resolution of the specific image being compressed. The image can be decompressed at any resolution. (See also Multiresolution Images, Iterated Function Systems, WFA.)

RLE

A general name for methods that compress data by replacing a run length of identical symbols with one code, or token, containing the symbol and the length of the run. RLE sometimes serves as one step in a multistep statistical or dictionary-based method. (See also Relative Encoding, Conditional Image RLE.)

Scalar Quantization

The dictionary definition of the term "quantization" is "to restrict a variable quantity to discrete values rather than to a continuous set of values." If the data to be compressed is in the form of large numbers, quantization is used to convert them

to small numbers. This results in (lossy) compression. If the data to be compressed is analog (e.g., a voltage that changes with time), quantization is used to digitize it into small numbers. This aspect of quantization is used by several audio compression methods. (See also Vector Quantization.)

SemiAdaptive Compression

A compression method that uses a two-pass algorithm, where the first pass reads the input stream to collect statistics on the data to be compressed, and the second pass performs the actual compression. The statistics (model) are included in the compressed stream. (See also Adaptive Compression, Locally Adaptive Compression.)

Semi-Structured text

Such text is defined as data that is human readable and also suitable for machine processing. A common example is HTML. The sequitur method of Section 8.10 performs especially well on such text.

Shannon-Fano Coding

An early algorithm for finding a minimum-length variable-size code given the probabilities of all the symbols in the data (Section 2.6). This method was later superseded by the Huffman method. (See also Statistical Methods, Huffman Coding.)

Simple Image

A simple image is one that uses a small fraction of the possible grayscale values or colors available to it. A common example is a bi-level image where each pixel is represented by eight bits. Such an image uses just two colors out of a palette of 256 possible colors. Another example is a grayscale image scanned from a bi-level image. Most pixels will be black or white, but some pixels may have other shades of gray. A cartoon is also an example of a simple image (especially a cheap cartoon, where just a few colors are used). A typical cartoon consists of uniform areas, so it may use a small number of colors out of a potentially large palette. The EIDAC method of Section 4.11 is especially designed for simple images.

Sliding Window Compression

The LZ77 method (Section 3.2) uses part of the previously seen input stream as the dictionary. The encoder maintains a window to the input stream, and shifts the input in that window from right to left as strings of symbols are being encoded. The method is thus based on a *sliding window*. (See also LZ Methods.)

Space-Filling Curves

A space-filling curve (Section 4.29) is a function $\mathbf{P}(t)$ that goes through every point in a given two-dimensional area, normally the unit square. Such curves are defined recursively and are used in image compression.

Sparse Strings

Regardless of what the input data represents—text, binary, images, or anything else—we can think of the input stream as a string of bits. If most of the bits are zeros, the string is called *sparse*. Sparse strings can be compressed very efficiently by specially designed methods (Section 8.5).

SPIHT

A progressive image encoding method that efficiently encodes the image after it has been transformed by any wavelet filter. SPIHT is embedded, progressive, and has a natural lossy option. It is also simple to implement, fast, and produces excellent results for all types of images. (See also EZW, Progressive Image Compression, Embedded Coding, Discrete Wavelet Transform.)

Statistical Methods

These methods (Chapter 2) work by assigning variable-size codes to symbols in the data, with the shorter codes assigned to symbols or groups of symbols that appear more often in the data (have a higher probability of occurrence). (See also Variable-Size Codes, Prefix Property, Shannon-Fano Coding, Huffman Coding, and Arithmetic Coding.)

Statistical Model

(See Model of Compression.)

String Compression

In general, compression methods based on strings of symbols can be more efficient than methods that compress individual symbols (Section 3.1).

Subsampling

Subsampling is, possibly, the simplest way to compress an image. One approach to subsampling is simply to ignore some of the pixels. The encoder may, for example, ignore every other row and every other column of the image, and write the remaining pixels (which constitute 25% of the image) on the compressed stream. The decoder inputs the compressed data and uses each pixel to generate four identical pixels of the reconstructed image. This, of course, involves the loss of much image detail and is rarely acceptable. (See also Lossy Compression.)

Symbol

The smallest unit of the data to be compressed. A symbol is normally a byte but may also be a bit, a trit $\{0, 1, 2\}$, or anything else. (See also Alphabet.)

Symbol Ranking

A context-based method (Section 8.2) where the context C of the current symbol S (the N symbols preceding S) is used to prepare a list of symbols that are likely to follow C. The list is arranged from most likely to least likely. The position of S in this list (position numbering starts from 0) is then written by the encoder, after being suitably encoded, on the output stream.

Taps

Wavelet filter coefficients. (See also Continuous Wavelet Transform, Discrete Wavelet Transform.)

TAR

The standard UNIX archiver. The name TAR stands for Tape ARchive. It groups a number of files into one file without compression. After being compressed by the UNIX `compress` program, a TAR file gets an extension name of `tar.z`.

Textual Image Compression

A compression method for hard copy documents containing printed or typed (but not handwritten) text. The text can be in many fonts and may consist of musical notes, hieroglyphs, or any symbols. Pattern recognition techniques are used to recognize text characters that are identical or at least similar. One copy of each group of identical characters is kept in a library. Any leftover material is considered residue. The method uses different compression techniques for the symbols and the residue. It includes a lossy option where the residue is ignored.

Token

A unit of data written on the compressed stream by some compression algorithms. A token consists of several fields that may have either fixed or variable sizes.

Transform

An image can be compressed by transforming its pixels (which are correlated) to a representation where they are *decorrelated*. Compression is achieved if the new values are smaller, on average, than the original ones. Lossy compression can be achieved by quantizing the transformed values. The decoder inputs the transformed values from the compressed stream and reconstructs the (precise or approximate) original data by applying the opposite transform. (See also Discrete Cosine Transform, Fourier Transform, Continuous Wavelet Transform, Discrete Wavelet Transform.)

Triangle mesh

Polygonal surfaces are very popular in computer graphics. Such a surface consists of flat polygons, mostly triangles, so there is a need for special methods to compress a triangle mesh. Such a method is edgebreaker (Section 8.11).

Trit

A ternary (base 3) digit. It can be 0, 1, or 2.

Unary Code

A way to generate variable-size codes in one step. The unary code of the nonnegative integer n is defined (Section 2.3.1) as $n-1$ ones followed by one zero (Table 2.3). There is also a general unary code. (See also Golomb Code.)

Unicode

A new international standard code, the Unicode, has been proposed, and is being developed by the international Unicode organization (`www.unicode.org`). Unicode uses 16-bit codes for its characters, so it provides for $2^{16} = 64\text{K} = 65{,}536$ codes. (Notice that doubling the size of a code much more than doubles the number of possible codes. In fact, it *squares* the number of codes.) Unicode includes all the ASCII codes in addition to codes for characters in foreign languages (including complete sets of Korean, Japanese, and Chinese characters) and many mathematical and other symbols. Currently, about 39,000 out of the 65,536 possible codes have been assigned, so there is room for adding more symbols in the future.

The Microsoft Windows NT operating system has adopted Unicode, as have also AT&T Plan 9 and Lucent Inferno. (See also ASCII, Codes.)

V.42bis Protocol

This is a standard, published by the ITU-T (page 91) for use in fast modems. It is based on the older V.32bis protocol and is supposed to be used for fast transmission rates, up to 57.6K baud. The standard contains specifications for data compression and error correction, but only the former is discussed, in Section 3.18.

V.42bis specifies two modes: a *transparent* mode, where no compression is used, and a *compressed* mode using an LZW variant. The former is used for data streams that don't compress well, and may even cause expansion. A good example is an already compressed file. Such a file looks like random data, it does not have any repetitive patterns, and trying to compress it with LZW will fill up the dictionary with short, two-symbol, phrases.

Variable-Size Codes

These are used by statistical methods. Such codes should satisfy the prefix property (Section 2.2) and should be assigned to symbols based on their probabilities. (See also Prefix Property, Statistical Methods.)

Vector Quantization

This is a generalization of the scalar quantization method. It is used for both image and sound compression. In practice, vector quantization is commonly used to compress data that has been digitized from an analog source, such as sampled sound and scanned images (drawings or photographs). Such data is called *digitally sampled analog data* (DSAD). (See also Scalar Quantization.)

Video Compression

Video compression is based on two principles. The first is the spatial redundancy that exists in each video frame. The second is the fact that very often, a video frame is very similar to its immediate neighbors. This is called *temporal redundancy*. A typical technique for video compression should thus start by encoding the first frame using an image compression method. It should then encode each successive frame by identifying the differences between the frame and its predecessor, and encoding these differences.

Voronoi Diagrams

Imagine a petri dish ready for growing bacteria. Four bacteria of different types are simultaneously placed in it at different points and immediately start multiplying. We assume that their colonies grow at the same rate. Initially, each colony consists of a growing circle around one of the starting points. After a while some of them meet and stop growing in the meeting area due to lack of food. The final result is that the entire dish gets divided into four areas, one around each of the four starting points, such that all the points within area i are closer to starting point i than to any other start point. Such areas are called *Voronoi regions* or *Dirichlet Tessellations*.

WFA

This method uses the fact that most images of interest have a certain amount of self similarity (i.e., parts of the image are similar, up to size or brightness, to the entire image or to other parts). It partitions the image into subsquares using a quadtree, and uses a recursive inference algorithm to express relations between parts of the image in a graph. The graph is similar to graphs used to describe finite-state automata. The method is lossy, since parts of a real image may be very similar to other parts. WFA is a very efficient method for compression of grayscale and color images. (See also GFA, Quadtrees, Resolution-Independent Compression.)

WSQ

An efficient lossy compression method specifically developed for compressing fingerprint images. The method involves a wavelet transform of the image, followed by scalar quantization of the wavelet coefficients, and by RLE and Huffman coding of the results. (See also Discrete Wavelet Transform.)

Zero-Probability Problem

When samples of data are read and analyzed in order to generate a statistical model of the data, certain contexts may not appear, leaving entries with zero counts and thus zero probability in the frequency table. Any compression method requires that such entries be somehow assigned nonzero probabilities.

Zip

Popular software that implements the so-called "deflation" algorithm (Section 3.19) that uses a variant of LZ77 combined with static Huffman coding. It uses a 32 Kb-long sliding dictionary, and a look-ahead buffer of 258 bytes. When a string is not found in the dictionary, it is emitted as a sequence of literal bytes. (See also Gzip.)

Necessity is the mother of compression.
Aesop (paraphrased)

Answers to Exercises

> A bird does not sing because he has an answer,
> he sings because he has a song.
>
> —Chinese Proverb

1: abstemious, abstentious, adventitious, annelidous, arsenious, arterious, facetious, sacrilegious.

2: When a software house has a popular product they tend to come up with new versions. A user can update an old version to a new one, and the update usually comes as a compressed file on a floppy disk. Over time the updates get bigger and, at a certain point, an update may not fit on a single floppy. This is why good compression is important in the case of software updates. The time it takes to compress and decompress the update is unimportant since these operations are typically done just once. Recently, software makers have taken to providing updates over the Internet, but even in such cases it is important to have small files because of the download times involved.

1.1: (1) ask a question, (2) absolutely necessary, (3) advance warning, (4) boiling hot, (5) climb up, (6) close scrutiny, (7) exactly the same, (8) free gift, (9) hot water heater, (10) my personal opinion, (11) newborn baby, (12) postponed until later, (13) unexpected surprise, (14) unsolved mysteries.

1.2: An obvious way is to use them to code the five most common strings in the text. Since irreversible text compression is a special-purpose method, the user may know what strings are common in any particular text to be compressed. The user may specify five such strings to the encoder, and they should also be written at the start of the output stream, for the decoder's use.

1.3: 6,8,0,1,3,1,4,1,3,1,4,1,3,1,4,1,3,1,2,2,2,2,6,1,1. The first two numbers are the bitmap resolution (6×8). If each number occupies a byte on the output stream, then its size is 25 bytes, compared to a bitmap size of only 6×8 bits $= 6$ bytes. The method does not work for small images.

1.4: RLE of images is based on the idea that adjacent pixels tend to be identical. The last pixel of a row, however, has no reason to be identical to the first pixel of the next row.

1.5: Each of the first four rows yields the eight runs 1,1,1,2,1,1,1,eol. Rows 6 and 8 yield the four runs 0,7,1,eol each. Rows 5 and 7 yield the two runs 8,eol each. The total number of runs (including the eol's) is thus 44.

When compressing by columns, columns 1, 3, and 6 yield the five runs 5,1,1,1,eol each. Columns 2, 4, 5, and 7 yield the six runs 0,5,1,1,1,eol each. Column 8 gives 4,4,eol, so the total number of runs is 42. This image is thus "balanced" with respect to rows and columns.

1.6: This results in five groups as follows:

$$\begin{aligned}
W_1 \text{ to } W_2 &: 00000, 11111, \\
W_3 \text{ to } W_{10} &: 00001, 00011, 00111, 01111, 11110, 11100, 11000, 10000, \\
W_{11} \text{ to } W_{22} &: 00010, 00100, 01000, 00110, 01100, 01110, \\
&\quad 11101, 11011, 10111, 11001, 10011, 10001, \\
W_{23} \text{ to } W_{30} &: 01011, 10110, 01101, 11010, 10100, 01001, 10010, 00101, \\
W_{31} \text{ to } W_{32} &: 01010, 10101.
\end{aligned}$$

1.7: The seven codes are

$$0000, 1111, 0001, 1110, 0000, 0011, 1111.$$

Forming a string with six runs. Applying the rule of complementing yields the sequence

$$0000, 1111, 1110, 1110, 0000, 0011, 0000,$$

with *seven* runs. The rule of complementing does not always reduce the number of runs.

1.8: As "11 22 90 00 00 33 44". The 00 following the 90 indicates no run, and the following 00 is interpreted as a regular character.

1.9: The six characters "`123ABC`" have ASCII codes 31, 32, 33, 41, 42 and 43. Translating these hexadecimal numbers to binary produces "`00110001 00110010 00110011 01000001 01000010 01000011`".
The next step is to divide this string of 48 bits into 6-bit blocks. They are 001100=12, 010011=19, 001000=8, 110011=51, 010000=16, 010100=20, 001001=9,

000011=3. The character at position 12 in the BinHex table is "-" (position numbering starts at zero). The one at position 19 is "6". The final result is the string "-6)c38*$".

1.10: Exercise 2.1 shows that the binary code of the integer i is $1+\lfloor\log_2 i\rfloor$ bits long. We add $\lfloor\log_2 i\rfloor$ zeros, bringing the total size to $1+2\lfloor\log_2 i\rfloor$ bits.

1.11: Table Ans.1 summarizes the results. In (a), the first string is encoded with $k=1$. In (b) it is encoded with $k=2$. Columns (c) and (d) are the encodings of the second string with $k=1$ and $k=2$, respectively. The averages of the four columns are 3.4375, 3.25, 3.56 and 3.6875; very similar! The move-ahead-k method used with small values of k does not favor strings satisfying the concentration property.

a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0
b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1
c	bacdmnop	2	c	bacdmnop	2	c	bacdmnop	2	c	bacdmnop	2
d	bcadmnop	3	d	cbadmnop	3	d	bcadmnop	3	d	cbadmnop	3
d	bcdamnop	2	d	cdbamnop	1	m	bcdamnop	4	m	cdbamnop	4
c	bdcamnop	2	c	dcbamnop	1	n	bcdmanop	5	n	cdmbanop	5
b	bcdamnop	0	b	cdbamnop	2	o	bcdmnaop	6	o	cdmnbaop	6
a	bcdamnop	3	a	bcdamnop	3	p	bcdmnoap	7	p	cdmnobap	7
m	bcadmnop	4	m	bacdmnop	4	a	bcdmnopa	7	a	cdmnopba	7
n	bcamdnop	5	n	bamcdnop	5	b	bcdmnoap	0	b	cdmnoapb	7
o	bcamndop	6	o	bamncdop	6	c	bcdmnoap	1	c	cdmnobap	0
p	bcamnodp	7	p	bamnocdp	7	d	cbdmnoap	2	d	cdmnobap	1
p	bcamnopd	6	p	bamnopcd	5	m	cdbmnoap	3	m	dcmnobap	2
o	bcamnpod	6	o	bampnocd	5	n	cdmbnoap	4	n	mdcnobap	3
n	bcamnopd	4	n	bamopncd	5	o	cdmnboap	5	o	mndcobap	4
m	bcanmopd	4	m	bamnopcd	2	p	cdmnobap	7	p	mnodcbap	7
	bcamnopd			mbanopcd			cdmnobpa			mnodcpba	
	(a)			(b)			(c)			(d)	

Table Ans.1: Encoding with Move-Ahead-k.

1.12: Table Ans.2 summarizes the decoding steps. Notice how similar it is to Table 1.16, indicating that move-to-front is a symmetric data compression method.

2.1: It is $1+\lfloor\log_2 i\rfloor$ as can be seen by simple experimenting.

2.2: Two is the smallest integer that can serve as the basis for a number system.

2.3: Replacing 10 by 3 we get $x=k\log_2 3\approx 1.58k$. A trit is thus worth about 1.58 bits.

Code input	A (before adding)	A (after adding)	Word
0the	()	(the)	the
1boy	(the)	(the, boy)	boy
2on	(boy, the)	(boy, the, on)	on
3my	(on, boy, the)	(on, boy, the, my)	my
4right	(my, on, boy, the)	(my, on, boy, the, right)	right
5is	(right, my, on, boy, the)	(right, my, on, boy, the, is)	is
5	(is, right, my, on, boy, the)	(is, right, my, on, boy, the)	the
2	(the, is, right, my, on, boy)	(the, is, right, my, on, boy)	right
5	(right, the, is, my, on, boy)	(right, the, is, my, on, boy)	boy
	(boy, right, the, is, my, on)		

Table Ans.2: Decoding Multiple-Letter Words.

2.4: We assume an alphabet with two symbols a_1 and a_2, with probabilities P_1 and P_2, respectively. Since $P_1 + P_2 = 1$, the entropy of the alphabet is $-P_1 \log_2 P_1 - (1 - P_1) \log_2(1 - P_1)$. Table Ans.3 shows the entropies for certain values of the probabilities. When $P_1 = P_2$, at least 1 bit is required to encode each symbol, reflecting the fact that the entropy is at its maximum, the redundancy is zero, and the data cannot be compressed. However, when the probabilities are very different, the minimum number of bits required per symbol drops significantly. We may not be able to develop a compression method using 0.08 bits per symbol but we know that when $P_1 = 99\%$, this is the theoretical minimum.

P_1	P_2	Entropy
99	1	0.08
90	10	0.47
80	20	0.72
70	30	0.88
60	40	0.97
50	50	1.00

Table Ans.3: Probabilities and Entropies of Two Symbols.

> An essential tool of this theory [information] is a quantity for measuring the amount of information conveyed by a message. Suppose a message is encoded into some long number. To quantify the information content of this message, Shannon proposed to count the number of its digits. According to this criterion, 3.14159, for example, conveys twice as much information as 3.14, and six times as much as 3. Struck by the similarity between this recipe and the famous equation on Boltzman's tomb (entropy is the number of digits of probability), Shannon called his formula the "information entropy."
>
> Hans Christian von Baeyer, *Maxwell's Demon*, 1998

2.5: It is easy to see that the unary code satisfies the prefix property, so it definitely can be used as a variable-size code. Since its length L satisfies $L = n$ we get $2^{-L} = 2^{-n}$, so it makes sense to use it in cases were the input data consists of integers n with probabilities $P(n) \approx 2^{-n}$. If the data lends itself to the use of the unary code, the entire Huffman algorithm can be skipped, and the codes of all the symbols can easily and quickly be constructed before compression or decompression starts.

2.6: The triplet $(n, 1, n)$ defines the standard n-bit binary codes, as can be verified by direct construction. The number of such codes is easily seen to be

$$\frac{2^{n+1} - 2^n}{2^1 - 1} = 2^n.$$

The triplet $(0, 0, \infty)$ defines the codes 0, 10, 110, 1110,... which are the unary codes but assigned to the integers 0, 1, 2,... instead of 1, 2, 3,... .

2.7: The number is $(2^{30} - 2^1)/(2^1 - 1) \approx$ A billion.

2.8: This is straightforward. Table Ans.4 shows the code. There are only three different codewords since "start" and "stop" are so close, but there are many codes since "start" is large.

n	$a = 10 + n \cdot 2$	nth codeword	Number of codewords	Range of integers
0	10	$0\underbrace{x...x}_{10}$	$2^{10} = 1K$	0–1023
1	12	$10\underbrace{xx...x}_{12}$	$2^{12} = 4K$	1024–5119
2	14	$11\underbrace{xx...xx}_{14}$	$2^{14} = 16K$	5120–21503
		Total	21504	

Table Ans.4: The General Unary Code (10,2,14).

2.9: Each part of C_4 is the standard binary code of some integer, so it starts with a 1. A part that starts with a 0 thus signals to the decoder that this is the last bit of the code.

2.10: We use the property that the Fibonacci representation of an integer does not have any adjacent 1's. If R is a positive integer, we construct its Fibonacci representation and append a 1-bit to the result. The Fibonacci representation of the integer 5 is 001, so the Fibonacci-prefix code of 5 is 0011. Similarly, the Fibonacci representation of 33 is 1010101, so its Fibonacci-prefix code is 10101011. It is obvious that each of these codes ends with two adjacent 1's, so they can be decoded uniquely. However, the property of not having adjacent 1's restricts the number of binary patterns available for such codes, so they are longer than the other codes shown here.

2.11: Subsequent splits can be done in different ways, but Table Ans.5 shows one way of assigning Shannon-Fano codes to the 7 symbols.

	Prob.	Steps				Final
1.	0.25	1	1			:11
2.	0.20	1	0			:101
3.	0.15	1	0			:100
4.	0.15	0	1			:01
5.	0.10	0	0	1		:001
6.	0.10	0	0	0	0	:0001
7.	0.05	0	0	0	0	:0000

Table Ans.5: Shannon-Fano Example.

The average size in this case is $0.25\times 2+0.20\times 3+0.15\times 3+0.15\times 2+0.10\times 3+0.10\times 4+0.05\times 4=2.75$ bits/symbols.

2.12: This is immediate $-2(0.25\times\log_2 0.25)-4(0.125\times\log_2 0.125)=2.5$.

2.13: If method C does not expand any of the n-bit strings, then the result of applying it to all the n-bit input strings is a set of 2^n binary strings, none of which is longer than n bits and some of which are shorter. This is impossible because the total number of n-bit strings is 2^n, so the total number of shorter strings must be less than that.

2.14: The method does not exploit the redundancy in the input. Even in our simple example the two consecutive `SS` of `SWISS` (and also the two `SS` of `MISS`) result in $5+1=6$ bits. If the dictionary size is 256, then a pair of identical input bytes results in $256+1=257$ bits; a considerable expansion. On the other hand, two consecutive 1-bits are generated only if two consecutive input characters are identical to two consecutive dictionary characters. It seems reasonable to assume that the number of consecutive zeros between two 1-bits is, on average, half the dictionary size. If half the dictionary size is greater than eight (the size of a character), expansion occurs.

Later that evening, we were all sitting around the table talking when someone said something and Murray Gell-Mann remarked, "Oh, that's a pleonasm." Everyone went, "What?" "It's a sentence with a triple redundancy," Gell-Mann stated. Gell-Mann is well known among his associates for his pedantic knowledge of language and facts. Feynman and I sneaked into my library where we looked it up in the dictionary. Gell-Mann was right. Feynman hit his fist on the table, and exclaimed, "DAMN IT! He's always GODDAMNED right, always!" "Let's see if we can catch him tonight," I replied.

— Al Seckel

2.15: 1: The initial 10 indicates that $v_1 = 1$. The following 0 indicates that $v_2 = 0$. The fourth bit is 1, indicating that there are more nonzero vector elements to be decoded. Bits 5–6 are 10, indicating that $v_3 = 1$. Bits 7–8 are 11, indicating that $v_4 = -1$. Bit 9 is zero, indicating that the remaining elements are all zeros.

2: The shortest code, for all-zero vectors, is 000, just three bits. As an example of the longest code let's assume an 8-component vector where all the elements are 1. The resulting code is 101011010110101010, an 18-bit number, longer than the fixed-size code of 13 bits.

This method results in very short codes for certain vectors and very long codes for others. Thus, this method should be used in cases where we have to compress vectors that tend to have trailing zeros.

