

Chapter 2

Digital Formats

Recent developments in computer science disciplines have caused an ever-increasing spread of digital technologies, which has consequently turned upside down nearly all human activities in many of their aspects. At the same time, the fragmentation of the computer science landscape and the tough competition among software producers have led to a proliferation of different, and unfortunately often incompatible, ways of representing documents in digital format. Only very recently, initial efforts have been started aimed at rationalization and standardization of such formats, although it must be said that some (limited) range of variability is needed and desirable to be able to properly represent documents having very different characteristics (for instance, it is hardly conceivable that different kinds of content as, say, text, images and music will be ever represented using the same ‘language’).

In the following, some of the most widespread formats will be introduced, grouped according to the kind and degree of structure they allow expressing in a document content representation. Raster image formats will be dealt with more in-depth, for showing and comparing different perspectives on how visual information can be encoded. Also PS and PDF will be given somewhat larger room because they are famous from an end-usage perspective, but are not very well-known as to internal representation. Also for comparison purposes to the raster formats, a particular attention will be given to their image representation techniques. Markup languages will be discussed with reference to HTML and XML. However, the aim here will not be providing an HTML/XML programmer’s manual (many valid books on this specific subject are available); rather, their representational rationale and approach will be stressed. HTML will be discussed with just the aim of considering a tag-based format and the possibilities it provides, while XML will be presented to give an idea of how flexible information representation can be obtained by a general tag definition mechanism supported by external processing techniques.

2.1 Compression Techniques

The availability of digital documents and the consequent need to store huge quantities thereof and to transmit them over computer networks have raised the need for reducing the amount of memory required to represent a document. In the former case, this would allow keeping together on one support large repositories of related documents; in the latter, this would enable quick delivery of documents to their final users. Such problems relate particularly, but not exclusively, to digital image representation, due to the large amount of data to be stored. Thus, a primary interest in this field has always been the development of effective and efficient techniques for *data compression*. *Lossless* compression techniques ensure that the original uncompressed data can be exactly restored, at the cost of a lower compression ratio. Conversely, *lossy* techniques allow exchanging a reasonable amount of quality in image reconstruction for significant improvement in compression performance. With this subject being pervasive in digital document representation, it will be discussed preliminarily in the presentation of the specific digital formats, by considering outstanding solutions that have been proposed and widely adopted so far.

RLE (Run Length Encoding) The *RLE* algorithm performs a lossless compression of input data based on sequences of identical values (called *runs*). It is a historical technique, originally exploited by fax machines and later adopted in image processing. Indeed, since in black&white images only two symbols are present, the chances to have long runs for a value, and hence the possibility of high compression rates, are significantly increased. The algorithm is quite easy: each run, instead of being represented explicitly, is translated by the encoding algorithm in a pair (l, v) where l is the length of the run and v is the value of the run elements. Of course, the longer the runs in the sequence to be compressed, the better the compression ratio.

Example 2.1 (RLE compression of a sample sequence) The sequence

xyzzxyzywxxy

would be represented by RLE as

(1, x) (1, y) (2, z) (1, x) (2, y) (1, z) (1, x) (1, y) (1, w) (1, x) (1, y).

Assuming that an integer takes a byte just as a character, here the ‘compressed’ version of the original 13-byte sequence takes 22 bytes. This is due to the large cardinality of the alphabet with respect to the length of the sequence, and to the high variability of the symbols therein.

Huffman Encoding The compression strategy devised by *Huffman* [20] relies on a greedy algorithm to generate a dictionary, i.e., a symbol–code table, that allows obtaining a nearly *optimal* data compression from an information-theoretic view-

point.¹ First of all, the alphabet is identified as the set of symbols appearing in the string to be encoded. Then, the frequency in such a string of each symbol of the alphabet is determined. Subsequently, a binary tree, whose nodes are associated to frequencies, is built as follows:

1. Set the symbols of the alphabet, with the corresponding frequencies, as leaf nodes
2. **while** the structure is not a tree (i.e., there is no single root yet)
 - (a) Among the nodes that do not still have a parent, select two whose associated frequency is minimum
 - (b) Insert a new node that becomes the parent of the two selected nodes and gets as associated frequency the sum of the frequencies of such nodes
3. In the resulting tree, mark each left branch as '1' and each right branch as '0'

Lastly, the binary code of each symbol is obtained as the sequence of branch labels in the path from the root to the corresponding leaf. Now, each symbol in the source sequence is replaced by the corresponding binary code. A consequence of this frequency-based procedure is that shorter codes are assigned to frequent symbols, and longer ones to rare symbols.