2.16: Figure Ans.6a,b,c shows the three trees. The codes sizes for the trees are

$$(5+5+5+5\cdot 2+3\cdot 3+3\cdot 5+3\cdot 5+12)/30 = 76/30,$$
$$(5+5+4+4\cdot 2+4\cdot 3+3\cdot 5+3\cdot 5+12)/30 = 76/30,$$
$$(6+6+5+4\cdot 2+3\cdot 3+3\cdot 5+3\cdot 5+12)/30 = 76/30.$$

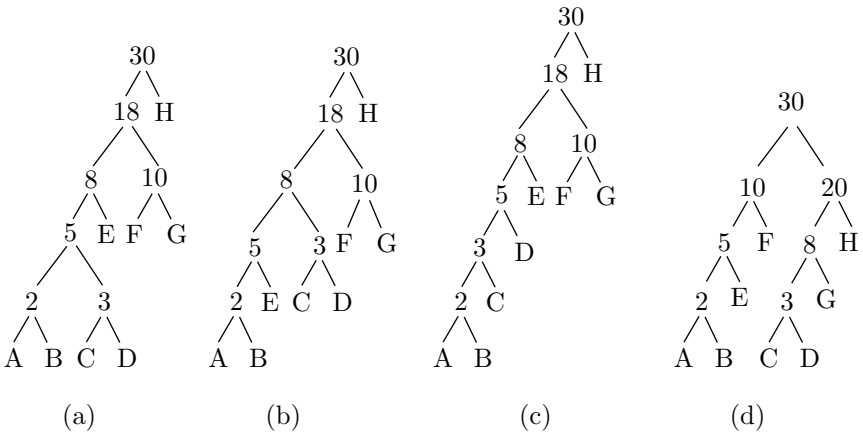


Figure Ans.6: Three Huffman Trees For Eight Symbols.

2.17: After adding symbols A, B, C, D, E, F, and G to the tree we were left with the three symbols ABEF (with probability 10/30), CDG (with probability 8/30), and H (with probability 12/30). The two symbols with lowest probabilities were ABEF and CDG, so they had to be merged. Instead, symbols CDG and H were merged, creating a non-Huffman tree.

2.18: The second row of Table Ans.7 (due to Guy Blelloch) shows a symbol whose Huffman code is three bits long, but for which $\lceil -\log_2 0.3 \rceil = \lceil 1.737 \rceil = 2$.

P_i	Code	$-\log_2 P_i$	$\lceil -\log_2 P_i \rceil$
.01	000	6.644	7
*.30	001	1.737	2
.34	01	1.556	2
.35	1	1.515	2

Table Ans.7: A Huffman Code Example.

2.19: Imagine a large alphabet where all the symbols have (about) the same probability. Since the alphabet is large, that probability will be small, resulting in long codes. Imagine the other extreme case, where certain symbols have high probabilities (and, therefore, short codes). Since the probabilities have to add up to 1, the rest of the symbols will have low probabilities (and, therefore, long codes). We thus see that the size of a code depends on the probability, but is indirectly affected by the size of the alphabet.

2.20: Answer not provided.

2.21: Figure Ans.8 shows Huffman codes for 5, 6, 7, and 8 symbols with equal probabilities. In the case where n is a power of 2, the codes are simply the fixed-sized ones. In other cases the codes are very close to fixed-size. This shows that symbols with equal probabilities do not benefit from variable-size codes. (This is another way of saying that random text cannot be compressed.) Table Ans.9 shows the codes, their average sizes and variances.

2.22: The number of groups increases exponentially from 2^s to $2^{s+n} = 2^s \times 2^n$.

2.23: The binary value of 127 is 01111111 and that of 128 is 10000000. Half the pixels in each bitplane will therefore be 0 and the other half, 1. In the worst case, each bitplane will be a checkerboard, i.e., will have many runs of size one. In such a case, each run requires a 1-bit code, leading to one codebit per pixel per bitplane, or eight codebits per pixel for the entire image, resulting in no compression at all. In comparison, a Huffman code for such an image requires just two codes (since there are just two pixel values) and they can be one bit each. This leads to one codebit per pixel, or a compression factor of eight.

2.24: The two trees are shown in Figure 2.24c,d. The average code size for the binary Huffman tree is

$$1\times.49 + 2\times.25 + 5\times.02 + 5\times.03 + 5\times.04 + 5\times.04 + 3\times.12 = 2\,\text{bits/symbol},$$

and that of the ternary tree is

$$1\times.26 + 3\times.02 + 3\times.03 + 3\times.04 + 2\times.04 + 2\times.12 + 1\times.49 = 1.34\,\text{trits/symbol}.$$

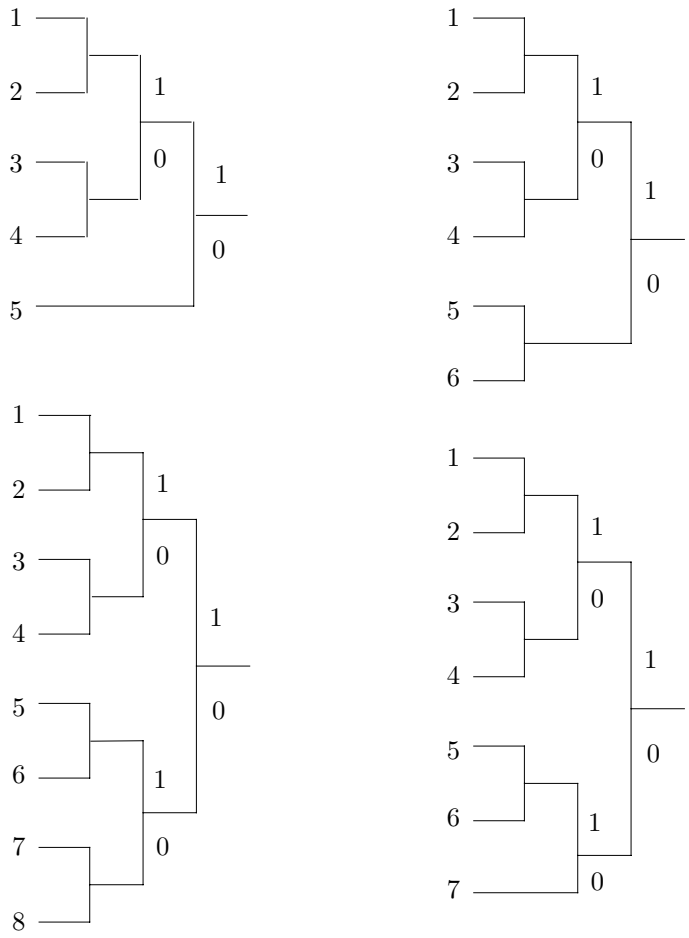


Figure Ans.8: Huffman Codes for Equal Probabilities.

n	p	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	Avg. size	Var.
5	0.200	111	110	101	100	0				2.6	0.64
6	0.167	111	110	101	100	01	00			2.672	0.2227
7	0.143	111	110	101	100	011	010	00		2.86	0.1226
8	0.125	111	110	101	100	011	010	001	000	3	0

Table Ans.9: Huffman Codes for 5–8 Symbols.

2.25: Figure Ans.10 shows how the loop continues until the heap shrinks to just one node that is the single pointer 2. This indicates that the total frequency (which happens to be 100 in our example) is stored in `A[2]`. All other frequencies have been replaced by pointers. Figure Ans.11a shows the heaps generated during the loop.

2.26: The code lengths for the seven symbols are 2, 2, 3, 3, 4, 3, and 4 bits. This can also be verified from the Huffman code-tree of Figure Ans.11b. A set of codes derived from this tree is shown in the following table:

Count:	25	20	13	17	9	11	5
Code:	01	11	101	000	0011	100	0010
Length:	2	2	3	3	4	3	4

2.27: A symbol with high frequency of occurrence should be assigned a shorter code. Therefore it has to appear high in the tree. The requirement that at each level the frequencies be sorted from left to right is artificial. In principle it is not necessary but it simplifies the process of updating the tree.

2.28: Figure Ans.12 shows the initial tree and how it is updated in the 11 steps (a) through (k). Notice how the *esc* symbol gets assigned different codes all the time, and how the different symbols move about in the tree and change their codes. Code 10, e.g., is the code of symbol "i" in steps (f) and (i), but is the code of "s" in steps (e) and (j). The code of a blank space is 011 in step (h), but 00 in step (k).

The final output is: "`s0i00r100␣1010000d011101000`". A total of $5\times8+22=62$ bits. The compression ratio is thus $62/88\approx0.7$.

2.29: A simple calculation shows that the average size of a token in Table 2.32 is about 9 bits. In stage 2, each 8-bit byte will be replaced, on the average, by a 9-bit token, resulting in an expansion factor of $9/8=1.125$ or 12.5%.

2.30: The decompressor will interpret the input data as "111110 0110 11000 0...", which is the string "XRP...".

2.31: A typical fax machine scans lines that are about 8.2 inches wide ($\approx$ 208 mm). A blank scan line thus produces 1,664 consecutive white pels.

2.32: These codes are needed for cases such as example 4, where the run length is 64, 128 or any length for which a make-up code has been assigned.

2.33: There may be fax machines (now or in the future) built for wider paper, so the Group 3 code was designed to accommodate them.

2.34: Each scan line starts with a white pel, so when the decoder inputs the next code it knows whether it is for a run of white or black pels. This is why the codes of Table 2.38 have to satisfy the prefix property in each column but not between the columns.

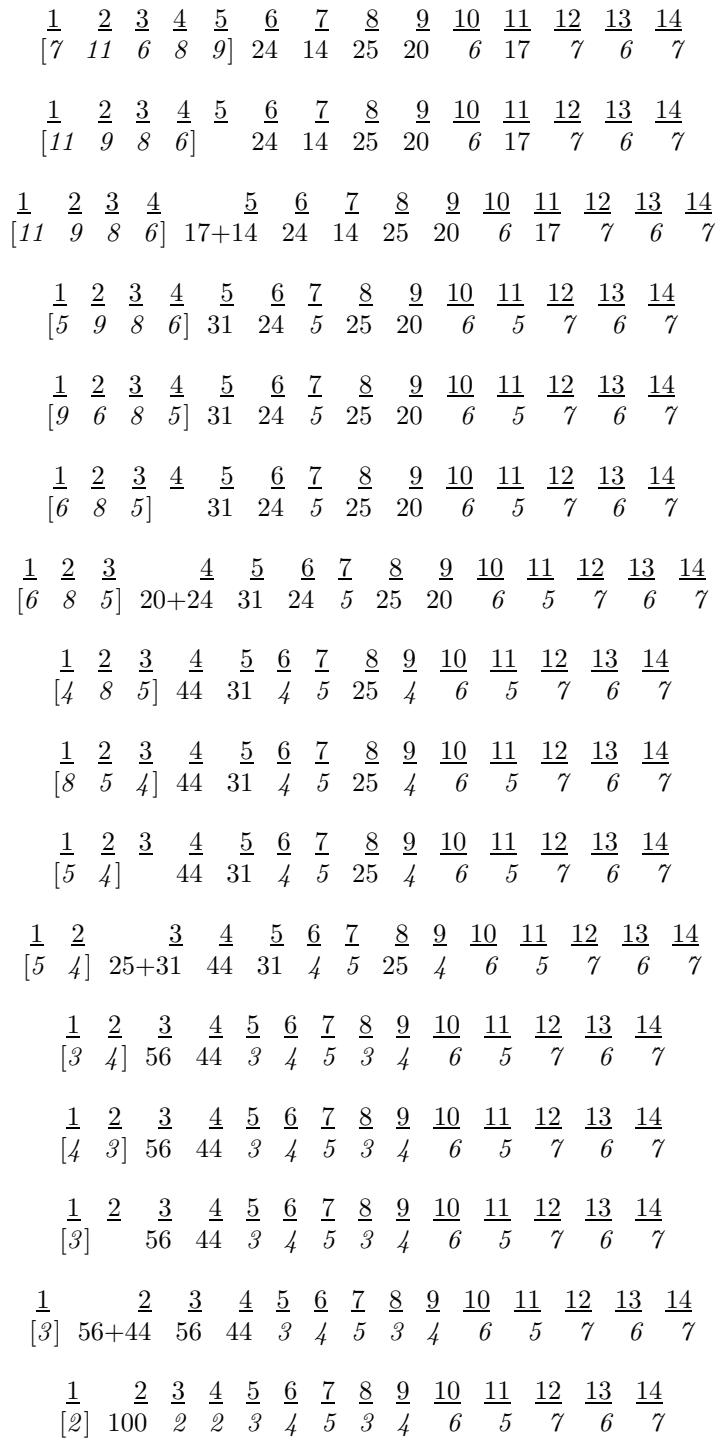
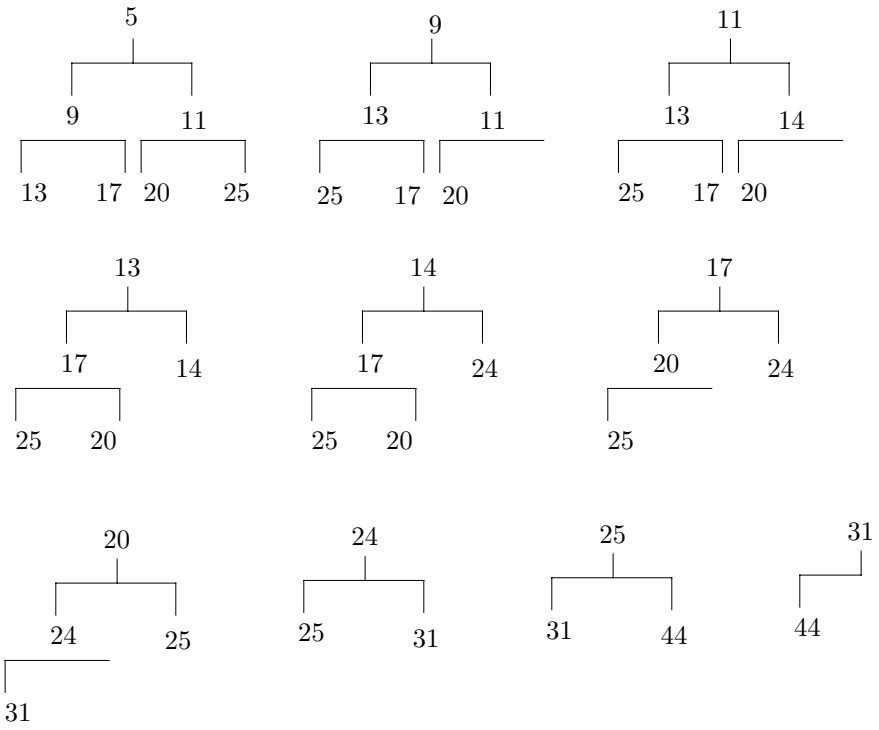
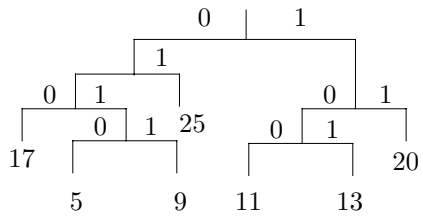


Figure Ans.10: Sifting the Heap.



(a)

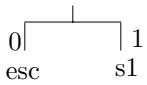


(b)

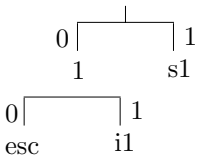
Figure Ans.11: (a) Heaps. (b) Huffman Code-Tree.

Initial tree

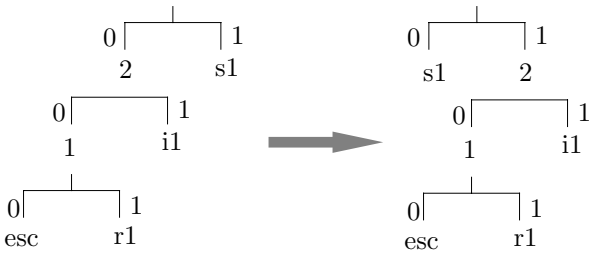
(a). Input: s. Output: ‘s’.
$esc\, s_1$



(b). Input: i. Output: 0‘i’.
$esc\, i_1\, 1\ s_1$



(c). Input: r. Output: 00‘r’.
$esc\, r_1\, 1\, i_1\, 2\, s_1 \rightarrow$
$esc\, r_1\, 1\, i_1\, s_1\, 2$



(d). Input: ␣. Output: 100‘␣’.
$esc_{\sqcup 1}\, 1\, r_1\, 2\, i_1\, s_1\, 3 \rightarrow$
$esc_{\sqcup 1}\, 1\, r_1\, s_1\, i_1\, 2\, 2$

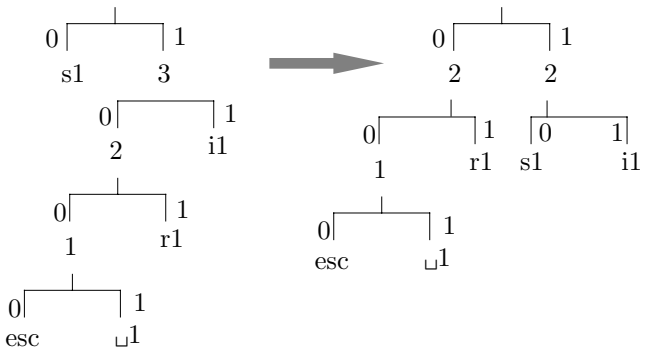
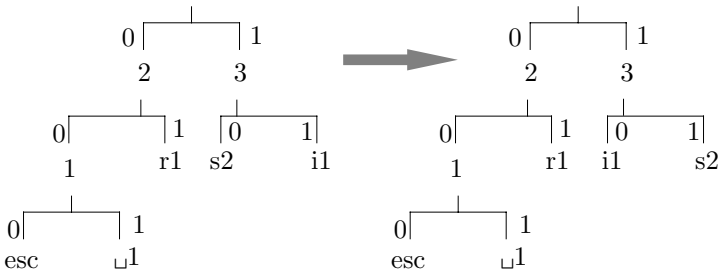
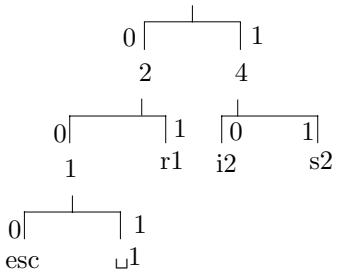


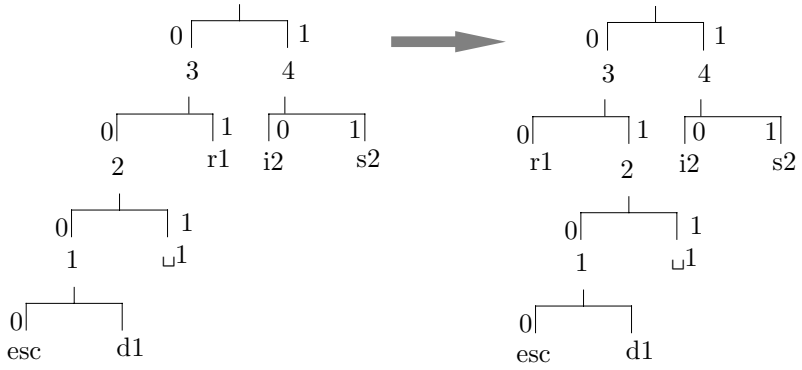
Figure Ans.12: Exercise 2.28. Part I.



(e). Input: s. Output: 10.
$esc_{\sqcup 1}\, 1\, r_1\, s_2\, i_1\, 2\, 3 \rightarrow$
$esc_{\sqcup 1}\, 1\, r_1\, i_1\, s_2\, 2\, 3$

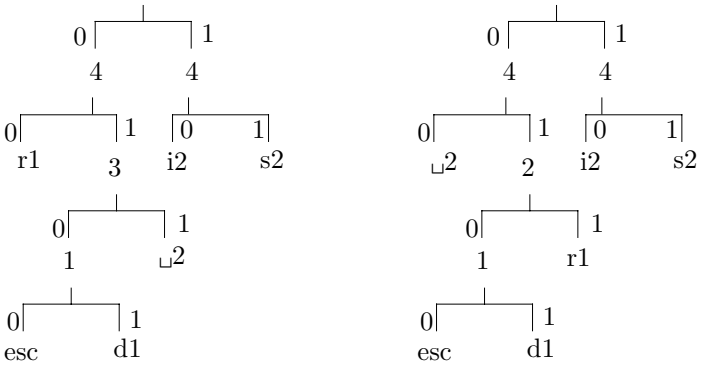


(f). Input: i. Output: 10.
$esc_{\sqcup 1}\, 1\, r_1\, i_2\, s_2\, 2\, 4$

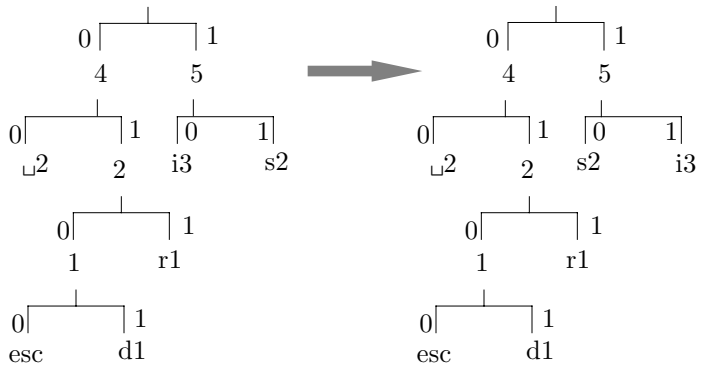


(g). Input: d. Output: 000'd'.
$esc\, d_1\, 1\, \sqcup_1\, 2\, r_1\, i_2\, s_2\, 3\, 4 \rightarrow$
$esc\, d_1\, 1\, \sqcup_1\, r_1\, 2\, i_2\, s_2\, 3\, 4$

Figure Ans.12: Exercise 2.28. Part II.

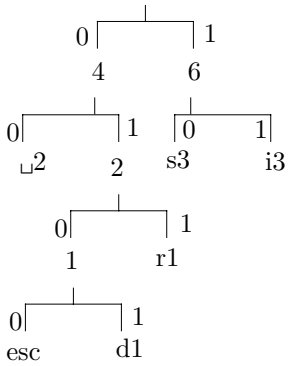


(h). Input: ␣. Output: 011.
$esc\, d_1\, 1\, \sqcup_2\, r_1\, 3\, i_2\, s_2\, 4\, 4 \rightarrow$
$esc\, d_1\, 1\, r_1\, \sqcup_2\, 2\, i_2\, s_2\, 4\, 4$

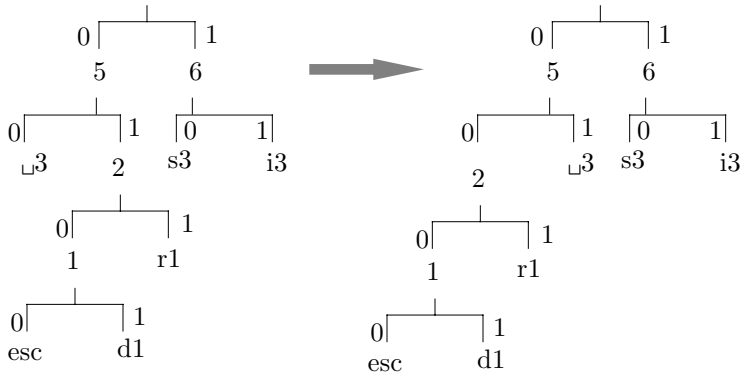


(i). Input: i. Output: 10.
$esc\, d_1\, 1\, r_1\, \sqcup_2\, 2\, i_3\, s_2\, 4\, 5 \rightarrow$
$esc\, d_1\, 1\, r_1\, \sqcup_2\, 2\, s_2\, i_3\, 4\, 5$

Figure Ans.12: Exercise 2.28. Part III.



(j). Input: s. Output: 10.
$esc\, d_1\, 1\, r_1\, \sqcup_2\, 2\, s_3\, i_3\, 4\, 6$



(k). Input: $\sqcup$. Output: 00.
$esc\, d_1\, 1\, r_1\, \sqcup_3\, 2\, s_3\, i_3\, 5\, 6 \rightarrow$
$esc\, d_1\, 1\, r_1\, 2\, \sqcup_3\, s_3\, i_3\, 5\, 6$

Figure Ans.12: Exercise 2.28. Part IV.

2.35: The code of a run length of one white pel is 000111, and that of one black pel is 010. Two consecutive pels of different colors are thus coded into 9 bits. Since the uncoded data requires just two bits (01 or 10), the compression ratio is 9/2=4.5 (the compressed stream is 4.5 times longer than the uncompressed one; a large expansion).

2.36: Figure Ans.13 shows the modes and the actual code generated from the two lines.

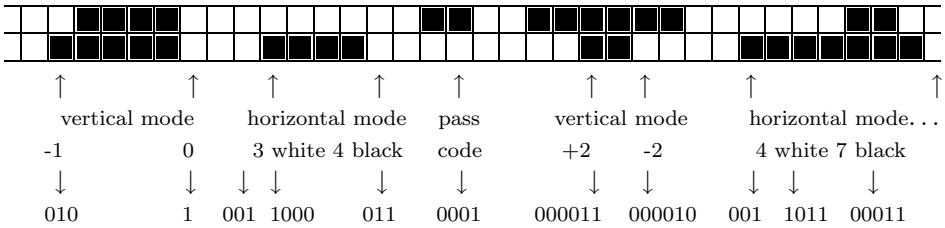


Figure Ans.13: Two-Dimensional Coding Example.

2.37: Table Ans.14 shows the steps of encoding the string $a_2a_2a_2a_2$. Because of the high probability of a_2 the low and high variables start at very different values and approach each other slowly.

a_2	$0.0 + (1.0 - 0.0) \times 0.023162 = 0.023162$
	$0.0 + (1.0 - 0.0) \times 0.998162 = 0.998162$
a_2	$0.023162 + .975 \times 0.023162 = 0.04574495$
	$0.023162 + .975 \times 0.998162 = 0.99636995$
a_2	$0.04574495 + 0.950625 \times 0.023162 = 0.06776322625$
	$0.04574495 + 0.950625 \times 0.998162 = 0.99462270125$
a_2	$0.06776322625 + 0.926859375 \times 0.023162 = 0.08923124309375$
	$0.06776322625 + 0.926859375 \times 0.998162 = 0.99291913371875$

Table Ans.14: Encoding the String $a_2a_2a_2a_2$.

2.38: An argument similar to the one in the previous exercise shows that there are two ways of writing this number. It can be written either as 0.1000... or 0.0111... .

2.39: In practice, the eof symbol has to be included in the original table of frequencies and probabilities. This symbol is the last to be encoded and the decoder stops when it detects an eof.

2.40: The encoding steps are simple (see first example on page 101). We start with the interval $[0,1)$. The first symbol a_2 reduces the interval to $[0.4,0.9)$. The second one, to $[0.6,0.85)$, the third one to $[0.7,0.825)$ and the eof symbol, to $[0.8125,0.8250)$. The approximate binary values of the last interval are 0.1101000000 and 0.1101001100, so we select the 7-bit number 1101000 as our code.

The probability of the string "$a_2a_2a_2$eof" is $(0.5)^3 \times 0.1 = 0.0125$, but since $-\log_2 0.125 \approx 6.322$ it follows that the practical minimum code size is 7 bits.

2.41: Perhaps the simplest way is to calculate a set of Huffman codes for the symbols, using their probabilities. This converts each symbol to a binary string, so the input stream can be encoded by the QM-coder. After the compressed stream is decoded by the QM-decoder, an extra step is needed, to convert the resulting binary strings back to the original symbols.

2.42: The results are shown in Tables Ans.15 and Ans.16. When all symbols are LPS, the output C always points at the bottom $A(1-Qe)$ of the upper (LPS) subinterval. When the symbols are MPS, the output always points at the bottom of the lower (MPS) subinterval, i.e., 0.

2.43: If the current input bit is an LPS, A is shrunk to Qe, which is always 0.5 or less, so A always has to be renormalized in such a case.

2.44: The results are shown in Tables Ans.17 and Ans.18 (compare with the answer to exercise 2.42).

2.45: The four decoding steps are as follows:

Step 1: $C = 0.981$, $A = 1$, the dividing line is $A(1-Qe) = 1(1-0.1) = 0.9$, so the LPS and MPS subintervals are $[0,0.9)$ and $[0.9,1)$. Since C points to the upper subinterval, an LPS is decoded. The new C is $0.981 - 1(1-0.1) = 0.081$ and the new A is $1\times 0.1 = 0.1$.
Step 2: $C = 0.081$, $A = 0.1$, the dividing line is $A(1-Qe) = 0.1(1-0.1) = 0.09$, so the LPS and MPS subintervals are $[0,0.09)$ and $[0.09,0.1)$, and an MPS is decoded. C is unchanged and the new A is $0.1(1-0.1) = 0.09$.
Step 3: $C = 0.081$, $A = 0.09$, the dividing line is $A(1-Qe) = 0.09(1-0.1) = 0.0081$, so the LPS and MPS subintervals are $[0,0.0081)$ and $[0.0081,0.09)$, and an LPS is decoded. The new C is $0.081-0.09(1-0.1) = 0$ and the new A is $0.09\times 0.1 = 0.009$.
Step 4: $C = 0$, $A = 0.009$, the dividing line is $A(1-Qe) = 0.009(1-0.1) = 0.00081$, so the LPS and MPS subintervals are $[0,0.00081)$ and $[0.00081,0.009)$, and an MPS is decoded. C is unchanged and the new A is $0.009(1-0.1) = 0.00081$.

2.46: In practice, an encoder may encode texts other than English, such as a foreign language or the source code of a computer program. Even in English there are some examples of a `q` not followed by a `u`, such as in this sentence. (The author has noticed that science-fiction writers tend to use non-English sounding words, such as Qaal, to name characters in their works.)

Symbol	C	A
Initially	0	1
s1 (LPS)	$0+1(1-0.5)=0.5$	$1\times 0.5=0.5$
s2 (LPS)	$0.5+0.5(1-0.5)=0.75$	$0.5\times 0.5=0.25$
s3 (LPS)	$0.75+0.25(1-0.5)=0.875$	$0.25\times 0.5=0.125$
s4 (LPS)	$0.875+0.125(1-0.5)=0.9375$	$0.125\times 0.5=0.0625$

Table Ans.15: Encoding Four Symbols With $Qe = 0.5$.

Symbol	C	A
Initially	0	1
s1 (MPS)	0	$1\times(1-0.1)=0.9$
s2 (MPS)	0	$0.9\times(1-0.1)=0.81$
s3 (MPS)	0	$0.81\times(1-0.1)=0.729$
s4 (MPS)	0	$0.729\times(1-0.1)=0.6561$

Table Ans.16: Encoding Four Symbols With $Qe = 0.1$.

Symbol	C	A	Renor. A	Renor. C
Initially	0	1		
s1 (LPS)	$0+1-0.5=0.5$	0.5	1	1
s2 (LPS)	$1+1-0.5=1.5$	0.5	1	3
s3 (LPS)	$3+1-0.5=3.5$	0.5	1	7
s4 (LPS)	$7+1-0.5=6.5$	0.5	1	13

Table Ans.17: Renormalization Added to Table Ans.15.

Symbol	C	A	Renor. A	Renor. C
Initially	0	1		
s1 (MPS)	0	$1-0.1=0.9$		
s2 (MPS)	0	$0.9-0.1=0.8$		
s3 (MPS)	0	$0.8-0.1=0.7$	1.4	0
s4 (MPS)	0	$1.4-0.1=1.3$		

Table Ans.18: Renormalization Added to Table Ans.16.

2.47: $256^2 = 65,536$, a manageable number, but $256^3 = 16,777,216$, perhaps too big for a practical implementation, unless a sophisticated data structure is used, or unless the encoder gets rid of older data from time to time.

2.48: A color or gray-scale image with 4-bit pixels. Each symbol is a pixel, and there are 16 different ones.

2.49: An object file generated by a compiler or an assembler normally has several distinct parts including the machine instructions, symbol table, relocation bits, and constants. Such parts may have different bit distributions.

2.50: The alphabet has to be extended, in such a case, to include one more symbol. If the original alphabet consisted of all the possible 256 8-bit bytes, it should be extended to 9-bit symbols, and should include 257 values.

2.51: Table Ans.19 shows the groups generated in both cases and makes it clear why these particular probabilities were assigned.

Context	f	p
abc→x	10	10/11
Esc	1	1/11

Context		f	p
abc→	a_1	1	1/20
→	a_2	1	1/20
→	a_3	1	1/20
→	a_4	1	1/20
→	a_5	1	1/20
→	a_6	1	1/20
→	a_7	1	1/20
→	a_8	1	1/20
→	a_9	1	1/20
→	a_{10}	1	1/20
Esc		10	10/20
Total		20	

Table Ans.19: Stable vs. Variable Data.

2.52: The d is added to the order-0 contexts with frequency 1. The escape frequency should be incremented from 5 to 6, bringing the total frequencies from 19 up to 21. The probability assigned to the new d is therefore 1/21, and that assigned to the escape is 6/21. All other probabilities are reduced from $x/19$ to $x/21$.

2.53: The new d would require switching from order-2 to order-0, sending two escapes that take 1 and 1.32 bits. The d is now found in order-0 with probability 1/21, so it is encoded in 4.39 bits. The total number of bits required to encode the second d is thus $1 + 1.32 + 4.39 = 6.71$, still greater than 5.

2.54: The first three cases don't change. They still code a symbol with 1, 1.32, and 6.57 bits, which is less than the 8 bits required for a 256-symbol alphabet without compression. Case 4 is different since the `d` is now encoded with a probability of 1/256, producing 8 instead of 4.8 bits. The total number of bits required to encode the `d` in case 4 is now $1 + 1.32 + 1.93 + 8 = 12.25$.

2.55: The final trie is shown in Figure Ans.20.

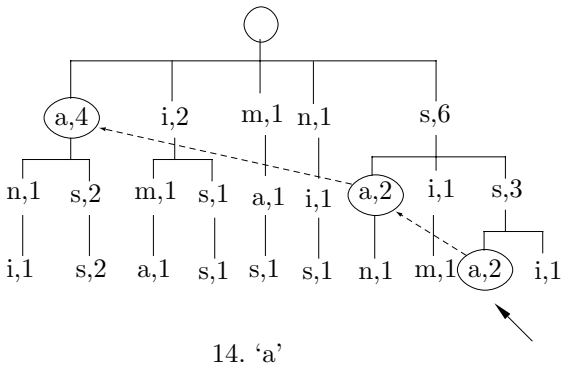


Figure Ans.20: Final Trie of "`assanissimassa`".

2.56: This is, of course

$$1 - P_e(b_{t+1} = 1|b_1^t) = 1 - \frac{b + 1/2}{a + b + 1} = \frac{a + 1/2}{a + b + 1}.$$

2.57: For the first string the single bit has a suffix of 00, so the probability of leaf 00 is $P_e(1, 0) = 1/2$. This is equal to the probability of string 0 without any suffix. For the second string each of the two zero bits has suffix 00, so the probability of leaf 00 is $P_e(2, 0) = 3/8 = 0.375$. This is greater than the probability 0.25 of string 00 without any suffix. Similarly, the probabilities of the remaining three strings are $P_e(3, 0) = 5/8 \approx 0.625$, $P_e(4, 0) = 35/128 \approx 0.273$, and $P_e(5, 0) = 63/256 \approx 0.246$. As the strings get longer, their probabilities get smaller but they are greater than the probabilities without the suffix. Having a suffix of 00 thus increases the probability of having strings of zeros following it.

2.58: This is straightforward and is shown in Figure 2.77b.

2.59: The four trees are shown in Figure Ans.21a–d. The weighted probability that the next bit will be a zero given that three zeros have just been generated is 0.5. The weighted probability to have two consecutive zeros given the suffix 000 is 0.375, higher than the 0.25 without the suffix.

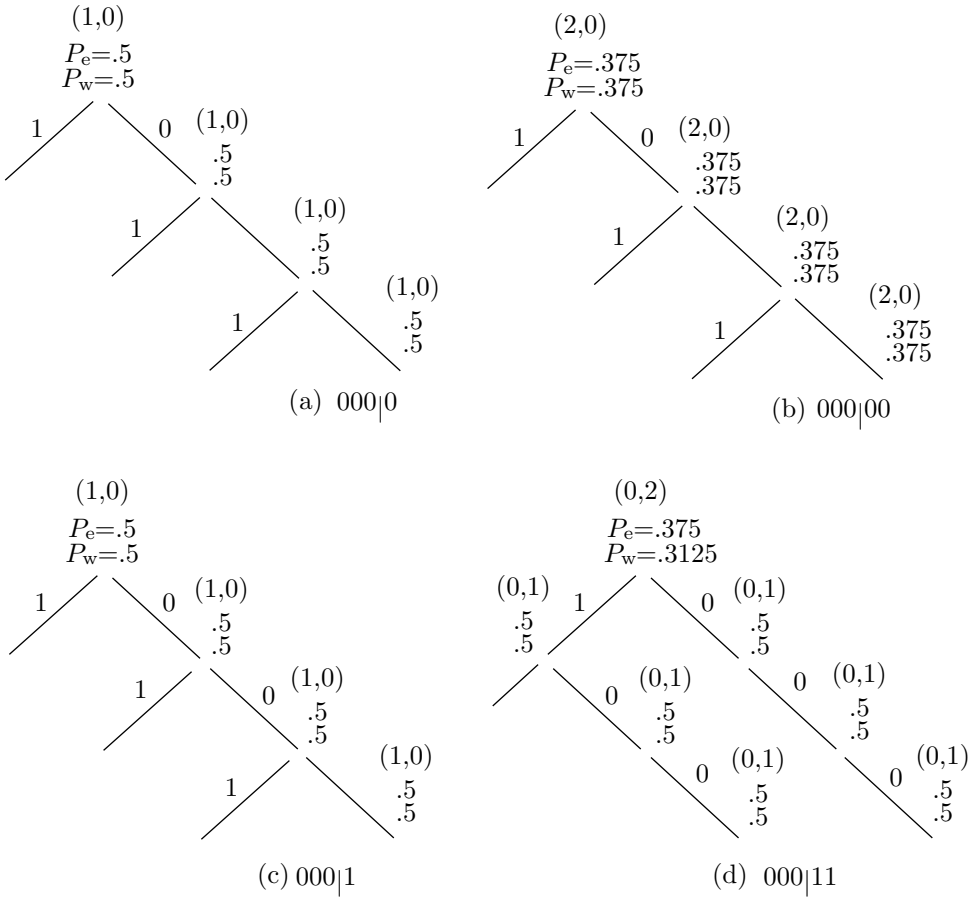


Figure Ans.21: Context Trees For 000|0, 000|00, 000|1, and 000|11.

3.1: The size of the output stream is $N[48 - 28P] = N[48 - 25.2] = 22.8N$. The size of the input stream is, as before, $40N$. The compression factor is thus $40/22.8 \approx 1.75$.

3.2: The decoder doesn't know but it does not need to know. The decoder simply reads tokens and uses each offset to find a string of text without having to know whether the string was a first or a last match.

3.3: The next step matches the space and encodes the string "␣e".

sir␣sid|␣eastman␣easily␣	⇒	(4,1,"e")
sir␣sid␣e|astman␣easily␣te	⇒	(0,0,"a")

and the next one matches nothing and encodes the "a".

3.4: The first two characters CA at positions 17–18 are a repeat of the CA at positions 9–10, so they will be encoded as a string of length 2 at offset $18 - 10 = 8$.

The next two characters AC at positions 19–20 are a repeat of the string at positions 8–9, so they will be encoded as a string of length 2 at offset $20 - 9 = 11$.

3.5: The decoder interprets the first 1 of the end marker as the start of a token. The second 1 is interpreted as the prefix of a 7-bit offset. The next 7 bits are 0 and they identify the end-marker as such, since a "normal" offset cannot be zero.

3.6: This is straightforward. The remaining steps are shown in Table Ans.22

Dictionary		Token	Dictionary		Token
15	"␣t"	(4, "t")	21	"␣si"	(19,"i")
16	"e"	(0, "e")	22	"c"	(0, "c")
17	"as"	(8, "s")	23	"k"	(0, "k")
18	"es"	(16,"s")	24	"␣se"	(19,"e")
19	"␣s"	(4, "s")	25	"al"	(8, "l")
20	"ea"	(4, "a")	26	"s(eof)"	(1, "(eof)")

Table Ans.22: Next 12 Encoding Steps in the LZ78 Example.

3.7: Table Ans.23 shows the last three steps.

p_src	3 chars	Hash index	P	Output	Binary output
11	"h t"	7	any→7	h	01101000
12	"␣th"	5	5→5	4,7	0000|0011|00000111
16	"ws"			ws	01110111|01110011

Table Ans.23: Last Steps of Encoding "that thatch thaws".

The final compressed stream consists of 1 control word followed by 11 items (9 literals and 2 copy items)
0000010010000000|01110100|01101000|01100001|01110100|00100000|0000|0011
|00000101|01100011|01101000|0000|0011|00000111|01110111|01110011.

3.8: Imagine a compression utility for a personal computer that maintains all the files (or groups of files) on the hard disk in compressed form, to save space. Such a utility should be transparent to the user; it should automatically decompress a file every time it is opened and automatically compress it when it is being closed. In order to be transparent, such a utility should be fast; with compression ratio being only a secondary feature.

I	in dict?	new entry	output	I	in dict?	new entry	output
a	Y			s	N	263-s	115 (s)
al	N	256-al	97 (a)	␣	Y		
l	Y			␣a	N	264-␣a	32 (␣)
lf	N	257-lf	108 (l)	a	Y		
f	Y			al	Y		
f	N	258-f	102 (f)	alf	N	265-alf	256 (al)
␣	Y			f	Y		
␣e	N	259-␣e	32 (w)	fa	N	266-fa	102 (f)
e	Y			a	Y		
ea	N	260-ea	101 (e)	al	Y		
a	Y			alf	Y		
at	N	261-at	97 (a)	alfa	N	267-alfa	265 (alf)
t	Y			a	Y		
ts	N	262-ts	116 (t)	a,eof	N		97 (a)
s	Y						

Table Ans.24: LZW Encoding of "alf eats alfalfa".

3.9: Table Ans.24 summarizes the steps. The output emitted by the encoder is

97 (a), 108 (l), 102 (f), 32 (␣), 101 (e), 97 (a), 116 (t), 115 (s), 32 (␣), 256 (al), 102 (f), 265 (alf), 97 (a),

and the following new entries are added to the dictionary

(256: al), (257: lf), (258: f), (259: ␣e), (260: ea), (261: at), (262: ts), (263: s), (264: ␣a), (265: alf), (266: fa), (267: alfa).

3.10: The encoder inputs the first a into I, searches and finds a in the dictionary. It inputs the next a but finds that Ix, which is now aa, is not in the dictionary. The encoder thus adds string aa to the dictionary as entry 256 and outputs the token 97 (a). Variable I is initialized to the second a. The third a is input, so Ix is the string aa, which is now in the dictionary. I becomes this string, and the fourth a is input. Ix is now aaa which is not in the dictionary. The encoder thus adds string aaa to the dictionary as entry 257 and outputs 256 (aa). I is initialized to the fourth a. Continuing this process is straightforward.

The result is that strings aa, aaa, aaaa,... are added to the dictionary as entries 256, 257, 258,..., and the output is

$$97\ (\texttt{a}),\ 256\ (\texttt{aa}),\ 257\ (\texttt{aaa}),\ 258\ (\texttt{aaaa}),\ldots$$

The output consists of pointers pointing to longer and longer strings of as. The first k pointers thus point at strings whose total length is $1+2+\cdots+k=(k+k^2)/2$.

Assuming an input stream that consists of one million as, we can find the size of the compressed output stream by solving the quadratic equation $(k+k^2)/2 = 1000000$ for the unknown k. The solution is $k \approx 1414$. The original, 8-million bit input is thus compressed into 1414 pointers, each at least 9-bit (and in practice,

probably 16-bit) long. The compression factor is thus either $8M/(1414 \times 9) \approx 628.6$ or $8M/(1414 \times 16) \approx 353.6$.

This is an impressive result but such input streams are rare (notice that this particular input can best be compressed by generating an output stream containing just "1000000 a", and without using LZW).

3.11: We simply follow the decoding steps described in the text. The results are:
1. Input 97. This is in the dictionary so set I="a" and output "a". String "ax" needs to be saved in the dictionary but x is still unknown..
2. Input 108. This is in the dictionary so set J="l" and output "l". Save "al" in entry 256. Set I="l".
3. Input 102. This is in the dictionary so set J="f" and output "f". Save "lf" in entry 257. Set I="f".
4. Input 32. This is in the dictionary so set J="␣" and output "␣". Save "f " in entry 258. Set I="␣".
5. Input 101. This is in the dictionary so set J="e" and output "e". Save "␣e" in entry 259. Set I="e".
6. Input 97. This is in the dictionary so set J="a" and output "a". Save "ea" in entry 260. Set I="a".
7. Input 116. This is in the dictionary so set J="t" and output "t". Save "at" in entry 261. Set I="t".
8. Input 115. This is in the dictionary so set J="s" and output "s". Save "ts" in entry 262. Set I="t".
9. Input 32. This is in the dictionary so set J="␣" and output "␣". Save "s " in entry 263. Set I="␣".
10. Input 256. This is in the dictionary so set J="al" and output "al". Save "␣a" in entry 264. Set I="al".
11. Input 102. This is in the dictionary so set J="f" and output "f". Save "alf" in entry 265. Set I="f".
12. Input 265. This has just been saved in the dictionary so set J="alf" and output "alf". Save "fa" in dictionary entry 266. Set I="alf".
13. Input 97. This is in the dictionary so set J="a" and output "a". Save "alfa" in entry 267 (even though it will never be used). Set I="a".
14. Read eof. Stop.

3.12: We assume that the dictionary is initialized to just the two entries (1: a) and (2: b). The encoder outputs

1 (a), 2 (b), 3 (ab), 5(aba), 4(ba), 7 (bab), 6 (abab), 9 (ababa), 8 (baba),...

and adds the new entries (3: ab), (4: ba), (5: aba), (6: abab), (7: bab), (8: baba), (9: ababa), (10:ababab), (11: babab),... to the dictionary. This regular behavior can be analyzed and the kth output pointer and dictionary entry predicted, but the effort is probably not worth it.

3.13: The answer to exercise 3.10 shows the relation between the size of the compressed file and the size of the largest dictionary string for the "worst case"

situation (input that creates the longest strings). For a 1Mbyte input stream, there will be 1,414 strings in the dictionary, the largest of which is 1,414 symbols long.

3.14: This is straightforward (Table Ans.25) but not very efficient since only one two-symbol dictionary phrase is used.

Step	Input	Output	S	Add to dict.	S'
	swiss miss				
1	s	115	s	—	s
2	w	119	w	256-sw	w
3	i	105	i	257-wi	i
4	s	115	s	258-is	s
5	s	115	s	259-ss	s
6	-	32	␣	260-s	␣
7	m	109	m	261-␣m	m
8	is	258	is	262-mis	is
9	s	115	s	263-iss	s

Table Ans.25: LZMW Compression of "swiss miss".

3.15: Table Ans.26 shows all the steps. In spite of the short input, the result is quite good (13 codes to compress 18-symbols) because the input contains concentrations of as and bs.

Step	Input	Output	S	Add to dict.	S'
	yabbadabbadabbadoo				
1	y	121	y	—	y
2	a	97	a	256-ya	a
3	b	98	b	257-ab	b
4	b	98	b	258-bb	b
5	a	97	a	259-ba	a
6	d	100	a	260-ad	a
7	ab	257	ab	261-dab	ab
8	ba	259	ba	262-abba	ba
9	dab	261	dab	263-badab	dab
10	ba	259	ba	264-dabba	ba
11	d	100	d	265-bad	d
12	o	111	o	266-do	o
13	o	111	o	267-o	o

Table Ans.26: LZMW Compression of "yabbadabbadabbadoo".

3.16: 1. The encoder starts by shifting the first two symbols `xy` to the search buffer, outputting them as literals and initializing all locations of the index table to the null pointer.

2. The current symbol is `a` (the first `a`) and the context is `xy`. It is hashed to, say, 5, but location 5 of the index table contains a null pointer, so `P` is null. Location 5 is set to point to the first `a`, which is then output as a literal. The data in the encoder's buffer is shifted to the left.

3. The current symbol is the second `a` and the context is `ya`. It is hashed to, say, 1, but location 1 of the index table contains a null pointer, so `P` is null. Location 1 is set to point to the second `a`, which is then output as a literal. The data in the encoder's buffer is shifted to the left.

4. The current symbol is the third `a` and the context is `aa`. It is hashed to, say, 2, but location 2 of the index table contains a null pointer, so `P` is null. Location 2 is set to point to the third `a`, which is then output as a literal. The data in the encoder's buffer is shifted to the left.

5. The current symbol is the fourth `a` and the context is `aa`. We know from step 4 that it is hashed to 2, and location 2 of the index table points to the third `a`. Location 2 is set to point to the fourth `a`, and the encoder tries to match the string starting with the third `a` to the string starting with the fourth `a`. Assuming that the look-ahead buffer is full of `a`s, the match length L will be the size of that buffer. The encoded value of L will be written to the compressed stream, and the data in the buffer shifted L positions to the left.

6. If the original input stream is long, more `a`'s will be shifted into the look-ahead buffer, and this step will also result in a match of length L. If only n `a`s remain in the input stream, they will be matched, and the encoded value of n output.

The compressed stream will consist of the three literals `x`, `y`, and `a`, followed by (perhaps several values of) L, and possibly ending with a smaller value.

3.17: T percent of the compressed stream is made up of literals, some appearing consecutively (and thus getting the flag "1" for two literals, half a bit per literal) and others with a match length following them (and thus getting the flag "01", one bit for the literal). We assume that two thirds of the literals appear consecutively and one third are followed by match lengths. The total number of flag bits created for literals is thus

$$\frac{2}{3}T \times 0.5 + \frac{1}{3}T \times 1.$$

A similar argument for the match lengths yields

$$\frac{2}{3}(1-T) \times 2 + \frac{1}{3}(1-T) \times 1$$

for the total number of the flag bits. We now write the equation

$$\frac{2}{3}T \times 0.5 + \frac{1}{3}T \times 1 + \frac{2}{3}(1-T) \times 2 + \frac{1}{3}(1-T) \times 1 = 1,$$

which is solved to yield $T = 2/3$. This means that if two thirds of the items in the compressed stream are literals, there would be 1 flag bit per item on the average. More literals would result in fewer flag bits.

3.18: The first three ones indicate six literals. The following 01 indicates a literal (`b`) followed by a match length (of 3). The 10 is the code of match length 3, and the last 1 indicates two more literals (`x` and `y`).

4.1: No. The definition of redundancy (Section 2.1) tells us that an image where each color appears with the same frequency has no redundancy (statistically), yet it is not necessarily random and may even be interesting and/or useful.