Decompression is performed using the symbol–code table, scanning the encoded binary string and emitting a symbol as soon as its code is recognized:

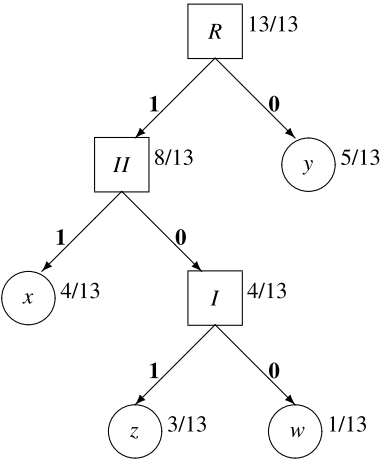
1. **while** the encoded sequence has not finished
 - (a) Set the current code to be empty
 - (b) **while** the current code does not correspond to any code in the table
 - (i) Add the next binary digit in the encoded sequence to the current code
 - (c) Output to the decompressed sequence the symbol corresponding to the code just recognized

Note that the code generation procedure is such that no code in the table is a prefix of another code, which ensures that as soon as a code is recognized left-to-right it is the correct one: neither a longer code can be recognized, nor recognition can stop before recognizing a code or having recognized a wrong code. Hence, the compressed sequence can be transmitted as a single and continuous binary sequence, without the need for explicit separators among codes.

Example 2.2 (Huffman compression of a sample sequence) Assume the message to be encoded is **xyzzxyzywx**. A node is created for each different symbol in the sequence: x , y , z and w . The symbol frequencies are $\frac{4}{13}$ for x , $\frac{5}{13}$ for y , $\frac{3}{13}$ for z and $\frac{1}{13}$ for w . The two nodes with minimum frequencies (z and w) are connected to a new parent node (let us call it I), that gets the sum of their frequencies ($\frac{4}{13}$). Now the two nodes without a parent and with minimum frequencies are x and I , that are connected to a new parent II , whose overall frequency is $\frac{8}{13}$. Lastly, the only two nodes without a parent, II and y , are connected to a new parent R that completes the tree and becomes its root. From the resulting tree (depicted in Fig. 2.1) the

¹As proved by Shannon [23], the number of bits required to specify sequences of length N , for large N , is equal to $N \cdot H$ (where H is the source entropy).

Fig. 2.1 Binary tree that defines the Huffman codes for the symbols in string `xyzzxyzywxwxy`



following table is obtained:

Symbol	x	y	z	w
Code	11	0	101	100

that is used to encode the initial string:

x	y	z	z	x	y	y	z	x	y	w	x	y
11	0	101	101	11	0	0	101	11	0	100	11	0

Given the compressed string, and scanning it from left to right: 1 is not a code, 11 is recognized as the code of `x`; 0 is immediately recognized as the code of `y`; 1 is not a code, nor is 10, while 101 is the code of `z`; and so on.

LZ77 and LZ78 (Lempel–Ziv) A series of compression techniques was developed by Lempel and Ziv, and hence denoted by the acronym LZ followed by the year of release: *LZ77* [28] dates back to 1977, while *LZ78* [29] dates to 1978. Both are dictionary-based encoders, and represent a basis for several other techniques (among which LZW, to be presented next). The former looks backward to find duplicated data, and must start processing the input from its beginning. The latter scans forward the buffer, and allows random access to the input but requires the entire dictionary to be available. They have been proved to be equivalent to each other when the entire data is decompressed. In the following, we will focus on the former.

LZ77 adopts a representation based on the following elements:

- Literal** a single character in the source stream;
- Length–Distance** pair to be interpreted as “the next *length* characters are equal to the sequence of characters that precedes them of *distance* positions in the source stream”.