4.2: Figure Ans.27 shows two 32×32 matrices. The first one, a, with random (and therefore decorrelated) values and the second one, b, is its inverse (and therefore with correlated values). Their covariance matrices are also shown and it is obvious that matrix cov(a) is close to diagonal, whereas matrix cov(b) is far from diagonal. The Matlab code for this figure is also listed.

4.3: The results are shown in Table Ans.28 together with the Matlab code used to calculate it.

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	0**1000**	01100	**10000**	11000	1**1000**	10100
0000**1**	00001	0100**1**	01101	1000**1**	11001	1100**1**	10101
000**10**	00011	010**10**	01111	100**10**	11011	110**10**	10111
0001**1**	00010	0101**1**	01110	1001**1**	11010	1101**1**	10110
00**100**	00110	01**100**	01010	10**100**	11110	11**100**	10010
0010**1**	00111	0110**1**	01011	1010**1**	11111	1110**1**	10011
001**10**	00101	011**10**	01001	101**10**	11101	111**10**	10001
0011**1**	00100	0111**1**	01000	1011**1**	11100	1111**1**	10000

Table Ans.28: First 32 Binary and Gray Codes.

```
a=linspace(0,31,32); b=bitshift(a,-1);
b=bitxor(a,b); dec2bin(b)
```

Code For Table Ans.28.

4.4: One feature is the regular way in which each of the five code bits alternates periodically between 0 and 1. It is easy to write a program that will set all five bits to 0, will flip the rightmost bit after two codes have been calculated, and will flip any of the other four code bits in the middle of the period of its immediate neighbor on the right.

Another feature is the fact that the second half of the table is a mirror image of the first half, but with the most significant bit set to one. After the first half

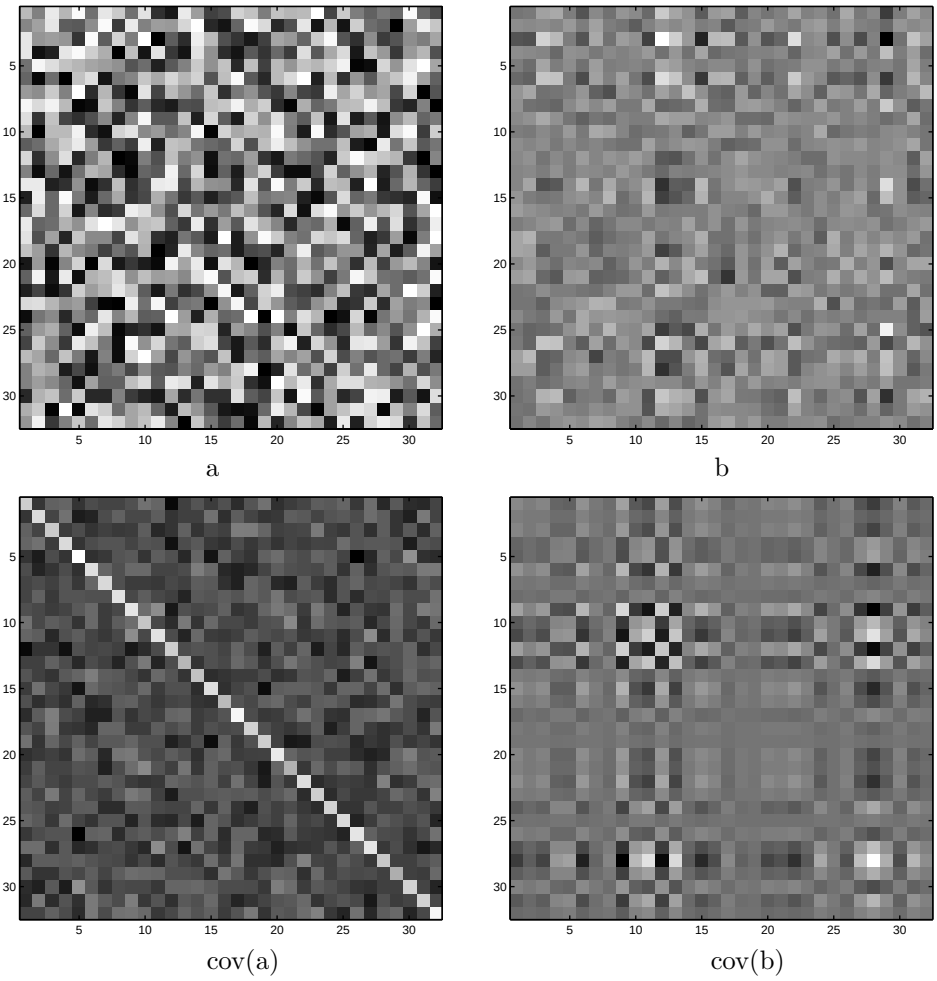


Figure Ans.27: Covariance Matrices of Correlated and Decorrelated Values

```
a=rand(32); b=inv(a);
figure(1), imagesc(a), colormap(gray); axis square
figure(2), imagesc(b), colormap(gray); axis square
figure(3), imagesc(cov(a)), colormap(gray); axis square
figure(4), imagesc(cov(b)), colormap(gray); axis square
```

Code for Figure Ans.27.

of the table has been computed, using any method, this symmetry can be used to quickly calculate the second half.

4.5: Figure Ans.29 is an *angular code wheel* representation of the 4 and 6-bit RGC codes (part a) and the 4 and 6-bit binary codes (part b). The individual bitplanes

are shown as rings, with the most significant bits as the innermost ring. It is easy to see that the maximum angular frequency of the RGC is half that of the binary code.

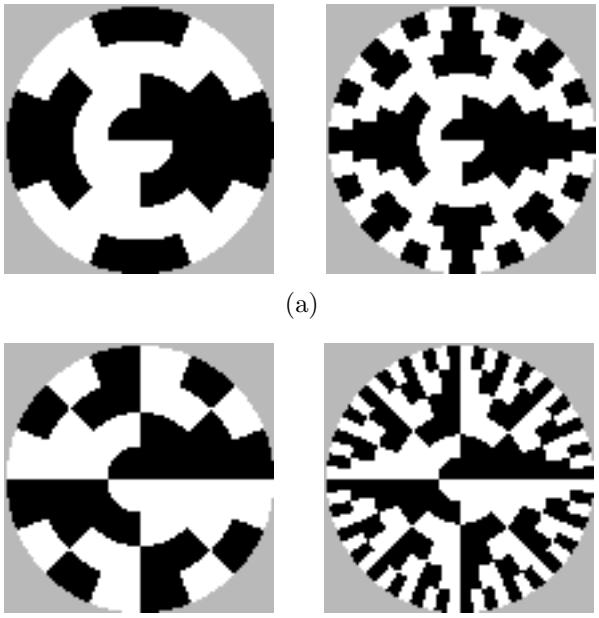


Figure Ans.29: Angular Code Wheels of RGC and Binary Codes.

4.6: No. If pixel values are in the range $[0, 255]$, a difference $(P_i - Q_i)$ can be at most 255. The worst case is where all the differences are 255. It is easy to see that such a case yields an RMSE of 255.

4.7: The code of Figure Ans.30 yields the coordinates of the rotated points

$$(7.071, 0),\ (9.19, 0.7071),\ (17.9, 0.78),\ (33.9, 1.41),\ (43.13, -2.12),$$

(notice how all the y coordinates are small numbers) and shows that the cross-correlation drops from 1729.72 before the rotation to -23.0846 after it. A significant reduction!

4.8: The eight values in the top row are close (the distances between them are either 2 or 3). Each of the other rows is obtained as a right-circular shift of the preceding row.

```
p={{5,5},{6, 7},{12.1,13.2},{23,25},{32,29}};
rot={{0.7071,-0.7071},{0.7071,0.7071}};
Sum[p[[i,1]]p[[i,2]], {i,5}]
q=p.rot
Sum[q[[i,1]]q[[i,2]], {i,5}]
```

Figure Ans.30: Code For Rotating Five Points.

4.9: It is obvious that such a block can be represented as a linear combination of the patterns in the leftmost column of Figure 4.21. The actual calculation yields the eight weights 4, 0.72, 0, 0.85, 0, 1.27, 0, and 3.62 for the patterns of this column. The other 56 weights are zero or very close to zero.

4.10: The *Mathematica* code below produces the eight coefficients

$$140, -71, 0, -7, 0, -2, 0, 0.$$

After clearing the last two nonzero weights (-7 and -2) and applying the one-dimensional IDCT, Equation(4.8),

```
DCT[i_]:={(1/2)Cr[i]Sum[Pixl[[x+1]]Cos[(2x+1)i Pi/16], {x,0,7,1}]};
  IDCT[x_]:={(1/2)Sum[Cr[i]G[[i+1]]Cos[(2x+1)i Pi/16], {i,0,7,1}]};
```

to the sequence $140, -71, 0, 0, 0, 0, 0, 0$, we get $15, 20, 30, 43, 56, 69, 79$, and 84. These are not identical to the original values, but the maximum difference is only 4.

4.11: Figure Ans.31 shows the results and the Matlab code. Notice that the same code can also be used to calculate and display the DCT basis images.

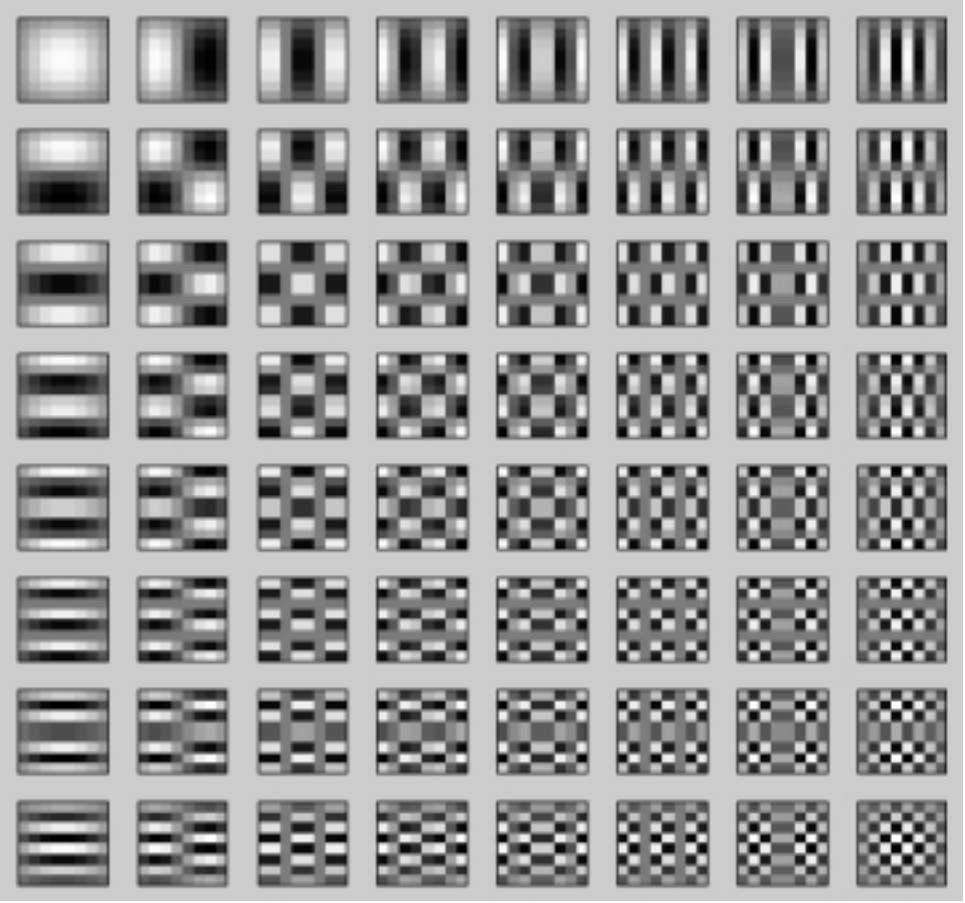
4.12: Figure Ans.32 shows the 64 basis images and the Matlab code to calculate and display them. Each basis image is an 8×8 matrix.

4.13: $\mathbf{A}_4$ is the 4×4 matrix

$$\mathbf{A}_4 = \begin{pmatrix} h_0(0/4) & h_0(1/4) & h_0(2/4) & h_0(3/4) \\ h_1(0/4) & h_1(1/4) & h_1(2/4) & h_1(3/4) \\ h_2(0/4) & h_2(1/4) & h_2(2/4) & h_2(3/4) \\ h_3(0/4) & h_3(1/4) & h_3(2/4) & h_3(3/4) \end{pmatrix} = \frac{1}{\sqrt{4}} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & -\sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{pmatrix}.$$

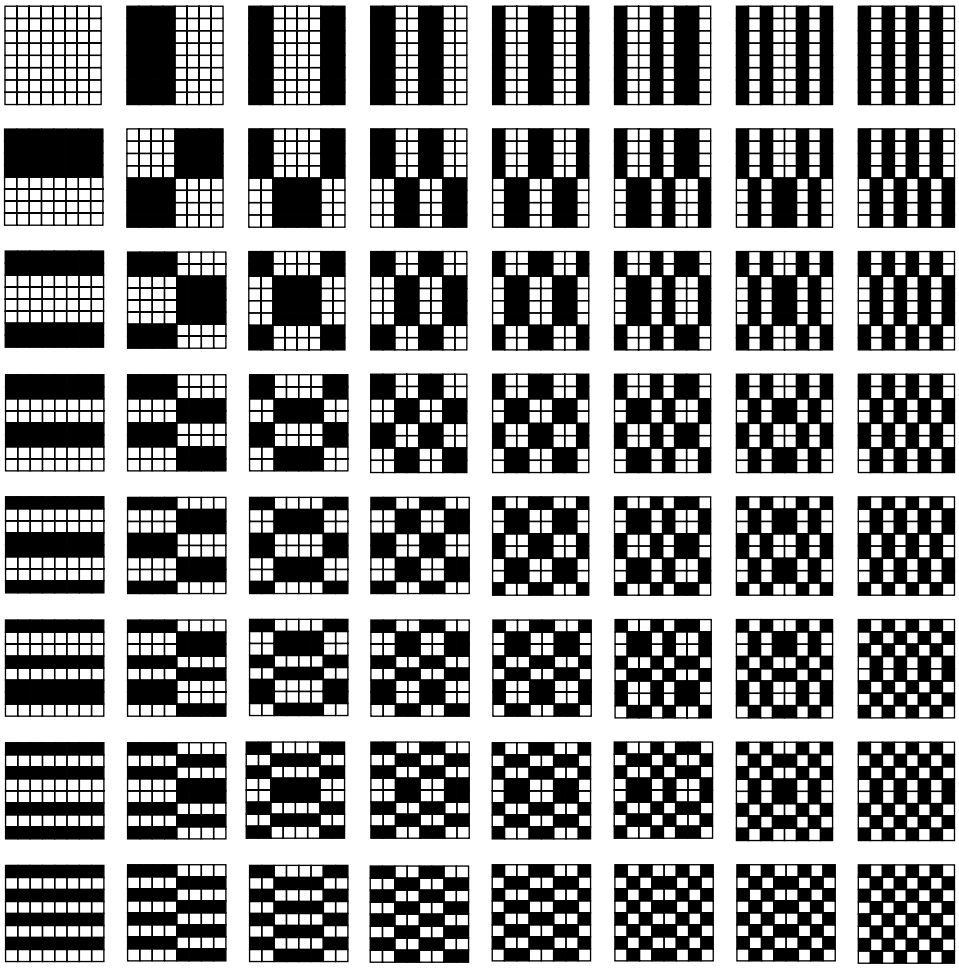
Similarly, $\mathbf{A}_8$ is the matrix

$$\mathbf{A}_8 = \frac{1}{\sqrt{8}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{pmatrix}$$



```
N=8;
m=[1:N]'*ones(1,N); n=m';
% can also use cos instead of sin
%A=sqrt(2/N)*cos(pi*(2*(n-1)+1).*(m-1)/(2*N));
A=sqrt(2/N)*sin(pi*(2*(n-1)+1).*(m-1)/(2*N));
A(1,:)=sqrt(1/N);
C=A';
for row=1:N
  for col=1:N
    B=C(:,row)*C(:,col).'; %tensor product
    subplot(N,N,(row-1)*N+col)
    imagesc(B)
    drawnow
  end
end
```

Figure Ans.31: The 64 Basis Images of the Two-Dimensional DST.



```
M=3; N=2^M; H=[1 1; 1 -1]/sqrt(2);
for m=1:(M-1) % recursion
  H=[H H; H -H]/sqrt(2);
end
A=H';
map=[1 5 7 3 4 8 6 2]; % 1:N
for n=1:N, B(:,n)=A(:,map(n)); end;
A=B;
sc=1/(max(abs(A(:))).^2); % scale factor
for row=1:N
  for col=1:N
    BI=A(:,row)*A(:,col).'; % tensor product
    subplot(N,N,(row-1)*N+col)
    oe=round(BI*sc); % results in -1, +1
    imagesc(oe), colormap([1 1 1; .5 .5 .5; 0 0 0])
    drawnow
  end
end
```

Figure Ans.32: The 8×8 WHT Basis Images and Matlab Code.

4.14: The average of vector $\mathbf{w}^{(i)}$ is zero, so Equation (4.17) yields

$$\left(\mathbf{W}\cdot\mathbf{W}^T\right)_{jj}=\sum_{i=1}^{k}w_j^{(i)}w_j^{(i)}=\sum_{i=1}^{k}\left(w_j^{(i)}-0\right)^2=\sum_{i=1}^{k}\left(c_i^{(j)}-0\right)^2=k\,\text{Variance}(\mathbf{c}^{(j)}).$$

4.15: The arguments of the cosine functions used by the DCT are of the form $(2x+1)i\pi/16$, where i and x are integers in the range $[0,7]$. Such an argument can be written in the form $n\pi/16$, where n is an integer in the range $[0,15\times 7]$. Since the cosine function is periodic, it satisfies $\cos(32\pi/16)=\cos(0\pi/16)$, $\cos(33\pi/16)=\cos(\pi/16)$, and so on. As a result, only the 32 values $\cos(n\pi/16)$ for $n=0,1,2,\ldots,31$ are needed. The author is indebted to V. Saravanan for pointing out this feature of the DCT.

4.16: When the following *Mathematica*™ code is applied to Table 4.53b it creates a data unit with 64 pixels, all having the value 140, which is the average value of the pixels in the original data unit 4.50.

```
Cr[i_]:=If[i==0, 1/Sqrt[2], 1];
IDCT[x_,y_]:={(1/4)Sum[Cr[i]Cr[j]G[[i+1,j+1]]Quant[[i+1,j+1]]*
 Cos[(2x+1)i Pi/16]Cos[(2y+1)j Pi/16], {i,0,7,1}, {j,0,7,1}]};
```

4.17: Selecting $R=1$ has produced the quantization coefficients of Table Ans.33a and the quantized data unit of Table Ans.33b. This table has 18 nonzero coefficients which, when used to reconstruct the original data unit, produce Table Ans.34, only a small improvement over Table 4.51.

4.18: The zigzag sequence is $1118,2,0,-2,\underbrace{0,\ldots,0}_{13},-1,0,\ldots$ (there are only four nonzero coefficients).

4.19: Perhaps the simplest way is to manually figure out the zigzag path and to record it in an array `zz` of structures, where each structure contains a pair of coordinates for the path as shown, e.g., in Figure Ans.35.

If the two components of a structure are `zz.r` and `zz.c`, then the zig-zag traversal can be done by a loop of the form :

```
for (i=0; i<64; i++){
row:=zz[i].r; col:=zz[i].c
...data_unit[row][col]...}
```

4.20: It is located in row 3 column 5, so it is encoded as 1110|101.

4.21: Thirteen consecutive zeros precede this coefficient, so Z = 13. The coefficient itself is found in Table 4.55 in row 1, column 0, so R = 1 and C = 0. Assuming that the Huffman code in position (R, Z) = (1, 13) of Table 4.58 is 1110101, the final code emitted for 1 is 1110101|0.

1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15

(a)

1118.	3	2	-1	1	0	0	0
-1	0	1	1	1	0	0	0
-3	-2	0	-2	0	0	0	0
-1	-2	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	-1	1	0	0	0
0	0	0	1	0	0	0	0

(b)

Table Ans.33: (a): The Quantization table $1 + (i + j) \times 1$. (b): Quantized Coefficients Produced by (a).

139	139	138	139	139	138	139	140
140	140	140	139	139	139	139	140
142	141	140	140	140	139	139	140
142	141	140	140	140	140	139	139
142	141	140	140	140	140	140	139
140	140	140	140	139	139	139	141
140	140	140	140	139	139	140	140
139	140	141	140	139	138	139	140

Table Ans.34: Restored data unit of Table 4.50.

(0,0)	(0,1)	(1,0)	(2,0)	(1,1)	(0,2)	(0,3)	(1,2)
(2,1)	(3,0)	(4,0)	(3,1)	(2,2)	(1,3)	(0,4)	(0,5)
(1,4)	(2,3)	(3,2)	(4,1)	(5,0)	(6,0)	(5,1)	(4,2)
(3,3)	(2,4)	(1,5)	(0,6)	(0,7)	(1,6)	(2,5)	(3,4)
(4,3)	(5,2)	(6,1)	(7,0)	(7,1)	(6,2)	(5,3)	(4,4)
(3,5)	(2,6)	(1,7)	(2,7)	(3,6)	(4,5)	(5,4)	(6,3)
(7,2)	(7,3)	(6,4)	(5,5)	(4,6)	(3,7)	(4,7)	(5,6)
(6,5)	(7,4)	(7,5)	(6,6)	(5,7)	(6,7)	(7,6)	(7,7)

Figure Ans.35: Coordinates for the Zigzag Path.

4.22: This is shown by multiplying the largest four n-bit number, $\underbrace{11\ldots1}_{n}$ by 4, which is easily done by shifting it 2 positions to the left. The result is the $n+2$-bit number $\underbrace{11\ldots1}_{n}00$.

4.23: Simple growth rules make for a more natural progressive growth of the image. They make it easier for a person watching the image develop on the screen to decide if and when to stop the decoding process and accept or discard the image.

4.24: The only specification that depends on the particular bits assigned to the two colors is Equation (4.19). All the other parts of JBIG are independent of the

bit assignment.

4.25: For the 16-bit template of Figure 4.89a the relative coordinates are

$$A_1 = (3, -1), \quad A_2 = (-3, -1), \quad A_3 = (2, -2), \quad A_4 = (-2, -2).$$

For the 13-bit template of Figure 4.89b the relative coordinates of A_1 are $(3, -1)$. For the 10-bit templates of Figure 4.89c,d the relative coordinates of A_1 are $(2, -1)$.

4.26: It produces better compression in cases where the text runs vertically.

4.27: Going back to step 1 we have the same points participate in the partition for each codebook entry (this happens because our points are concentrated in four distinct regions, but in general a partition $P_i^{(k)}$ may consist of different image blocks in each iteration k). The distortions calculated in step 2 are summarized in Table Ans.37. The average distortion $D_i^{(1)}$ is

$$D^{(1)} = (277+277+277+277+50+50+200+117+37+117+162+117)/12 = 163.17,$$

much smaller than the original 603.33. If step 3 indicates no convergence, more iterations should follow (Exercise 4.28), reducing the average distortion and improving the values of the four codebook entries.

4.28: Each new codebook entry $C_i^{(k)}$ is calculated, in step 4 of iteration k, as the average of the block images comprising partition $P_i^{(k-1)}$. In our example the image blocks (points) are concentrated in four separate regions, so the partitions calculated for iteration $k = 1$ are the same as those for $k = 0$. Another iteration, for $k = 2$, will therefore compute the same partitions in its step 1 yielding, in step 3, an average distortion $D^{(2)}$ that equals $D^{(1)}$. Step 3 will therefore indicate convergence.

4.29: It is $4^{-8} \approx 0.000015$ the area of the entire space.

4.30: Monitor the compression ratio and delete the dictionary and start afresh each time compression performance drops below a certain threshold.

4.31: Here are steps 4 and 5. Step 4: Point $(2, 0)$ is popped out of the GPP. The pixel value at this position is 7. The best match for this point is with the dictionary entry containing 7. The encoder outputs the pointer 7. The match does not have any concave corners, so we push the point on the right of the matched block, $(2, 1)$, and the point below it, $(3, 0)$, into the GPP. The GPP now contains points $(2, 1)$, $(3, 0)$, $(0, 2)$, and $(1, 1)$. The dictionary is updated by appending to it (at location 18) the block $\boxed{\frac{4}{7}}$.

Step 5: Point $(1, 1)$ is popped out of the GPP. The pixel value at this position is 5. The best match for this point is with the dictionary entry containing 5. The encoder outputs the pointer 5. The match does not have any concave corners, so

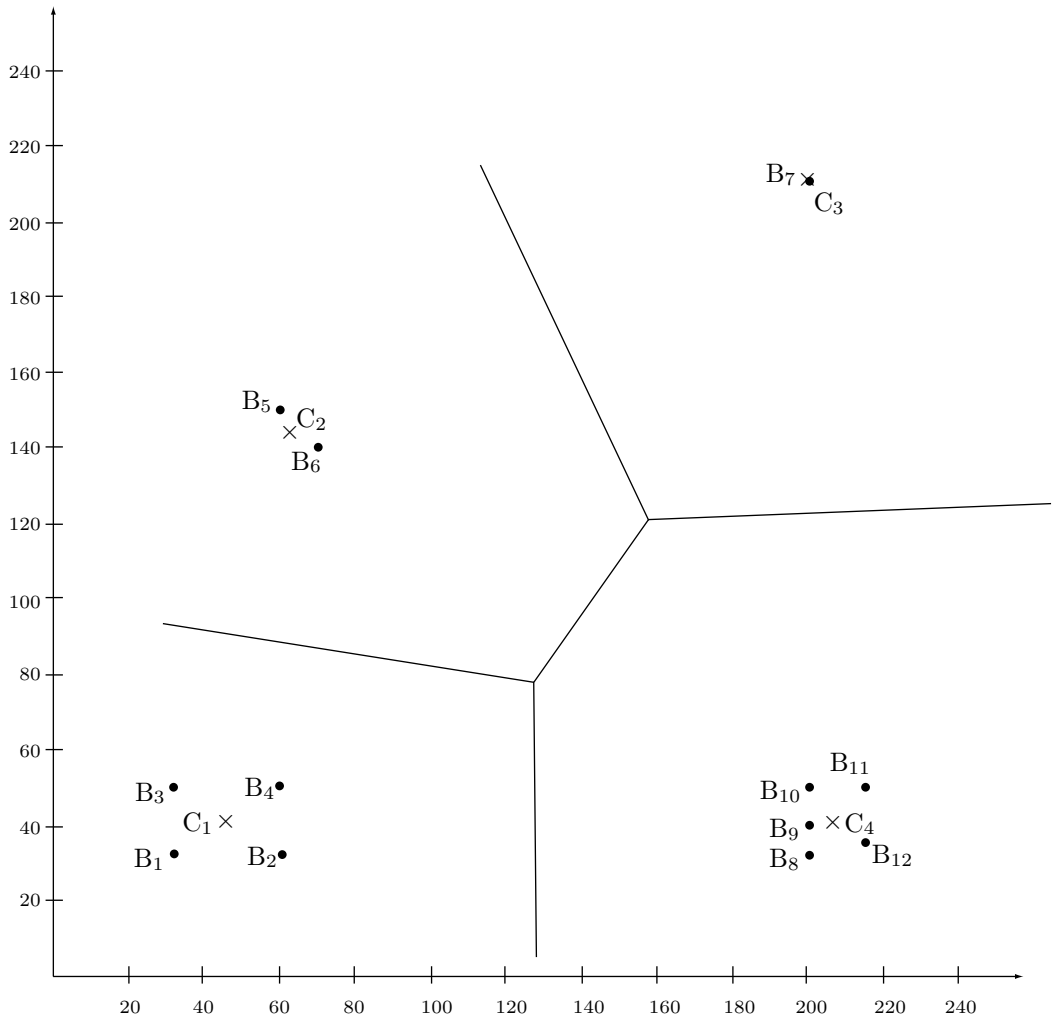


Figure Ans.36: Twelve Points and Four Codebook Entries $C_i^{(1)}$.

I:	$(46-32)^2+(41-32)^2=277$,	$(46-60)^2+(41-32)^2=277$,
	$(46-32)^2+(41-50)^2=277$,	$(46-60)^2+(41-50)^2=277$,
II:	$(65-60)^2+(145-150)^2=\ 50$,	$(65-70)^2+(145-140)^2=\ 50$,
III:	$(210-200)^2+(200-210)^2=200$,	
IV:	$(206-200)^2+(41-32)^2=117$,	$(206-200)^2+(41-40)^2=\ 37$,
	$(206-200)^2+(41-50)^2=117$,	$(206-215)^2+(41-50)^2=162$,
	$(206-215)^2+(41-35)^2=117$.	

Table Ans.37: Twelve Distortions For $k=1$.

we push the point to the right of the matched block, $(1,2)$, and the point below it, $(2,1)$, into the GPP. The GPP contains points $(1,2)$, $(2,1)$, $(3,0)$, and $(0,2)$. The dictionary is updated by appending to it (at locations 19, 20) the two blocks $\boxed{\begin{matrix}2\\5\end{matrix}}$ and $\boxed{4|5}$.

4.32: It may simply be too long. When compressing text, each symbol is normally 1-byte long (2 bytes in Unicode). However, images with 24-bit pixels are very common, and a 16-pixel block in such an image is 48-bytes long.

4.33: If the encoder uses a $(2,1,k)$ general unary code, then the value of k should be included in the header.

4.34: The mean and standard deviation are $\bar{p}=115$ and $\sigma=77.93$, respectively. The counts become $n^+=n^-=8$, and Equations (4.27) are solved to yield $p^+=193$ and $p^-=37$. The original block is compressed to the 16 bits

$$\begin{pmatrix}1&0&1&1\\1&0&0&1\\1&1&0&1\\0&0&0&0\end{pmatrix},$$

and the two 8-bit values 37 and 193.

4.35: Table Ans.38 summarizes the results. Notice how a 1-pixel with a context of 00 is assigned high probability after being seen 3 times.

#	Pixel	Context	Counts	Probability	New counts
5	0	10=2	1,1	1/2	2,1
6	1	00=0	1,3	3/4	1,4
7	0	11=3	1,1	1/2	2,1
8	1	10=2	2,1	1/3	2,2

Table Ans.38: Counts and Probabilities for Next four Pixels.

4.36: Such a thing is possible for the encoder but not for the decoder. A compression method using "future" pixels in the context is useless because its output would be impossible to decompress.

4.37: A 2nd order Markov model. In such a model the value of the current data item depends on just two of its past neighbors, not necessarily the two immediate ones.

4.38: The two previously seen neighbors of P=8 are A=1 and B=11. P is thus in the central region, where all codes start with a zero, and L=1, H=11. The computations are straightforward:

$$k = \lfloor \log_2(11 - 1 + 1) \rfloor = 3, \qquad a = 2^{3+1} - 11 = 5, \qquad b = 2(11 - 2^3) = 6.$$

Table Ans.39 lists the five 3-bit codes and six 4-bit codes for the central region. The code for 8 is thus 0|111.

The two previously seen neighbors of P=7 are A=2 and B=5. P is thus in the right outer region, where all codes start with 11, and L=2, H=7. We are looking for the code of $7 - 5 = 2$. Choosing $m = 1$ yields, from Table 4.112, the code 11|01.

The two previously seen neighbors of P=0 are A=3 and B=5. P is thus in the left outer region, where all codes start with 10, and L=3, H=5. We are looking for the code of $3 - 0 = 3$. Choosing $m = 1$ yields, from Table 4.112, the code 10|100.

Pixel P	Region code	Pixel code
1	0	0000
2	0	0010
3	0	0100
4	0	011
5	0	100
6	0	101
7	0	110
8	0	111
9	0	0001
10	0	0011
11	0	0101

Table Ans.39: The Codes for a Central Region.

4.39: Because the decoder has to resolve ties in the same way as the encoder.

4.40: Because this will result in a weighted sum whose value is in the same range as the values of the pixels. If pixel values are, e.g., in the range $[0, 15]$ and the weights add up to 2, a prediction may result in values of up to 30.

4.41: Each of the three weights 0.0039, −0.0351, and 0.3164 is used twice. The sum of the weights is thus 0.5704 and the result of dividing each weight by this sum is 0.0068, −0.0615, and 0.5547. It is easy to verify that the sum of the renormalized weights $2(0.0068 - 0.0615 + 0.5547)$ equals 1.

4.42: One such example is an archive of static images. NASA has a large archive of images taken by various satellites. They should be kept highly compressed, but they never change so each image has to be compressed only once. A slow encoder is therefore acceptable but a fast decoder is certainly handy. Another example is an art collection. Many museums have digitized their collections of paintings, and those are also static.

4.43: The decoder knows this pixel since it knows the value of average $\mu[i-1,j] = 0.5(I[2i-2,2j]+I[2i-1,2j+1])$ and since it has already decoded pixel $I[2i-2,2j]$

4.44: When the decoder inputs the 5, it knows that the difference between p (the pixel being decoded) and the reference pixel starts at position 6 (counting from left). Since bit 6 of the reference pixel is 0, that of p must be 1.

4.45: Yes, but compression would suffer. One way to apply this method is to separate each byte into two 4-bit pixels and encode each pixel separately. This approach is bad since the prefix and suffix of a 4-bit pixel may often consist of more than 4 bits. Another approach is to ignore the fact that a byte contains two pixels, and use the method as originally described. This may still compress the image, but is not very efficient, as the following example illustrates.

Example: The two bytes 1100|1101 and 1110|1111 represent four pixels, each differing from its immediate neighbor by its least significant bit. The four pixels thus have similar colors (or grayscales). Comparing consecutive pixels results in prefixes of 3 or 2, but comparing the 2 bytes produces the prefix 2.

4.46: Because this produces a value X in the same range as A, B, and C. If the weights were, for instance, 1, 100, and 1, X would have much bigger values than any of the three pixels.

4.47: the four vectors are

$$\begin{aligned}\mathbf{a} &= (90,95,100,80,90,85),\\ \mathbf{b}^{(1)} &= (100,90,95,102,80,90),\\ \mathbf{b}^{(2)} &= (101,128,108,100,90,95),\\ \mathbf{b}^{(3)} &= (128,108,110,90,95,100),\end{aligned}$$

and the code of Figure Ans.40 produces the solutions $w_1 = 0.1051$, $w_2 = 0.3974$, and $w_3 = 0.3690$. Their total is 0.8715, compared with the original solutions, which added up to 0.9061. The point is that the numbers involved in the equations (the elements of the four vectors) are not independent (for example, pixel 80 appears in $\mathbf{a}$ and in $\mathbf{b}^{(1)}$) except for the last element (85 or 91) of $\mathbf{a}$ and the first element 101 of $\mathbf{b}^{(2)}$, which are independent. Changing these two elements affects the solutions, which is why the solutions do not always add up to unity. However, compressing nine pixels produces solutions whose total is closer to one than in the case of six pixels. Compressing an entire image, with many thousands of pixels, produces solutions whose sum is very close to 1.

4.48: Figure Ans.41a,b,c shows the results, with all H_i values shown in small type. Most H_i values are zero because the pixels of the original image are so highly correlated. The H_i values along the edges are very different because of the simple edge rule used. The result is that the H_i values are highly decorrelated and have low entropy. Thus, they are candidates for entropy coding.

```
a={90.,95,100,80,90,85};
b1={100,90,95,100,80,90};
b2={100,128,108,100,90,95};
b3={128,108,110,90,95,100};
Solve[{b1.(a-w1 b1-w2 b2-w3 b3)==0,
b2.(a-w1 b1-w2 b2-w3 b3)==0,
b3.(a-w1 b1-w2 b2-w3 b3)==0},{w1,w2,w3}]
```

Figure Ans.40: Solving For Three Weights.

1	.	3	.	5	.	7	.	1	.	7	.	5	.	5	.
.	0	.	0	.	0	.	-5	.	.	.	.	.	.	.	.
17	.	19	.	21	.	23	.	15	.	19	.	11	.	23	.
.	0	.	0	.	0	.	-13	.	.	.	.	.	.	.	.
33	.	35	.	37	.	39	.	33	.	0	.	37	.	0	.
.	0	.	0	.	0	.	-21	.	.	.	.	.	.	.	.
49	.	51	.	53	.	55	.	-33	.	51	.	-35	.	55	.
.	-33	.	-34	.	-35	.	-64	.	.	.	.	.	.	.	.

(a) (b)

1	.	.	.	5	.	.	.	
.	.	.	.	.	.	.	.	
.	.	0	.	.	.	-5	.	
.	.	.	.	.	.	.	.	
33	.	.	.	37	.	.	.	
.	.	.	.	.	.	.	.	
.	.	-33	.	.	.	-55	.	
.	.	.	.	.	.	.	.	

(c)

Figure Ans.41: (a) Bands L_2 and H_2. (b) Bands L_3 and H_3. (c) Bands L_4 and H_4.

4.49: There are 16 values. The value 0 appears nine times, and each of the other seven values appears once. The entropy is thus

$$-\sum p_i \log_2 p_i = -\frac{9}{16}\log_2\left(\frac{9}{16}\right) - 7\frac{1}{16}\log_2\left(\frac{1}{16}\right) \approx 2.2169.$$

Not very small, since seven of the 16 values have the same probability. In practice, values of an H_i difference band tend to be small, are both positive and negative, and are concentrated around zero, so their entropy is small.

4.50: Because the decoder needs to know how the encoder estimated X for each H_i difference value. If the encoder uses one of three methods for prediction, it has to precede each difference value in the compressed stream with a code that tells the decoder which method was used. Such a code can have variable size (for example, 0, 10, 11) but even adding just one or two bits to each prediction reduces compression performance significantly, since each H_i value needs to be predicted, and the number of these values is close to the size of the image.

4.51: The binary tree is shown in Figure Ans.42. From this tree, it is easy to see that the progressive image file is 3 6|5 7|7 7 10 5.

4.52: They are shown in Figure Ans.43

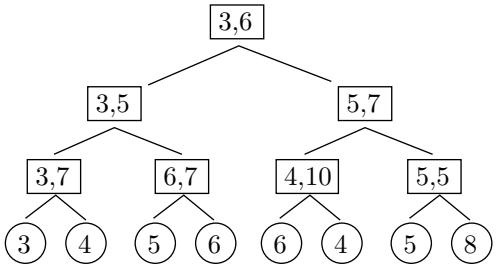


Figure Ans.42: A Binary Tree For An 8-Pixel Image.

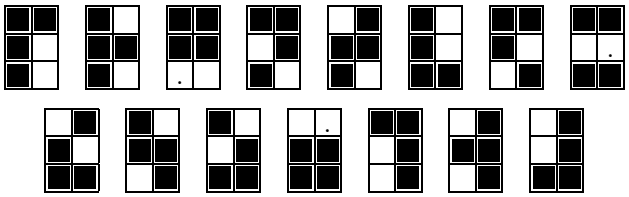


Figure Ans.43: The 15 6-Tuples With Two White Pixels.

4.53: No. An image with little or no correlation between the pixels will not compress with quadrisection, even though the size of the last matrix is always small. Even without knowing the details of quadrisection we can confidently state that such an image will produce a sequence of matrices M_j with few or no identical rows. In the extreme case, where the rows of any M_j are all distinct, each M_j will have four times the number of rows of its predecessor. This will create indicator vectors I_j that get longer and longer, thereby increasing the size of the compressed stream and reducing the overall compression performance.

4.54: This is just the concatenation of the 12 distinct rows of M_4

$$M_5^T = (0000|0001|1111|0011|1010|1101|1000|0111|1110|0101|1011|0010).$$

4.55: M_4 has four columns, so it can have at most 16 distinct rows, implying that M_5 can have at most $4\times 16 = 64$ elements.

4.56: The decoder has to read the entire compressed stream, save it in memory, and start the decoding with L_5. Grouping the eight elements of L_5 yields the four distinct elements 01, 11, 00, and 10 of L_4, so I_4 can now be used to reconstruct L_4. The four zeros of I_4 correspond to the four distinct elements of L_4, and the remaining 10 elements of L_4 can be constructed from them. Once L_4 has been constructed, its 14 elements are grouped to form the seven distinct elements of L_3. These elements are 0111, 0010, 1100, 0110, 1111, 0101, and 1010, and they correspond to the seven zeros of I_3. Once L_3 has been constructed, its eight elements are grouped to form

the four distinct elements of L_2. Those four elements are the entire L_2 since I_2 is all zero. Reconstructing L_1 and L_0 is now trivial.

4.57: The two halves of L_0 are distinct, so L_1 consists of the two elements

$$L_1 = (0101010101010101, 1010101010101010),$$

and the first indicator vector is $I_1 = (0, 0)$. The two elements of L_1 are distinct, so L_2 has the four elements

$$L_2 = (01010101, 01010101, 10101010, 10101010),$$

and the second indicator vector is $I_2 = (0, 1, 0, 2)$. Two elements of L_2 are distinct, so L_3 has the four elements $L_3 = (0101, 0101, 1010, 1010)$, and the third indicator vector is $I_3 = (0, 1, 0, 2)$. Again two elements of L_3 are distinct, so L_4 has the four elements $L_4 = (01, 01, 10, 10)$, and the fourth indicator vector is $I_4 = (0, 1, 0, 2)$. Only two elements of L_4 are distinct, so L_5 has the four elements $L_5 = (0, 1, 1, 0)$.

The output thus consists of $k = 5$, the value 2 (indicating that I_2 is the first nonzero vector) I_2, I_3, and I_4 (encoded), followed by $L_5 = (0, 1, 1, 0)$.

4.58: Using a Hilbert curve produces the 21 runs 5, 1, 2, 1, 2, 7, 3, 1, 2, 1, 5, 1, 2, 2, 11, 7, 2, 1, 1, 1, 6. RLE produces the 27 runs 0, 1, 7, eol, 2, 1, 5, eol, 5, 1, 2, eol, 0, 3, 2, 3, eol, 0, 3, 2, 3, eol, 0, 3, 2, 3, eol, 4, 1, 3, eol, 3, 1, 4, eol.

4.59: The string 2011.

4.60: This particular numbering makes it easy to convert between the number of a subsquare and its image coordinates. (We assume that the origin is located at the bottom-left corner of the image and that image coordinates vary from 0 to 1.) As an example, translating the digits of the number 1032 to binary results in (01)(00)(11)(10). The first bits of these groups constitute the x coordinate of the subsquare, and the second bits constitute the y coordinate. Thus, the image coordinates of subsquare 1032 are $x = .0011_2 = 3/16$ and $y = .1010_2 = 5/8$, as can be directly verified from Figure 4.152c.

4.61: This is shown in Figure Ans.44.

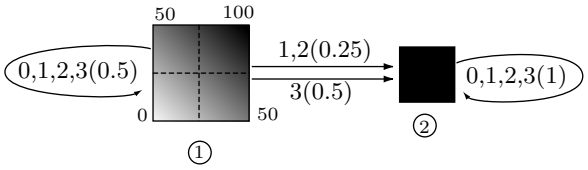


Figure Ans.44: A Two-State Graph.

4.62: The function is

$$f(x,y)=\begin{cases} x+y, & \text{if } x+y\le 1,\\ 0, & \text{if } x+y>1.\end{cases}$$

4.63: The graph has five states, so each transition matrix is of size 5×5. Direct computation from the graph yields

$$W_0=\begin{pmatrix} 0 & 1 & 0 & 0 & 0\\ 0 & 0.5 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1\\ 0 & -0.5 & 0 & 0 & 1.5\\ 0 & -0.25 & 0 & 0 & 1\end{pmatrix},\quad W_3=\begin{pmatrix} 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & -1 & 0 & 0 & 1.5\end{pmatrix},$$

$$W_1=W_2=\begin{pmatrix} 0 & 0 & 1 & 0 & 0\\ 0 & 0.25 & 0 & 0 & 0.5\\ 0 & 0 & 0 & 1.5 & 0\\ 0 & 0 & -0.5 & 1.5 & 0\\ 0 & -0.375 & 0 & 0 & 1.25\end{pmatrix}.$$

The final distribution is the five-component vector

$$F=(0.25,0.5,0.375,0.4125,0.75)^T.$$

4.64: One way to specify the center is to construct string $033\ldots3$. This yields

$$\begin{aligned}\psi_i(03\ldots3)&=(W_0\cdot W_3\cdots W_3\cdot F)_i\\ &=\begin{pmatrix}0.5 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}0.5 & 0.5\\ 0 & 1\end{pmatrix}\cdots\begin{pmatrix}0.5 & 0.5\\ 0 & 1\end{pmatrix}\begin{pmatrix}0.5\\ 1\end{pmatrix}_i\\ &=\begin{pmatrix}0.5 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}0 & 1\\ 0 & 1\end{pmatrix}\begin{pmatrix}0.5\\ 1\end{pmatrix}_i=\begin{pmatrix}0.5\\ 1\end{pmatrix}_i.\end{aligned}$$

4.65: Figure Ans.45 shows Matlab code for such a matrix.

```
dim=256;
for i=1:dim
 for j=1:dim
   m(i,j)=(i+j-2)/(2*dim-2);
 end
end
m
```

Figure Ans.45: Matlab Code For A Matrix $m_{i,j}=(i+j)/2$.

4.66: A direct examination of the graph yields the ψ_i values

$$\begin{aligned}
\psi_i(0) &= (W_0 \cdot F)_i = (0.5, 0.25, 0.75, 0.875, 0.625)_i^T,\\
\psi_i(01) &= (W_0 \cdot W_1 \cdot F)_i = (0.5, 0.25, 0.75, 0.875, 0.625)_i^T,\\
\psi_i(1) &= (W_1 \cdot F)_i = (0.375, 0.5, 0.61875, 0.43125, 0.75)_i^T,\\
\psi_i(00) &= (W_0 \cdot W_0 \cdot F)_i = (0.25, 0.125, 0.625, 0.8125, 0.5625)_i^T,\\
\psi_i(03) &= (W_0 \cdot W_3 \cdot F)_i = (0.75, 0.375, 0.625, 0.5625, 0.4375)_i^T,\\
\psi_i(3) &= (W_3 \cdot F)_i = (0, 0.75, 0, 0, 0.625)_i^T,
\end{aligned}$$

and the f values

$$\begin{aligned}
&f(0) = I \cdot \psi(0) = 0.5, \quad f(01) = I \cdot \psi(01) = 0.5, \quad f(1) = I \cdot \psi(1) = 0.375,\\
&f(00) = I \cdot \psi(00) = 0.25, \quad f(03) = I \cdot \psi(03) = 0.75, \quad f(3) = I \cdot \psi(3) = 0.
\end{aligned}$$

4.67: Figure Ans.46a,b shows the six states and all 21 edges. We use the notation $i(q,t)j$ for the edge with quadrant number q and transformation t from state i to state j. This GFA is more complex than pervious ones since the original image is less self-similar.

4.68: The transformation can be written $(x, y) \to (x, -x + y)$, so $(1, 0) \to (1, -1)$, $(3, 0) \to (3, -3)$, $(1, 1) \to (1, 0)$ and $(3, 1) \to (3, -2)$. The original rectangle is thus transformed into a parallelogram.

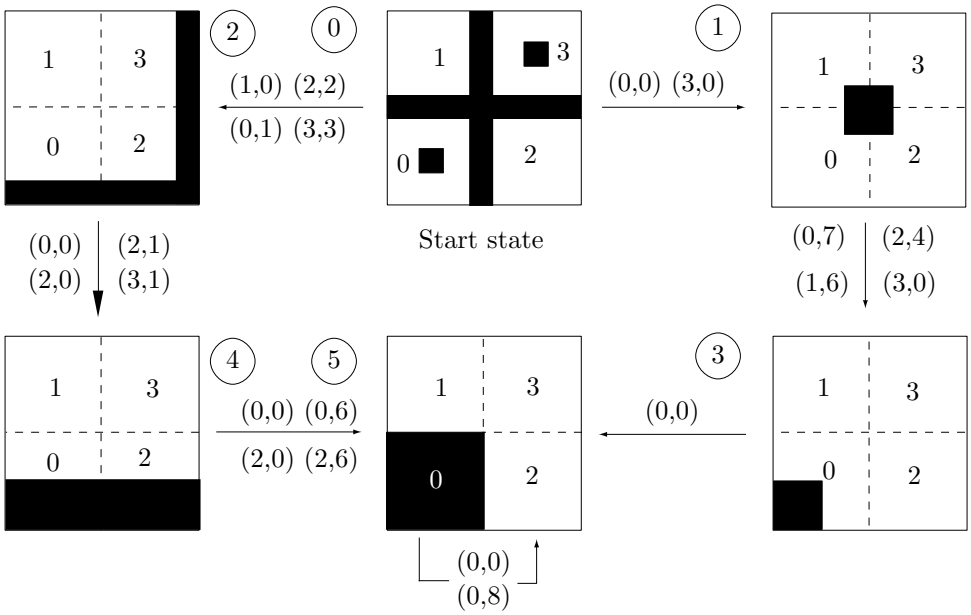
4.69: The two sets of transformations produce the same Sierpiński triangle but at different sizes and orientations.

4.70: All three transformations shrink an image to half its original size. In addition, w_2 and w_3 place two copies of the shrunken image at relative displacements of $(0, 1/2)$ and $(1/2, 0)$, as shown in Figure Ans.47. The result is the familiar Sierpiński gasket but in a different orientation.

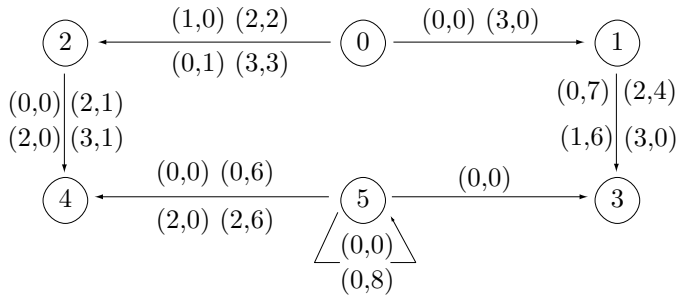
4.71: There are $32 \times 32 = 1,024$ ranges and $(256 - 15) \times (256 - 15) = 58,081$ domains. The total number of steps is thus $1,024 \times 58,081 \times 8 = 475,799,552$, still a large number. PIFS is thus computationally intensive.

4.72: Suppose that the image has G levels of gray. A good measure of data loss is the difference between the value of an average decompressed pixel and its correct value, expressed in number of gray levels. For large values of G (hundreds of gray levels) an average difference of $\log_2 G$ gray levels (or fewer) is considered satisfactory.

5.1: A written page. A person can place marks on a page and read them later as text, mathematical expressions, and drawings. This is a two-dimensional representation of the information on the page. The page can later be scanned by, e.g., a fax machine, and its contents transmitted as a one-dimensional stream of bits that constitute a different representation of the same information.



(a)



(b)

0(0,0)1	0(3,0)1	0(0,1)2	0(1,0)2	0(2,2)2	0(3,3)2	1(0,7)3
1(1,6)3	1(2,4)3	1(3,0)3	2(0,0)4	2(2,0)4	2(2,1)4	2(3,1)4
3(0,0)5	4(0,0)5	4(0,6)5	4(2,0)5	4(2,6)5	5(0,0)5	5(0,8)5

Figure Ans.46: A GFA for Exercise 4.67.

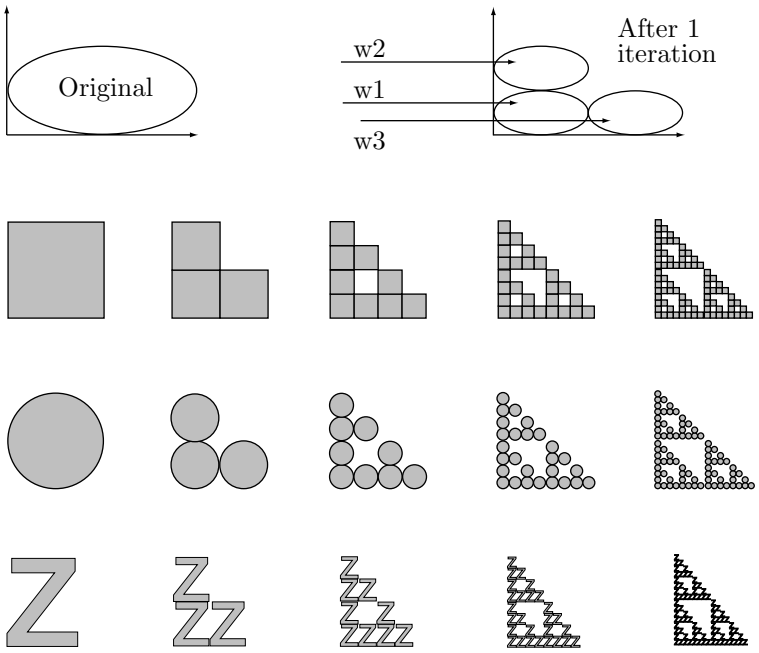


Figure Ans.47: Another Sierpiński Gasket.

5.2: Figure Ans.48 shows $f(t)$ and three shifted copies of the wavelet, for $a = 1$ and $b = 2$, 4, and 6. The inner product $W(a, b)$ is plotted below each copy of the wavelet. It is easy to see how the inner products are affected by the increasing frequency.

The table of Figure Ans.49 lists 15 values of $W(a, b)$, for $a = 1$, 2, and 3 and for $b = 2$ through 6. The density plot of the figure, where the bright parts correspond to large values, shows those values graphically. For each value of a, the CWT yields values that drop with b, reflecting the fact that the frequency of $f(t)$ increases with t. The five values of $W(1, b)$ are small and very similar, while the five values of $W(3, b)$ are larger and differ more. This shows how scaling the wavelet up makes the CWT more sensitive to frequency changes in $f(t)$.

5.3: They are shown in Figure 5.11c.

5.4: Figure Ans.50a shows a simple, 8×8 image with one diagonal line above the main diagonal. Figure Ans.50b,c shows the first two steps in its pyramid decomposition. It is obvious that the transform coefficients in the bottom-right subband (HH) indicate a diagonal artifact located above the main diagonal. It is also easy to see that subband LL is a low-resolution version of the original image.

5.5: The average can easily be calculated. It turns out to be 131.375, which is exactly 1/8 of 1051. The reason the top-left transform coefficient is eight times the

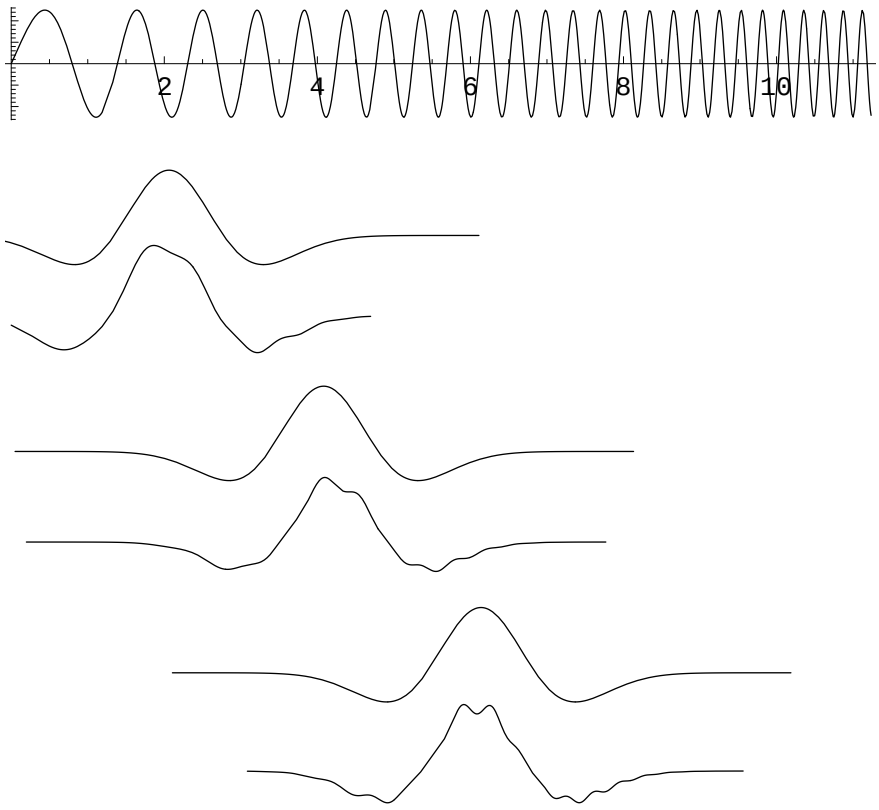


Figure Ans.48: An Inner Product for $a = 1$ and $b = 2, 4, 6$.

a	$b=2$	3	4	5	6
1	0.032512	0.000299	1.10923×10^{-6}	2.73032×10^{-9}	8.33866×10^{-11}
2	0.510418	0.212575	0.0481292	0.00626348	0.00048097
3	0.743313	0.629473	0.380634	0.173591	0.064264

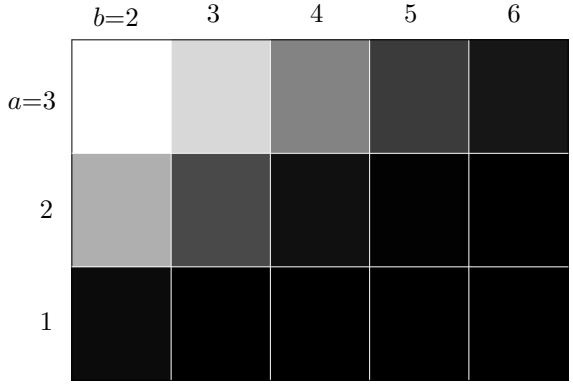


Figure Ans.49: Fifteen Values And a Density Plot of $W(a, b)$.

12	16	12	12	12	12	12	12
12	12	16	12	12	12	12	12
12	12	12	16	12	12	12	12
12	12	12	12	16	12	12	12
12	12	12	12	12	16	12	12
12	12	12	12	12	12	16	12
12	12	12	12	12	12	12	16
12	12	12	12	12	12	12	12

(a)

14	12	12	12	4	0	0	0
12	14	12	12	0	4	0	0
12	14	12	12	0	4	0	0
12	12	14	12	0	0	4	0
12	12	14	12	0	0	4	0
12	12	12	14	0	0	0	4
12	12	12	14	0	0	0	4
12	12	12	12	0	0	0	0

(b)

13	13	12	12	2	2	0	0
12	13	13	12	0	2	2	0
12	12	13	13	0	0	2	2
12	12	12	13	0	0	0	2
2	2	0	0	4	4	0	0
0	2	2	0	0	4	4	0
0	0	2	2	0	0	4	4
0	0	0	2	0	0	0	4

(c)

Figure Ans.50: The Subband Decomposition of a Diagonal Line.

average is that the Matlab code that did the calculations uses $\sqrt{2}$ instead of 2 (see function `individ(n)` in Figure 5.22).

5.6: Figure Ans.51a–c shows the results of reconstruction from 3277, 1639, and 820 coefficients, respectively. Despite the heavy loss of wavelet coefficients, only a very small loss of image quality is noticeable. The number of wavelet coefficients is, of course, the same as the image resolution $128\times128 = 16,384$. Using 820 out of 16,384 coefficients corresponds to discarding 95% of the smallest of the transform coefficients (notice, however, that some of the coefficients were originally zero, so the actual loss may amount to less than 95%).

5.7: The Matlab code of Figure Ans.52 calculates W as the product of the three matrices A_1, A_2, and A_3 and computes the 8×8 matrix of transform coefficients. Notice that the top-left value 131.375 is the average of all the 64 image pixels.

5.8: A simple example of such input is the vector of alternating values $x = (\ldots,1,-1,1,-1,1,\ldots)$.

5.9: For eight-tap filters, rules 1 and 2 imply

$$\begin{gathered}
h_0^2(0)+h_0^2(1)+h_0^2(2)+h_0^2(3)+h_0^2(4)+h_0^2(5)+h_0^2(6)+h_0^2(7)=1,\\
h_0(0)h_0(2)+h_0(1)h_0(3)+h_0(2)h_0(4)+h_0(3)h_0(5)+h_0(4)h_0(6)+h_0(5)h_0(7)=0,\\
h_0(0)h_0(4)+h_0(1)h_0(5)+h_0(2)h_0(6)+h_0(3)h_0(7)=0,\\
h_0(0)h_0(6)+h_0(1)h_0(7)=0,
\end{gathered}$$

and rules 3–5 yield

$$\begin{aligned}
f_0&=\big(h_0(7),h_0(6),h_0(5),h_0(4),h_0(3),h_0(2),h_0(1),h_0(0)\big),\\
h_1&=\big(-h_0(7),h_0(6),-h_0(5),h_0(4),-h_0(3),h_0(2),-h_0(1),h_0(0)\big),\\
f_1&=\big(h_0(0),-h_0(1),h_0(2),-h_0(3),h_0(4),-h_0(5),h_0(6),-h_0(7)\big).
\end{aligned}$$

The eight coefficients are listed in Table 5.35 (this is the Daubechies D8 filter).

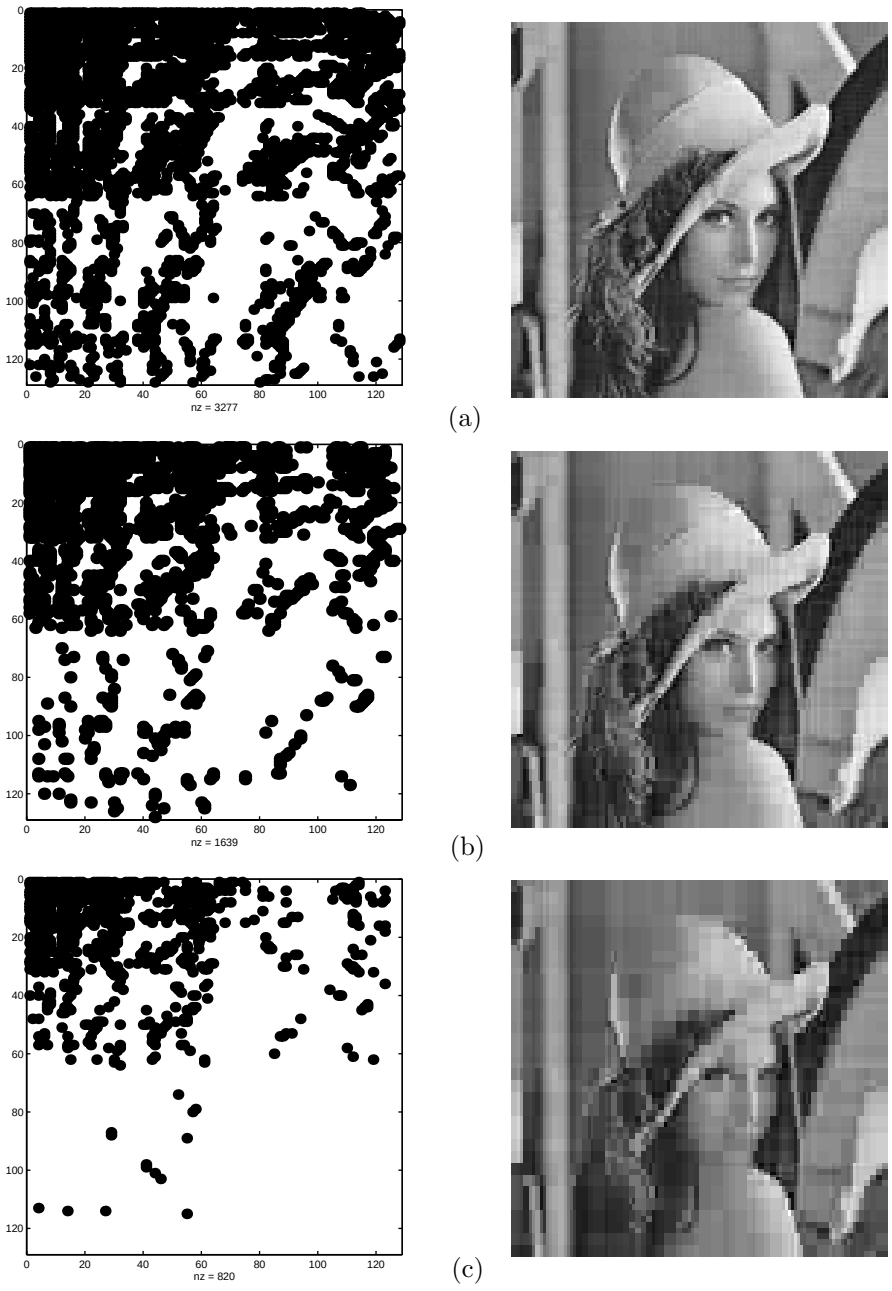


Figure Ans.51: Three Lossy Reconstructions of the 128×128 Lena Image.

```
clear
a1=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0;
 0 0 0 0 1/2 1/2 0 0; 0 0 0 0 0 0 1/2 1/2;
 1/2 -1/2 0 0 0 0 0 0; 0 0 1/2 -1/2 0 0 0 0;
 0 0 0 0 1/2 -1/2 0 0; 0 0 0 0 0 0 1/2 -1/2];
% a1*[255; 224; 192; 159; 127; 95; 63; 32];
a2=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0;
 1/2 -1/2 0 0 0 0 0 0; 0 0 1/2 -1/2 0 0 0 0;
 0 0 0 0 1 0 0 0; 0 0 0 0 0 1 0 0;
 0 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
a3=[1/2 1/2 0 0 0 0 0 0; 1/2 -1/2 0 0 0 0 0 0;
 0 0 1 0 0 0 0 0; 0 0 0 1 0 0 0 0;
 0 0 0 0 1 0 0 0; 0 0 0 0 0 1 0 0;
 0 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
w=a3*a2*a1;
dim=8; fid=fopen('8x8','r');
img=fread(fid,[dim,dim])'; fclose(fid);
w*img*w' % Result of the transform
```

131.375	4.250	−7.875	−0.125	−0.25	−15.5	0	−0.25
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75
12.000	59.875	39.875	31.875	15.75	32.0	16	15.75

Figure Ans.52: Code and Results For the Calculation of Matrix W and Transform $W \cdot I \cdot W^T$.

5.10: Figure Ans.53 lists the Matlab code of the inverse wavelet transform function `iwt1(wc,coarse,filter)` and a test.

5.11: Figure Ans.54 shows the result of blurring the "lena" image. Parts (a) and (b) show the logarithmic multiresolution tree and the subband structure, respectively. Part (c) shows the results of the quantization. The transform coefficients of subbands 5–7 have been divided by two, and all the coefficients of subbands 8–13 have been cleared. We can say that the blurred image of part (d) has been reconstructed from the coefficients of subbands 1–4 (1/64th of the total number of transform coefficients) and half of the coefficients of subbands 5-7 (half of 3/64, or 3/128). On average, the image has been reconstructed from $5/128 \approx 0.039$ or 3.9% of the transform coefficients. Notice that the Daubechies D8 filter was used in the calculations. Readers are encouraged to use this code and experiment with the performance of other filters.

5.12: This is written `a-=b/2;  b+=a;`.

```
function dat=iwt1(wc,coarse,filter)
% Inverse Discrete Wavelet Transform
dat=wc(1:2^coarse);
n=length(wc); j=log2(n);
for i=coarse:j-1
 dat=ILoPass(dat,filter)+ ...
  IHiPass(wc((2^(i)+1):(2^(i+1))),filter);
end

function f=ILoPass(dt,filter)
f=iconv(filter,AltrntZro(dt));

function f=IHiPass(dt,filter)
f=aconv(mirror(filter),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% return filter coefficients with alternating signs
sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% returns a vector of length 2*n with zeros
% placed between consecutive values
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;
```

Figure Ans.53: Code For the One-Dimensional Inverse Discrete Wavelet Transform.

A simple test of `iwt1` is

```
n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt)
rec=iwt1(wc,1,filt)
```

5.13: We sum Equation (5.13) over all the values of l to get

$$\sum_{l=0}^{2^{j-1}-1} s_{j-1,l} = \sum_{l=0}^{2^{j-1}-1} (s_{j,2l} + d_{j-1,l}/2) = \frac{1}{2} \sum_{l=0}^{2^{j-1}-1} (s_{j,2l} + s_{j,2l+1}) = \frac{1}{2} \sum_{l=0}^{2^j-1} s_{j,l}. \tag{Ans.1}$$

Therefore, the average of set s_{j-1} equals

$$\frac{1}{2^{j-1}} \sum_{l=0}^{2^{j-1}-1} s_{j-1,l} = \frac{1}{2^{j-1}} \frac{1}{2} \sum_{l=0}^{2^j-1} s_{j,l} = \frac{1}{2^j} \sum_{l=0}^{2^j-1} s_{j,l}$$

the average of set s_j.

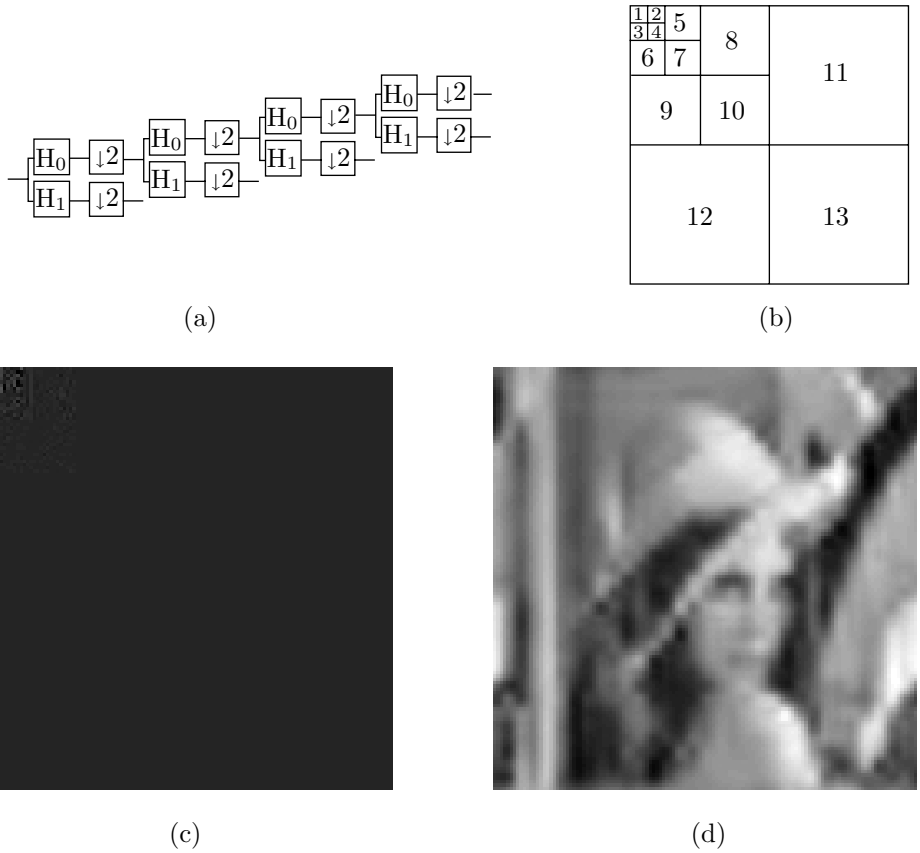


Figure Ans.54: Blurring As A Result of Coarse Quantization.

```
clear, colormap(gray);
filename='lena128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
filt=[0.23037,0.71484,0.63088,-0.02798, ...
 -0.18703,0.03084,0.03288,-0.01059];
fwim=fwt2(img,3,filt);
figure(1), imagesc(fwim), axis square
 fwim(1:16,17:32)=fwim(1:16,17:32)/2;
 fwim(1:16,33:128)=0;
 fwim(17:32,1:32)=fwim(17:32,1:32)/2;
 fwim(17:32,33:128)=0;
 fwim(33:128,:)=0;
figure(2), colormap(gray), imagesc(fwim)
rec=iwt2(fwim,3,filt);
figure(3), colormap(gray), imagesc(rec)
```

Code For Figure Ans.54.

5.14: The code of Figure Ans.55 produces the expression

$$0.0117\mathbf{P}_1 - 0.0977\mathbf{P}_2 + 0.5859\mathbf{P}_3 + 0.5859\mathbf{P}_4 - 0.0977\mathbf{P}_5 + 0.0117\mathbf{P}_6.$$

```
Clear[p,a,b,c,d,e,f];
p[t_]:=a t^5+b t^4+c t^3+d t^2+e t+f;
Solve[{p[0]==p1, p[1/5.]==p2, p[2/5.]==p3,
p[3/5.]==p4, p[4/5.]==p5, p[1]==p6}, {a,b,c,d,e,f}];
sol=ExpandAll[Simplify[%]];
Simplify[p[0.5] /.sol]
```

Figure Ans.55: Code for a Degree-5 Interpolating Polynomial.

5.15: The Matlab code of Figure Ans.56 does that and produces the transformed integer vector $y = (111, -1, 84, 0, 120, 25, 84, 3)$. The inverse transform generates vector z that is identical to the original data x. Notice how the detail coefficients are much smaller than the weighted averages. Notice also that Matlab arrays are indexed from 1, whereas the discussion in the text assumes arrays indexed from 0. This causes the difference in index values in Figure Ans.56.

```
clear;
N=8; k=N/2;
x=[112,97,85,99,114,120,77,80];
% Forward IWT into y
for i=0:k-2,
 y(2*i+2)=x(2*i+2)-floor((x(2*i+1)+x(2*i+3))/2);
end;
y(N)=x(N)-x(N-1);
y(1)=x(1)+floor(y(2)/2);
for i=1:k-1,
 y(2*i+1)=x(2*i+1)+floor((y(2*i)+y(2*i+2))/4);
end;
% Inverse IWT into z
z(1)=y(1)-floor(y(2)/2);
for i=1:k-1,
 z(2*i+1)=y(2*i+1)-floor((y(2*i)+y(2*i+2))/4);
end;
for i=0:k-2,
 z(2*i+2)=y(2*i+2)+floor((z(2*i+1)+x(2*i+3))/2);
end;
z(N)=y(N)+z(N-1);
```

Figure Ans.56: Matlab Code For Forward and Inverse IWT.

5.16: Images g_0 through g_5 will have dimensions

$$(3{\cdot}2^5+1\times 4{\cdot}2^5+1)=97\times 129,\quad 49\times 65,\quad 25\times 33,\quad 13\times 17,\ \text{and}\ 7\times 9.$$

5.17: In the sorting pass of the third iteration the encoder transmits the number $l=3$ (the number of coefficients $c_{i,j}$ in our example that satisfy $2^{12}\le|c_{i,j}|<2^{13}$), followed by the three pairs of coordinates $(3,3)$, $(4,2)$, and $(4,1)$ and by the signs of the three coefficients. In the refinement step it transmits the six bits *cdefgh*. These are the 13th most significant bits of the coefficients transmitted in all the previous iterations.

The information received so far enables the decoder to further improve the 16 approximate coefficients. The first nine become

$$c_{2,3}=s1ac0\ldots0,\ c_{3,4}=s1bd0\ldots0,\ c_{3,2}=s01e00\ldots0,$$
$$c_{4,4}=s01f00\ldots0,\ c_{1,2}=s01g00\ldots0,\ c_{3,1}=s01h00\ldots0,$$
$$c_{3,3}=s0010\ldots0,\ c_{4,2}=s0010\ldots0,\ c_{4,1}=s0010\ldots0,$$

and the remaining seven are not changed.

5.18: The simple equation $10\times 2^{20}\times 8=(500x)\times(500x)\times 8$ is solved to yield $x^2=40$ square inches. If the card is square, it is approximately 6.32 inches on a side. Such a card has 10 rolled impressions (about 1.5×1.5 each), two plain impressions of the thumbs (about 0.875×1.875 each), and simultaneous impressions of both hands (about 3.125×1.875 each). All the dimensions are in inches.

5.19: The bit of 10 is encoded, as usual, in pass 2. The bit of 1 is encoded in pass 1 since this coefficient is still insignificant but has significant neighbors. This bit is 1, so coefficient 1 becomes significant (a fact that is not used later). Also, this bit is the first 1 of this coefficient, so the sign bit of the coefficient is encoded following this bit. The bits of coefficients 3 and -7 are encoded in pass 2 since these coefficients are significant.

6.1: It is easy to calculate that $525\cdot 4/3=700$ pixels.

6.2: The vertical height of the picture on the author's 27 in. television set is 16 in., which translates to a viewing distance of $7.12\times 16=114$ in. or about 9.5 feet. It is easy to see that individual scan lines are visible at any distance shorter than about 6 feet.

6.3: There aren't many, but here are three examples: (1) Surveillance camera, (2) an old, silent movie being restored and converted from film to video, and (3) a video presentation taken underwater.

6.4: The golden ratio $\phi\approx 1.618$ has traditionally been considered the aspect ratio that is most pleasing to the eye. This suggests that 1.77 is the better aspect ratio.

6.5: Imagine a camera panning from left to right. New objects will enter the field of view from the right all the time. A block on the right side of the frame may thus contain objects that did not exist in the previous frame.

6.6: Since $(4,4)$ is at the center of the "+", the value of s is halved, to 2. The next step searches the four blocks labeled 4, centered on $(4,4)$. Assuming that the best match is at $(6,4)$, the two blocks labeled 5 are searched. Assuming that $(6,4)$ is the best match, s is halved to 1, and the eight blocks labeled 6 are searched. The diagram shows that the best match is finally found at location $(7,4)$.

6.7: The picture consists of 18×18 macroblocks, and each macroblock constitutes six 8×8 blocks of samples. The total number of samples is, thus, $18\times18\times6\times64 = 124,416$.

6.8: The size category of zero is 0, so code 100 is emitted, followed by zero bits. The size category of 4 is 3, so code 110 is first emitted, followed by the three least-significant bits of 4, which are 100.

6.9: The zigzag sequence is

$$118, 2, 0, -2, \underbrace{0, \ldots, 0}_{13}, -1, 0, \ldots.$$

The run-level pairs are $(0,2)$, $(1,-2)$, and $(13,-1)$, so the final codes are (notice the sign bits following the run-level codes)

$$0100\,0|000110\,1|00100000\,1|10,$$

(without the vertical bars).

6.10: There are no nonzero coefficients, no run-level codes, just the 2-bit EOB code. However, in nonintra coding, such a block is encoded in a special way.

7.1: An average book may have 60 characters per line, 45 lines per page, and 400 pages. This comes to $60\times45\times400 = 1,080,000$ characters, requiring one byte of storage each.

7.2: The period of a wave is its speed divided by its frequency. For sound we get

$$\frac{34380\,\text{cm/s}}{22000\,\text{Hz}} = 1.562\text{ cm}, \quad \frac{34380}{20} = 1719\text{ cm}.$$

7.3: The (base-10) logarithm of x is a number y such that $10^y = x$. The number 2 is the logarithm of 100 since $10^2 = 100$. Similarly, 0.3 is the logarithm of 2 since $10^{0.3} = 2$. Also, The base-b logarithm of x is a number y such that $b^y = x$ (for any real $b > 1$).

7.4: Each doubling of the sound intensity increases the dB level by 3. Therefore, the difference of 9 dB $(3+3+3)$ between A and B corresponds to three doublings of the sound intensity. Thus, source B is $2{\cdot}2{\cdot}2 = 8$ times louder than source A.

7.5: Each 0 would result in silence and each sample of 1, in the same tone. The result would be a nonuniform buzz. Such sounds were common on early personal computers.

7.6: The experiment should be repeated with several persons, preferably of different ages. The person should be placed in a sound insulated chamber and a pure tone of frequency f should be played. The amplitude of the tone should be gradually increased from zero until the person can just barely hear it. If this happens at a decibel value d, point (d, f) should be plotted. This should be repeated for many frequencies until a graph similar to Figure 7.4a is obtained.

7.7: Imagine that the sound being compressed contains one second of a pure tone (just one frequency). This second will be digitized to 44,100 consecutive samples per channel. The samples indicate amplitudes, so they don't have to be the same. However, after filtering, only one subband (although in practice perhaps two subbands) will have nonzero signals. All the other subbands correspond to different frequencies, so they will have signals that are either zero or very close to zero.

7.8: Assuming that a noise level P_1 translates to x decibels

$$20\log\left(\frac{P_1}{P_2}\right) = x \text{ dB SPL},$$

results in the relation

$$20\log\left(\frac{\sqrt[3]{2}P_1}{P_2}\right) = 20\left[\log_{10}\sqrt[3]{2} + \log\left(\frac{P_1}{P_2}\right)\right] = 20(0.1 + x/20) = x + 2.$$

Thus, increasing the sound level by a factor of $\sqrt[3]{2}$ increases the decibel level by 2 dB SPL.

7.9: The decoder has to decode $44{,}100/384 \approx 114.84$ frames per second. Thus, each frame has to be decoded in approximately 8.7 ms. In order to output 114.84 frames in 64,000 bits, each frame must have $B_f = 557$ bits available to encode it. The number of slots per frame is thus $557/32 \approx 17.41$. Thus, the last (18th) slot is not full and has to padded.

7.10: Table 7.31 shows that the scale factor is 111 and the select information is 2. The third rule in Table 7.32 shows that a scfsi of 2 means that only one scale factor was coded, occupying just six bits in the compressed output. The decoder assigns these six bits as the values of all three scale factors.

7.11: Typical layer II parameters are (1) a sampling rate of 48000 samples/s, (2) a bitrate of 64000 bits/s, and (3) 1152 quantized signals per frame. The decoder has to decode 48000/1152 = 41.66 frames per second. Thus, each frame has to be decoded in 24 ms. In order to output 41.66 frames in 64000 bits, each frame must have $B_f = 1536$ bits available to encode it.

7.12: A program to play `.mp3` files is an MPEG layer III *decoder*, not an encoder. Decoding is much simpler since it does not use a psychoacoustic model, nor does it have to anticipate preechoes and maintain the bit reservoir.

8.1: Because the original string S can be reconstructed from L but not from F.

8.2: A direct application of equation (8.1) eight more times produces:

S[10-1-2]=L[T^2[I]]=L[T[T^1[I]]]=L[T[7]]=L[6]=i;
S[10-1-3]=L[T^3[I]]=L[T[T^2[I]]]=L[T[6]]=L[2]=m;
S[10-1-4]=L[T^4[I]]=L[T[T^3[I]]]=L[T[2]]=L[3]=␣;
S[10-1-5]=L[T^5[I]]=L[T[T^4[I]]]=L[T[3]]=L[0]=s;
S[10-1-6]=L[T^6[I]]=L[T[T^5[I]]]=L[T[0]]=L[4]=s;
S[10-1-7]=L[T^7[I]]=L[T[T^6[I]]]=L[T[4]]=L[5]=i;
S[10-1-8]=L[T^8[I]]=L[T[T^7[I]]]=L[T[5]]=L[1]=w;
S[10-1-9]=L[T^9[I]]=L[T[T^8[I]]]=L[T[1]]=L[9]=s;

The original string "`swiss miss`" is indeed reproduced in S from right to left.

8.3: Figure Ans.57 shows the rotations of S and the sorted matrix. The last column, L of Ans.57b happens to be identical to S, so S=L="`sssssssssh`". Since A=(`s`,`h`), a move-to-front compression of L yields C = (1, 0, 0, 0, 0, 0, 0, 0, 0, 1). Since C contains just the two values 0 and 1, they can serve as their own Huffman codes, so the final result is 1000000001, 1 bit per character!

ssssssssh	hsssssssss
sssssssshs	shssssssss
ssssssshss	sshsssssss
sssssshsss	ssshssssss
ssssshssss	sssshsssss
sssshsssss	ssssshssss
ssshssssss	sssssshsss
sshsssssss	ssssssshss
shssssssss	sssssssshs
hsssssssss	ssssssssh
(a)	(b)

Figure Ans.57: Permutations of "`sssssssssh`".

8.4: The encoder starts at T[0], which contains 5. The first element of L is thus the last symbol of permutation 5. This permutation starts at position 5 of S, so its last element is in position 4. The encoder thus has to go through symbols S[T[i-1]] for $i = 0, \ldots, n-1$, where the notation $i-1$ should be interpreted cyclically (i.e., $0-1$ should be $n-1$). As each symbol S[T[i-1]] is found, it is compressed using move-to-front. The value of I is the position where T contains 0. In our example, T[8]=0, so I=8.

8.5: The first element of a triplet is the distance between two dictionary entries, the one best matching the content and the one best matching the context. In this case there is no content match, no distance, so any number could serve as the first element, 0 being the best (smallest) choice.

8.6: Because the three lines are sorted in ascending order. The bottom two lines of Table 8.13c are not in sorted order. This is why the "`zz...z`" part of string S must be preceded and followed by complementary bits.

8.7: The encoder places S between two entries of the sorted associative list and writes the (encoded) index of the entry above or below S on the compressed stream. The fewer the number of entries, the smaller this index, and the better the compression.

8.8: Context 5 is compared to the three remaining contexts 6, 7, and 8, and it is most similar to context 6 (they share a suffix of "b"). Context 6 is compared to 7 and 8 and, since they don't share any suffix, context 7, the shorter of the two, is selected. The remaining context 8 is, of course, the last one in the ranking. The final context ranking is

$$1 \to 3 \to 4 \to 0 \to 5 \to 6 \to 7 \to 8.$$

8.9: Equation (8.3) shows that the third "a" is assigned rank 1 and the "b" and "a" following it are assigned ranks 2 and 3, respectively.

8.10: Table Ans.58 shows the sorted contexts. Equation (Ans.2) shows the context ranking at each step.

$$\begin{array}{llll} 0\ , & 0\to 2\ , & 1\to 3\to 0\ , & \\ u & u\quad b & l\quad b\quad u & \\ 0\to 2\to 3\to 4\ , & 2\to 4\to 1\to 5\to 0\ , & & \\ u\quad l\quad a\quad b & l\quad a\quad d\quad b\quad u & & \\ 3\to 5\to 2\to 6\to 5\to 0\ . & & & \\ i\quad a\quad l\quad b\quad d\quad u & & & \end{array} \tag{Ans.2}$$

The final output is "u 2 b 3 l 4 a 5 d 6 i 6." Notice that each of the distinct input symbols appears once in this output in raw format.

																			0		λ	u
															0		λ	u	1		ubla	d
									0		λ	u			1		ubla	d	2		ub	l
					0		λ	u	1		ubla	x			2		ub	l	3		ublad	i
		0		λ	u		1	ub	l		2	ub	l		3		ublad	x	4		ubladi	x
0	λ	u		1	ub	x		2	ubl	x		3	ubl	a		4	ubl	a	5		ubl	a
1	u	x		2	u	b		3	u	b		4	u	b		5	u	b	6		u	b
	(a)			(b)				(c)				(d)				(e)				(f)			

Table Ans.58: Constructing the Sorted Lists For `ubladiu`.

8.11: All n_1 bits of string L_1 need be written on the output stream. This already shows that there is going to be no compression. String L_2 consists of n_1/k 1's, so all of it has to be written on the output stream. String L_3 similarly consists of n_1/k^2 1's, and so on. The size of the output stream is thus

$$n_1 + \frac{n_1}{k} + \frac{n_1}{k^2} + \frac{n_1}{k^3} + \cdots + \frac{n_1}{k^m} = n_1 \frac{k^{m+1}-1}{k^m(k-1)},$$

for some value of m. The limit of this expression, when $m \to \infty$, is $n_1 k/(k-1)$. For $k = 2$ this equals $2n_1$. For larger values of k this limit is always between n_1 and $2n_1$.

For the curious reader, here is how the sum above is calculated. Given the series

$$S = \sum_{i=0}^{m} \frac{1}{k^i} = 1 + \frac{1}{k} + \frac{1}{k^2} + \frac{1}{k^3} + \cdots + \frac{1}{k^{m-1}} + \frac{1}{k^m},$$

we multiply both sides by $1/k$

$$\frac{S}{k} = \frac{1}{k} + \frac{1}{k^2} + \frac{1}{k^3} + \cdots + \frac{1}{k^m} + \frac{1}{k^{m+1}} = S + \frac{1}{k^{m+1}} - 1,$$

and subtract

$$\frac{S}{k}(k-1) = \frac{k^{m+1}-1}{k^{m+1}} \to S = \frac{k^{m+1}-1}{k^m(k-1)}.$$

8.12: The input stream consists of:
1. A run of three zero groups, coded as 10|1 since 3 is in second position in class 2.
2. The nonzero group 0100, coded as 111100.
3. Another run of three zero groups, again coded as 10|1.
4. The nonzero group 1000, coded as 01100.
5. A run of four zero groups, coded as 010|00 since 4 is in first position in class 3.
6. 0010, coded as 111110.
7. A run of two zero groups, coded as 10|0.

The output is thus the 31-bit string 1011111001010110001000111110100.

8.13: The input stream consists of:
1. A run of three zero groups, coded as R_2R_1 or 101|11.
2. The nonzero group 0100, coded as 00100.
3. Another run of three zero groups, again coded as 101|11.
4. The nonzero group 1000, coded as 01000.
5. A run of four zero groups, coded as $R_4 = 1001$.
6. 0010, coded as 00010.
7. A run of two zero groups, coded as $R_2 = 101$.

The output is thus the 32-bit string 10111001001011101000100100010101.

8.14: The input stream consists of:
1. A run of three zero groups, coded as F_3 or 1001.
2. The nonzero group 0100, coded as 00100.
3. Another run of three zero groups, again coded as 1001.
4. The nonzero group 1000, coded as 01000.
5. A run of four zero groups, coded as $F_3F_1 = 1001|11$.
6. 0010, coded as 00010.
7. A run of two zero groups, coded as $F_2 = 101$.

The output is thus the 32-bit string 10010010010010100010011100010101.

8.15: Yes, if they are located in different quadrants or subquadrants. Pixels 123 and 301, for example, are adjacent in Figure 8.27 but have different prefixes.

8.16: No, since all prefixes have the same probability of occurrence. In our example the prefixes are four bits long and all 16 possible prefixes have the same probability since a pixel may be located anywhere in the image. A Huffman code calculated for 16 equally-probable symbols has an average size of four bits per symbol, so nothing would be gained. The same is true for suffixes.

8.17: This is possible, but it places severe limitations on the size of the string. In order to rearrange a one-dimensional string into a four-dimensional cube, the string size should be 2^{4n}. If the string size happens to be $2^{4n} + 1$, it has to be extended to $2^{4(n+1)}$, which doubles its size. It is possible to rearrange the string into a rectangular box, not just a cube, but then its size will have to be of the form $2^{n1}2^{n2}2^{n3}2^{n4}$ where the four ni's are integers.

8.18: The LZW method, which starts with the entire alphabet stored at the beginning of its dictionary. However, an adaptive version of LZW can be designed to compress words instead of individual characters.

8.19: Relative values (or *offsets*). Each (x, y) pair may specify the position of a character relative to its predecessor. This results in smaller numbers for the coordinates, and smaller numbers are easier to compress.

8.20: There may be such letters in other, "exotic" alphabets, but a more common example is a rectangular box enclosing text. The four rules comprising such a box should be considered a mark, but the text characters inside the box should be identified as separate marks.

8.21: Because this guarantees that the two probabilities will add up to 1.

8.22: Figure Ans.59 shows how state A feeds into the new state D' which, in turn, feeds into states E and F. Notice how states B and C haven't changed. Since the new state D' is identical to D, it is possible to feed A into either D or D' (cloning can be done in two different but identical ways). The original counts of state D should now be divided between D and D' in proportion to the counts of the transitions $A \to D$ and $B, C \to D$.

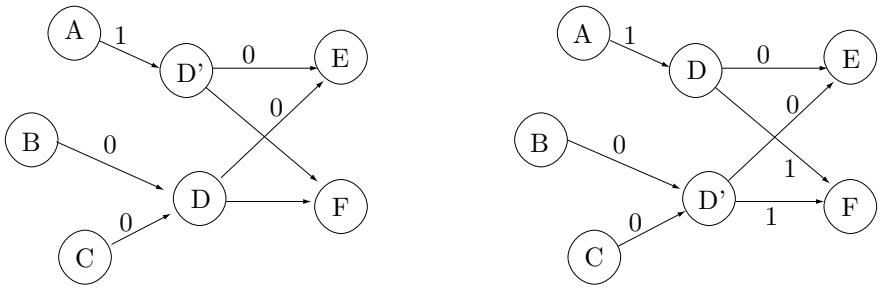


Figure Ans.59: New State D' Cloned.

8.23: Figure Ans.60 shows the new state 6. Its 1-output is identical to that of state 1, and its 0-output is a copy of the 0-output of state 3.

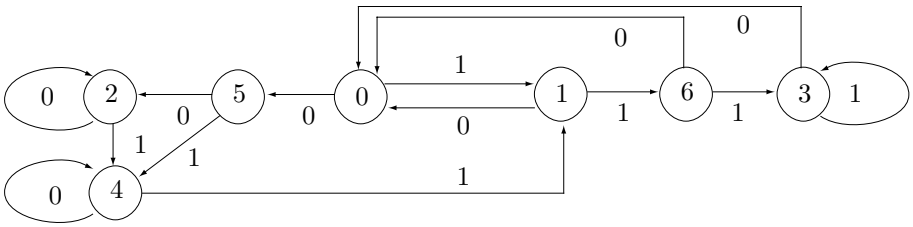


Figure Ans.60: State 6 Added.

8.24: A precise answer requires many experiments with various data files. A little thinking, though, shows that the larger k, the better the initial model that is created when the old one is discarded. Larger values of k thus minimize the loss of compression. However, very large values may produce an initial model that is already large and cannot grow much. The best value for k is therefore one that produces an initial model large enough to provide information about recent correlations in the data, but small enough so it has room to grow before it too has to be discarded.

8.25: The number of marked points can be written $8(1+2+3+5+8+13) = 256$ and the numbers in parentheses are the Fibonacci numbers.

8.26: It is very small. A segment pointing in direction D_i can be preceded by another segment pointing in the same direction only if the original curve is straight or very close to straight for more than 26 coordinate units (half the width of grid S_{13}).

8.27: A point has two coordinates. If each coordinate occupies 8 bits, then the use of Fibonacci numbers reduces the 16-bit coordinates to an 8-bit number, a compression ratio of 0.5. The use of Huffman codes can typically reduce this 8-bit number to (on average) a 4-bit code, and the use of the Markov model can perhaps cut this by another bit. The result is an estimated compression ratio of $3/16 = 0.1875$. If each coordinate is a 16-bit number, then this ratio improves to $3/32 = .09375$.

8.28: The resulting, shorter grammar is shown in Figure Ans.61. It is one rule and one symbol shorter.

Input	Grammar
S → abcdbcabcdbc	S → CC
	A → bc
	C → aAdA

Figure Ans.61: Improving the Grammar of Figure 8.42.

8.29: Generating rule C has made rule B underused (i.e., used just once).

8.30: Rule S consists of two copies of rule A. The first time rule A is encountered, its contents aBdB are sent. This involves sending rule B twice. The first time rule B is sent, its contents bc are sent (and the decoder does not know that the string bc it is receiving is the contents of a rule). The second time rule B is sent, the pair $(1, 2)$ is sent (offset 1, count 2). The decoder identifies the pair and uses it to set up the rule 1 → bc. Sending the first copy of rule A therefore amounts to sending abcd$(1, 2)$. The second copy of rule A is sent as the pair $(0, 4)$ since A starts at offset 0 in S and its length is 4. The decoder identifies this pair and uses it to set up the rule 2 → a1d1. The final result is therefore abcd$(1, 2)(0, 4)$.

8.31: In each of these cases, the encoder removes one edge from the boundary and inserts two new edges. There is a net gain of one edge.

8.32: They create triangles $(18, 2, 3)$ and $(18, 3, 4)$, and reduce the boundary to the sequence of vertices

$$(4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18).$$

A.1: It is a combination of "4x and "9, or "49.

B.1: The values of X are listed below. Each has a probability of 1/8.

$$\begin{aligned} X(HHH) = 3, \quad X(HHT) = 2, \quad X(HTH) = 2, \quad X(HTT) = 1, \\ X(THH) = 2, \quad X(THT) = 1, \quad X(TTH) = 1, \quad X(TTT) = 0. \end{aligned}$$

B.2: The definition of expectation implies that $E(X) = v$. The expected value of a constant random variable is the constant value.

B.3: Denote $m = E(X)$. From $\mathrm{Var}(X) = E[(X - m)^2]$ we get

$$\begin{aligned} \mathrm{Var}(X) &= E(X^2 - 2Xm + m^2) \\ &= E(X^2) - E(-2mX) + E(m^2) \\ &= E(X^2) - 2mE(X) + m^2 \\ &= E(X^2) - 2m^2 + m^2 \\ &= E(X^2) - m^2. \end{aligned}$$

B.4: The probability of rolling a double-six with two dice is 1/36. The complement probability is 35/36. The probability of rolling a double-six in the first throw, or the second throw,..., or the 24th throw is

$$1 - \underbrace{(35/36)(35/36)\cdots(35/36)}_{24} \approx 1 - 0.5086 = 0.4914.$$

For game B, the probability of rolling a six is 1/6, its complement is 5/6, so the probability of rolling a six in four tries is

$$1 - (5/6)(5/6)(5/6)(5/6) \approx 1 - 0.482 = 0.518;$$

slightly higher than the winning probability of game A.

B.5: This problem is easy to solve intuitively. Once B has withdrawn, one of the remaining two companies will win the contract. All we have to do to find their new chances is to scale their old chances such that they add up to 1. Since the old chances add up to 3/5, they have to be scaled by 5/3 to bring their new sum to 1. The new chances are therefore $(2/5)(5/3) = 2/3$ and $(1/5)(5/3) = 1/3$.

Next, we use conditional probabilities to solve the same problem. We are looking for the conditional probabilities $P(A|\bar{B})$ and $P(C|\bar{B})$. We know that $P(B)$ was 2/5, so $P(\bar{B}) = 1 - P(B) = 3/5$. The quantity $P(A \cdot \bar{B})$ is the probability that A will win *and* B will not win. It equals $P(A)$, since if A wins, B cannot win. Equation (B.2) therefore yields the conditional probabilities

$$\begin{aligned} P(A|\bar{B}) &= \frac{P(A \cdot \bar{B})}{P(\bar{B})} = \frac{(2/5)}{(3/5)} = \frac{2}{3}, \\ P(C|\bar{B}) &= \frac{P(C \cdot \bar{B})}{P(\bar{B})} = \frac{(1/5)}{(3/5)} = \frac{1}{3}. \end{aligned}$$

B.6: Applying Bayes' theorem, the results are 0.24, 0.24, and 0.36, respectively. The knowledge that the student got an A made is less likely that he selected mathematics, more likely that he selected biology, and did not much affect the probabilities that he selected physics or chemistry.

B.7: Using appropriate mathematical software it is easy to obtain this integral separately for negative and nonnegative values of x.

$$\int L(V,x)\,dx = \begin{cases} \frac{-1}{V\exp\left(\sqrt{\frac{2}{V}}x\right)}, & x \geq 0, \\ \frac{1}{\sqrt{2V}}\exp\left(\sqrt{\frac{2}{V}}x\right), & x < 0. \end{cases}$$

C.1: A straight line segment from a to b is an example of a one-dimensional curve that passes through every point in the interval a, b.

C.2: The key is to realize that P_0 is a single point, and P_1 is constructed by connecting nine copies of P_0 with straight segments. Similarly, P_2 consists of nine copies of P_1, in different orientations, connected by segments (the dashed segments in Figure Ans.62).

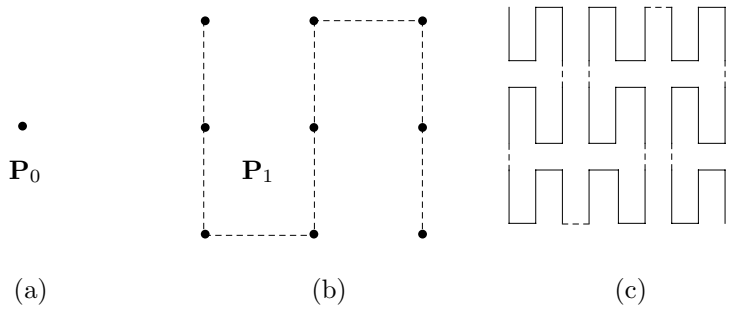


Figure Ans.62: The First Three Iterations of the Peano Curve.

C.3: Written in binary, the coordinates are $(1101, 0110)$. We iterate four times, each time taking 1 bit from the x coordinate and 1 bit from the y coordinate to form an (x, y) pair. The pairs are 10, 11, 01, 10. The first one yields [from Table C.12(1)] 01. The second pair yields [also from Table C.12(1)] 10. The third pair [from Table C.12(1)] 11, and the last pair [from Table C.12(4)] 01. The result is thus $01|10|11|01 = 109$.

C.4: Table C.2 shows that this traversal is based on the sequence 2114.

C.5: This is straightforward

$$\begin{aligned}(00,01,11,10) &\rightarrow (000,001,011,010)(100,101,111,110)\\ &\rightarrow (000,001,011,010)(110,111,101,100)\\ &\rightarrow (000,001,011,010,110,111,101,100).\end{aligned}$$

D.1: It is $j \cdot m + (i+1)$.

D.2: It makes sense to store the degree n of the polynomial in the first array location.

D.3: Yes. In a ternary tree, the three children of node a are stored in locations $2a$, $2a+1$, and $2a+2$ and the parent of a can be found at array location $\lfloor a/3 \rfloor$.

D.4: The pre-order, in-order, and level-order traversals are, respectively

$$\begin{gathered}A,((B,(D,E)),(C,(F,(G,H)))),\\ ((D,B,E),A,(null,C,(G,F,H))),\\ (A,(B,C),(D,E,F),(G,H)).\end{gathered}$$

D.5: Each of the 8 characters of a name can be one of the 26 letters or the ten digits, so the total number of names is $36^8 = 2,821,109,907,456$; close to 3 trillion.

E.1: A direct check shows that when only a single bit is changed in any of the codewords of code_2, the result is not any of the other codewords.

E.2: A direct check shows that code_4 has a Hamming distance of 4, which is more than enough to detect all 2-bit errors.

E.3: b_2 is the parity of b_3, b_6, b_7, b_{10}, and b_{11}. b_4 is the parity of b_5, b_6, and b_7. b_8 is the parity of b_9, b_{10}, and b_{11}.

E.4: Table Ans.63 summarizes the definitions of the five parity bits required for this case.

Parity	Data bits															
bits	3	5	6	7	9	10	11	12	13	14	15	17	18	19	20	21
1	x	x		x	x		x		x		x	x		x		x
2	x		x	x		x	x			x	x		x	x		
4		x	x	x				x	x	x	x				x	x
8					x	x	x	x	x	x	x					
16												x	x	x	x	x

Table Ans.63: Hamming Code for $m = 16$.

E.5: It is a variable-size code (Chapter 2). It's easy to see that it satisfies the prefix property.

F.1: Yes. It alternates between states 1, 2, 3, 4, and 1.

H.1: All the shades of gray.

H.2: A yellow surface absorbs blue and reflects green and red.

H.3: Yes! These are three colors that produce white when mixed. Examples are red, green, and blue; cyan, magenta, and yellow.

H.4: For point 1, it is easy to see that $R = 0$ and $G = 1$. Since it is on the $U = 0$ plane, where $Y = B$, it is easy to calculate that $B = 0.663$. For point 2 we have $R = 0$ and $B = 1$. Since it is on the $Y = 0.3$ plane, we get $G = 0.317$. Similarly, point 3 has $R = 0$ and the two equations $Y = 0.3$ and $U = 0$ are solved to yield $G = 0.453$ and $B = 0.3$.

H.5: Because white isn't a pure color; it is a mixture of all colors.

H.6: Recall that the sum of a dyad is white. Since illuminant white is in the middle of the line connecting c and d, it is obtained by adding equal amounts of them $(0.5c + 0.5d)$. This is why they are complementary.

H.7: Saturation refers to the amount of white in a color. Point f corresponds to full saturation, whereas illuminant white corresponds to no saturation. The saturation of the color of point e is, therefore, the ratio of the distances fw/ew.

H.8: If we continue the line from w to g, it intercepts the pure spectral curve at the bottom, an area that does not correspond to any wavelength. We therefore continue the line in the opposite direction until it intercepts the pure spectral curve at h and we say that the dominant wavelength of point g is 497_c (where c stands for "complement").

H.9: Direct calculations using matrix D_{44} produce the areas in Figure Ans.64. The three areas have black pixel percentages of 1/16, 2/16, and 16/16, respectively.

$$A[x,y] = \begin{array}{ccc} 0 & 1 & 15 \\ 10001000\ldots & 10001000\ldots & 11111111\ldots \\ 00000000\ldots & 00000000\ldots & 11111111\ldots \\ 00000000\ldots & 00100010\ldots & 11111111\ldots \\ 00000000\ldots & 00000000\ldots & 11111111\ldots \end{array}$$

Figure Ans.64: Ordered Dither: Three Uniform Areas.

H.10: A direct application of Equation (H.2) yields

$$D_{88} = \begin{array}{|cccccccc|}\hline 0 & 32 & 8 & 40 & 2 & 34 & 10 & 42 \\ 48 & 16 & 56 & 24 & 50 & 18 & 58 & 26 \\ 12 & 44 & 4 & 36 & 14 & 46 & 6 & 38 \\ 60 & 28 & 52 & 20 & 62 & 30 & 54 & 22 \\ 3 & 35 & 11 & 43 & 1 & 33 & 9 & 41 \\ 51 & 19 & 59 & 27 & 49 & 17 & 57 & 25 \\ 15 & 47 & 7 & 39 & 13 & 45 & 5 & 37 \\ 63 & 31 & 55 & 23 & 61 & 29 & 53 & 21 \\ \hline \end{array}.$$

H.11: A checkerboard pattern. This can be seen by manually simulating the algorithm of Figure H.22b for a few pixels.

H.12: We assume that the test is

if $p \geq 0.5$, then $p := 1$ else $p := 0$; add the error $0.5 - p$ to the next pixel q.

The first pixel is thus set to 1 and the error of $0.5 - 1 = -0.5$ is added to the second pixel, changing it from 0.5 to 0. The second pixel is set to 0 and the error, which is $0 - 0 = 0$, is added to the third pixel, leaving it at 0.5. The third pixel is thus set to 1 and the error of $0.5 - 1 = -0.5$ is added to the fourth pixel, changing it from 0.5 to 0. The results are

$$\boxed{.5}\boxed{.5}\boxed{.5}\boxed{.5}\boxed{.5} \rightarrow \boxed{1}\boxed{0}\boxed{.5}\boxed{.5}\boxed{.5} \rightarrow \boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{.5} \rightarrow \boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}$$

H.13: Direct examination shows that the barons are 62 and 63 and the near-barons are 60 and 61.

H.14: A checkerboard pattern, similar to the one produced by diffusion dither. This can be seen by manually executing the algorithm of Figure H.24 for a few pixels.

H.15: Classes 14, 15, and 10 are barons. Classes 12 and 13 are near-barons. The class numbers in positions (i, j) and $(i, j + 2)$ add up to 15.

I.1: This is straightforward

$$\mathbf{A}+\mathbf{B} = \begin{pmatrix} 8 & 10 & 12 \\ 8 & 10 & 12 \\ 8 & 10 & 12 \end{pmatrix}, \mathbf{A}-\mathbf{B} = \begin{pmatrix} -6 & -6 & -6 \\ 0 & 0 & 0 \\ 6 & 6 & 6 \end{pmatrix}, \mathbf{A}\times\mathbf{B} = \begin{pmatrix} 18 & 24 & 30 \\ 54 & 69 & 84 \\ 90 & 114 & 138 \end{pmatrix}.$$

I.2: Equation (I.4) gives

$$\mathbf{T}^{-1} = \frac{1}{1 \cdot 1 - 1 \cdot 1}\Big(\cdots\Big),$$

which is undefined. Matrix $\mathbf{T}$ is thus *singular*; it does not have an inverse! This becomes easy to understand when we think of $\mathbf{T}$ as the coefficients matrix of the

system of equations (I.3) above. This is a system of three equations in the three unknowns x, y, and z, but its first two equations are contradictory. The first one says that $x - y$ equals 1, while the second one says that the same $x - y$ equals -2. Mathematically, such a system has a singular coefficients matrix.

I.3: This is easily proved by showing that both dot products $(\mathbf{P} \times \mathbf{Q}) \bullet \mathbf{P}$ and $(\mathbf{P} \times \mathbf{Q}) \bullet \mathbf{Q}$ equal zero.

$$(\mathbf{P} \times \mathbf{Q}) \bullet \mathbf{P} = P_1(P_2Q_3 - P_3Q_2) + P_2(-P_1Q_3 + P_3Q_1) + P_3(P_1Q_2 - P_2Q_1) = 0.$$

And similarly for $(\mathbf{P} \times \mathbf{Q}) \bullet \mathbf{Q}$.

I.4: In the special case where $\mathbf{i} = (1, 0, 0)$ and $\mathbf{j} = (0, 1, 0)$ it is easy to verify that the product $\mathbf{i} \times \mathbf{j}$ equals $(0, 0, 1) = \mathbf{k}$. The triplet $(\mathbf{i}, \mathbf{j}, \mathbf{i} \times \mathbf{j} = \mathbf{k})$ thus has the handedness of the coordinate system (it is either right-handed or left-handed, depending on the coordinate system). In a right-handed coordinate system, the right hand rule makes it easy to predict the direction of $\mathbf{P} \times \mathbf{Q}$. The rule is: if your thumb points in the direction of $\mathbf{P}$ and your second finger, in the direction of $\mathbf{Q}$, then your middle finger will point in the direction of $\mathbf{P} \times \mathbf{Q}$. In a left-handed coordinate system, a similar left-hand rule applies.

I.5: They either have the same direction, or they point in opposite directions.

I.6: We are looking for a vector $\mathbf{P}(t)$ that is linear in t and that satisfies $\mathbf{P}(0) = \mathbf{P}_1$ and $\mathbf{P}(1) = \mathbf{P}_2$. It is easy to guess that

$$\mathbf{P}(t) = (1 - t)\mathbf{P}_1 + t\mathbf{P}_2 = t(\mathbf{P}_2 - \mathbf{P}_1) + \mathbf{P}_1$$

satisfies both conditions.

I.7: This is not especially hard

$$\begin{aligned}\mathbf{c} &= \frac{2 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1)}{1^2 + 0^2 + (-1)^2}(1, 0, -1) = (-1/2, 0, 1/2),\\ \mathbf{d} &= \mathbf{a} - \mathbf{c} = (2.5, 1, 2.5).\end{aligned}$$

I.8: Because of the wide spread use of computers, the world of science and engineering has moved from analog to digital. Instead of a continuous function $f(t)$ we now have an n-tuple $(f_1, f_2, \ldots, f_n)$, and this is a vector (in n dimensions).

I.9: The multiplication rule yields $(0, 1) \times (0, 1) = (-1, 0)$ or $i \times i = -1$. The strange rule of complex number multiplication results in $i^2 = -1$ or $\sqrt{-1} = i$.

I.10: The multiplication rule yields $(a, b) \times (-a, -b) = (0, 0)$, which justifies calling $(-a, -b)$ the opposite of (a, b).

I.11: We start with $i = \cos(\pi/2) + i\sin(\pi/2)$, from which we get

$$\sqrt{i} = \left(\cos\frac{\pi}{2} + i\sin\frac{\pi}{2}\right)^{1/2}.$$

By DeMoivre's theorem, this equals

$$\cos\frac{\pi}{4} + i\sin\frac{\pi}{4} = \frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}} = \frac{1+i}{\sqrt{2}}. \qquad \text{(Ans.3)}$$

A simple check gives

$$\left(\frac{1+i}{\sqrt{2}}\right)^2 = \frac{1+2i+i^2}{2} = i.$$

(DeMoivre's theorem states that $\sin(nx) + i\cos(nx)$ is one of the values of $(\sin x + i\cos x)^n$.)

I.12: We start with the elegant formula

$$e^{it} = \cos t + i\sin t.$$

Substituting $t = \pi/2$ yields

$$e^{i\pi/2} = i\sin(\pi/2) = i.$$

Both sides are now raised to the ith power, yielding

$$i^i = (e^{i\pi/2})^i = e^{i^2\pi/2} = e^{-\pi/2}.$$

Surprisingly, this is a real number. In the past, calculating many digits of this number (or others like it) involved years of toil. Today, however, the single Matlab statement `vpa('i^i',140)` produces, in less than a second, the 140-digit number

0.20787 95763 50761 90854 69556 19834 97877 00338 77841
63176 96080 75135 88305 54198 77285 48213 97886 00277
86542 60353 40521 77330 72350 21808 19061 97303 74663
98700

Since $a = \exp(\ln a)$ for any a, we also get

$$e^{\ln i} = i = e^{i\pi/2},$$

from which it is clear that $\ln i = i\pi/2$.

By the way, the exponential function e^z is defined for any complex number $z = x + iy$ by

$$e^z = 1 + \frac{z}{1!} + \frac{z^2}{2!} + \frac{z^3}{3!} + \cdots.$$

Leonhard Euler, the great Eighteenth century mathematician, introduced the notation i for $\sqrt{-1}$ in 1777. It is interesting to note that electrical engineers use the notation j instead, since they deal with electrical voltages and currents and find it convenient to reserve i to indicate current.

I.13: The axiom and the production rules stay the same. Only the initial heading and the turn angle change. The initial heading can be either 0 or 90°, and the turn angle either 90° or 270°.

I.14: Such a polynomial depends on three coefficients **b**, **c**, and **d** that can be considered three-dimensional points, and any three points are on the same plane.

I.15:

$$\begin{aligned}\mathbf{P}(2/3) =& (0,-9)(2/3)^3 + (-4.5,13.5)(2/3)^2 + (4.5,-3.5)(2/3)\\ =& (0,-8/3) + (-2,6) + (3,-7/3)\\ =& (1,1) = \mathbf{P}_3\end{aligned}$$

I.16: We use the relations $\sin 30° = \cos 60° = .5$ and the approximation $\cos 30° = \sin 60° \approx .866$. The four points are $\mathbf{P}_1 = (1,0)$, $\mathbf{P}_2 = (\cos 30°, \sin 30°) = (.866,.5)$, $\mathbf{P}_3 = (.5,.866)$, and $\mathbf{P}_4 = (0,1)$. The relation $\mathbf{A} = \mathbf{N}\cdot\mathbf{P}$ becomes

$$\begin{pmatrix}\mathbf{a}\\ \mathbf{b}\\ \mathbf{c}\\ \mathbf{d}\end{pmatrix} = \mathbf{A} = \mathbf{N}\cdot\mathbf{P} = \begin{pmatrix}-4.5 & 13.5 & -13.5 & 4.5\\ 9.0 & -22.5 & 18 & -4.5\\ -5.5 & 9.0 & -4.5 & 1.0\\ 1.0 & 0 & 0 & 0\end{pmatrix}\begin{pmatrix}(1,0)\\ (.866,.5)\\ (.5,.866)\\ (0,1)\end{pmatrix}$$

and the solutions are

$$\begin{aligned}\mathbf{a} &= -4.5(1,0) + 13.5(.866,.5) - 13.5(.5,.866) + 4.5(0,1) = (.441,-.441),\\ \mathbf{b} &= 19(1,0) - 22.5(.866,.5) + 18(.5,.866) - 4.5(0,1) = (-1.485,-0.162),\\ \mathbf{c} &= -5.5(1,0) + 9(.866,.5) - 4.5(.5,.866) + 1(0,1) = (0.044,1.603),\\ \mathbf{d} &= 1(1,0) - 0(.866,.5) + 0(.5,.866) - 0(0,1) = (1,0).\end{aligned}$$

The PC is thus $\mathbf{P}(t) = (.441,-.441)t^3 + (-1.485,-0.162)t^2 + (0.044,1.603)t + (1,0)$. The midpoint is $\mathbf{P}(.5) = (.7058,.7058)$, only 0.2% away from the midpoint of the arc, which is at $(\cos 45°, \sin 45°) \approx (.7071,.7071)$.

I.17: The new equations are easy enough to set up. Using *Mathematica*, they are also easy to solve. The following code

```
Solve[{d==p1,
a al^3+b al^2+c al+d==p2,
a be^3+b be^2+c be+d==p3,
a+b+c+d==p4},{a,b,c,d}];
```

```
ExpandAll[Simplify[%]]
```

(where `al` and `be` stand for α and β, respectively) produces the (messy) solutions

$$\begin{aligned}
\mathbf{a} &= -\frac{\mathbf{P}_1}{\alpha\beta} + \frac{\mathbf{P}_2}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta} + \frac{\mathbf{P}_3}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} + \frac{\mathbf{P}_4}{1-\alpha-\beta+\alpha\beta}\\
\mathbf{b} &= \mathbf{P}_1\left(-\alpha+\alpha^3+\beta-\alpha^3\beta-\beta^3+\alpha\beta^3\right)/\gamma + \mathbf{P}_2\left(-\beta+\beta^3\right)/\gamma\\
&\quad + \mathbf{P}_3\left(\alpha-\alpha^3\right)/\gamma + \mathbf{P}_4\left(\alpha^3\beta-\alpha\beta^3\right)/\gamma\\
\mathbf{c} &= -\mathbf{P}_1\left(1+\frac{1}{\alpha}+\frac{1}{\beta}\right) + \frac{\beta\mathbf{P}_2}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta}\\
&\quad + \frac{\alpha\mathbf{P}_3}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} + \frac{\alpha\beta\mathbf{P}_4}{1-\alpha-\beta+\alpha\beta}\\
\mathbf{d} &= \mathbf{P}_1.\\
&\text{where}\quad \gamma = (-1+\alpha)\alpha(-1+\beta)\beta(-\alpha+\beta).
\end{aligned}$$

From here, the basis matrix immediately follows

$$\begin{pmatrix}
-\frac{1}{\alpha\beta} & \frac{1}{-\alpha^2+\alpha^3\alpha\beta-\alpha^2\beta} & \frac{1}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} & \frac{1}{1-\alpha-\beta+\alpha\beta}\\
\frac{-\alpha+\alpha^3+\beta-\alpha^3\beta-\beta^3+\alpha\beta^3}{\gamma} & \frac{-\beta+\beta^3}{\gamma} & \frac{\alpha-\alpha^3}{\gamma} & \frac{\alpha^3\beta-\alpha\beta^3}{\gamma}\\
-\left(1+\frac{1}{\alpha}+\frac{1}{\beta}\right) & \frac{\beta}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta} & \frac{\alpha}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} & \frac{\alpha\beta}{1-\alpha-\beta+\alpha\beta}\\
1 & 0 & 0 & 0
\end{pmatrix}$$

A direct check, again using *Mathematica*, for $\alpha = 1/3$ and $\beta = 2/3$, reduces this matrix to matrix $\mathbf{N}$ of Equation (I.22).

I.18: The missing points will have to be estimated by interpolation or extrapolation from the known points before our method can be applied. Obviously, the fewer points are known, the worse the final interpolation. Note that 16 points are necessary, since a bicubic polynomial has 16 coefficients.

I.19: Figure Ans.65a shows a diamond-shaped grid of 16 equally-spaced points. The eight points with negative weights are shown in black. Figure Ans.65b shows a cut (labeled xx) through four points in this surface. The cut is a curve that passes through pour data points. It is easy to see that when the two exterior (black) points are raised, the center of the curve (and, as a result, the center of the surface) gets lowered. It is now clear that points with negative weights push the center of the surface in a direction opposite that of the points.

Figure Ans.65c is a more detailed example that also shows why the four corner points should have positive weights. It shows a simple symmetric surface patch that interpolates the 16 points

$$\begin{aligned}
&\mathbf{P}_{00} = (0,0,0), \quad \mathbf{P}_{10} = (1,0,1), \quad \mathbf{P}_{20} = (2,0,1), \quad \mathbf{P}_{30} = (3,0,0),\\
&\mathbf{P}_{01} = (0,1,1), \quad \mathbf{P}_{11} = (1,1,2), \quad \mathbf{P}_{21} = (2,1,2), \quad \mathbf{P}_{31} = (3,1,1),
\end{aligned}$$

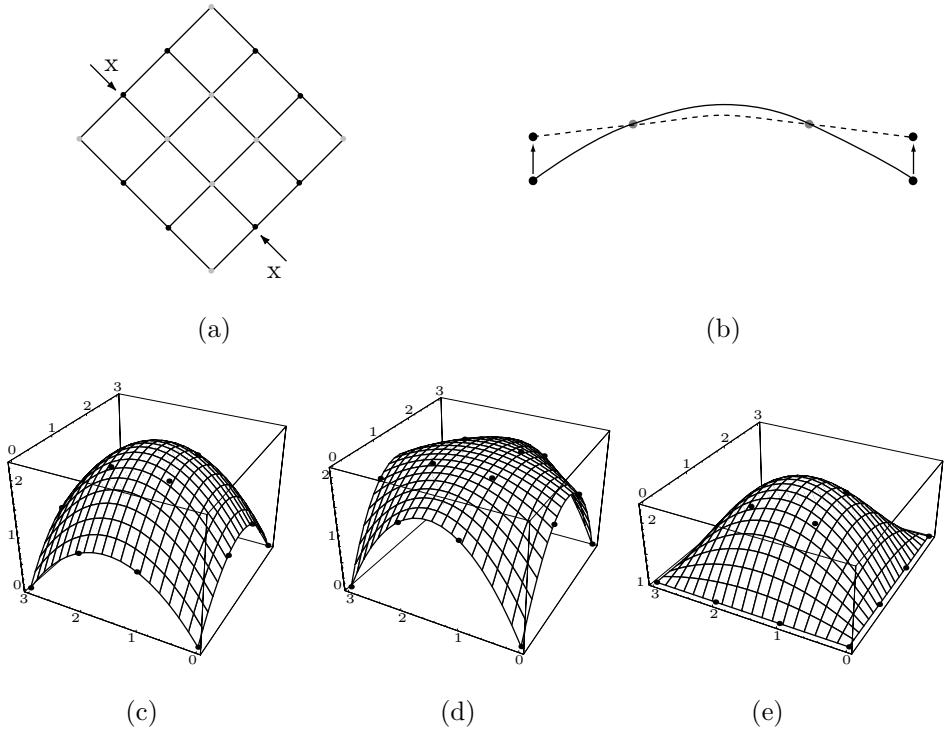


Figure Ans.65: An Interpolating Bicubic Surface Patch.

$$\mathbf{P}_{02} = (0,2,1), \quad \mathbf{P}_{12} = (1,2,2), \quad \mathbf{P}_{22} = (2,2,2), \quad \mathbf{P}_{32} = (3,2,1),$$
$$\mathbf{P}_{03} = (0,3,0), \quad \mathbf{P}_{13} = (1,3,1), \quad \mathbf{P}_{23} = (2,3,1), \quad \mathbf{P}_{33} = (3,3,0).$$

We first raise the eight boundary points from $z = 1$ to $z = 1.5$. Figure Ans.65d shows how the center point $\mathbf{P}(.5, .5)$ gets lowered from $(1.5, 1.5, 2.25)$ to $(1.5, 1.5, 2.10938)$. We next return those points to their original positions and instead raise the four corner points from $z = 0$ to $z = 1$. Figure Ans.65e shows how this raises the center point from $(1.5, 1.5, 2.25)$ to $(1.5, 1.5, 2.26563)$.

the talk of the ordinary Englishman made me sick, I couldn't get enough exercise, and the amusements of London seemed as flat as soda-water that has been standing in the sun.

John Buchan, 1915, *The Thirty-nine Steps*

```
Clear[Nh,p,pnts,U,W];
p00={0,0,0}; p10={1,0,1}; p20={2,0,1}; p30={3,0,0};
p01={0,1,1}; p11={1,1,2}; p21={2,1,2}; p31={3,1,1};
p02={0,2,1}; p12={1,2,2}; p22={2,2,2}; p32={3,2,1};
p03={0,3,0}; p13={1,3,1}; p23={2,3,1}; p33={3,3,0};
Nh={{-4.5,13.5,-13.5,4.5},{9,-22.5,18,-4.5},
 {-5.5,9,-4.5,1},{1,0,0,0}};
pnts={{p33,p32,p31,p30},{p23,p22,p21,p20},
 {p13,p12,p11,p10},{p03,p02,p01,p00}};
U[u_]:={u^3,u^2,u,1};   W[w_]:={w^3,w^2,w,1};
(* prt [i] extracts component i from the 3rd dimen of P *)
prt[i_]:=pnts[[Range[1,4],Range[1,4],i]];
p[u_,w_]:={U[u].Nh.prt[1].Transpose[Nh].W[w],
 U[u].Nh.prt[2].Transpose[Nh].W[w], \
 U[u].Nh.prt[3].Transpose[Nh].W[w]};
 g1=ParametricPlot3D[p[u,w], {u,0,1},{w,0,1},
 Compiled->False, DisplayFunction->Identity];
g2=Graphics3D[{AbsolutePointSize[2],
 Table[Point[pnts[[i,j]]],{i,1,4},{j,1,4}]}];
Show[g1,g2, ViewPoint->{-2.576, -1.365, 1.718}]
```

Code For Figure Ans.65.

Index

a transparent thing becomes invisible if it is put in any medium of almost the same refractive index

—H. G. Wells, *The Invisible Man* (1898)