The repetitions are searched in a limited buffer consisting of the last N kB (where $N \in \{2, 4, 32\}$), whence it is called a *sliding window* technique. Implementations may adopt different strategies to distinguish *length–distance* pairs from *literals* and to output the encoded data. The original version exploits triples

$$(\text{length}, \text{distance}, \text{literal}),$$

where

- *length–distance* refers to the longest match found in the buffer, and
- *literal* is the character following the match

(*length* = 0 if two consecutive characters can only be encoded as literals).

LZW (Lempel–Ziv–Welch) The information lossless compression technique by *Lempel–Ziv–Welch* (whence the acronym *LZW*) [26] leverages the redundancy of characters in a message and increases compression rate with the decreasing of the number of symbols used (e.g., colors in raster images). It uses fixed-length codes to encode symbol sequences of variable length that appear frequently, and inserts them in a symbol–code conversion dictionary D (that is not stored, since it can be reconstructed during the decoding phase). If implemented as an array, the index of elements in D corresponds to their code. In the following, T will denote the source stream, composed on an alphabet A made up of n symbols, and C the compressed one.

To start compression, one needs to know how many bits are available to represent each code (m bits denote 2^m available codes, ranging in $[0, 2^m - 1]$). The symbols of A are initially inserted in the first n positions (from 0 to $n - 1$) of D . The compression algorithm continues exploiting an output string C and two variables (a prefix string P , initially empty, and a symbol s):

1. P is initially empty
2. **while** T is not finished
 - (a) $s \leftarrow$ next symbol in T
 - (b) **if** string Ps is already present in D
 - (i) P becomes Ps
 - (c) **else**
 - (i) Insert string Ps in the first available position of D
 - (ii) Append to C the code of P found in D
 - (iii) P becomes s
3. Append to C the code of P found in D

After compression, D can be forgotten, since it is not needed for decompression. Due to the many accesses to D , it is usefully implemented using a hashing technique, based on the string as a key. Of course, there is the risk that all available codes in the table are exploited, and special steps have to be taken to handle the case in which an additional one is needed.

The only information needed during the decoding phase is the alphabet A , whose symbols are initially placed in the first n positions of dictionary D . Encoding is such

that all codes in the coded stream can be translated into a string. Decoding continues as follows, using two temporary variables c and o for the current and old codes, respectively:

1. $c \leftarrow$ first code from C
2. Append to T the string for c found in D
3. **while** C is not finished
 - (a) $o \leftarrow c$
 - (b) $c \leftarrow$ next code in C
 - (c) **if** c is already present in D
 - (i) Append to T the string found in D for c
 - (ii) $P \leftarrow$ translation for o
 - (iii) $s \leftarrow$ first symbol of translation for c
 - (iv) Add Ps to D
 - (d) **else**
 - (i) $P \leftarrow$ translation for o
 - (ii) $s \leftarrow$ first character of P
 - (iii) Append Ps to T and add it to D

Note that both coding and decoding generate the same character-code table. More space can be saved as follows. Notice that all insertions in D are in the form Ps , where P is already in D . Thus, instead of explicitly reporting Ps in the new entry, it is possible to insert (c, s) , where c is the code for P (that usually has a shorter bit representation than P itself).

Example 2.3 (LZW compression of a sample sequence) Consider the following string to be compressed:

x y z z x y y z x y w x y												
↓												
String	w	x	y	z	xy	yz	zz	zx	xyy	yzx	xyw	wx
Compact	w	x	y	z	1+y	2+z	3+z	3+x	4+y	5+x	4+w	0+x
Code ₁₀	0	1	2	3	4	5	6	7	8	9	10	11
Code ₂	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011
↑												
1 2 3 3 4 5 4 0 4												
0001 0010 0011 0011 0100 0101 0100 0000 0100												

In this case, the string to be compressed is 26 bits long (13 symbols \times 2 bits needed to encode each symbol in an alphabet of 4) and the compressed string is 36 bits long (9 codes \times 4 bits needed to represent each). This happens because the message is too short and varied, which generates many codes but does not provide the opportunity to reuse them.

For very long data streams and with a large alphabet (e.g., 256 symbols) the compression estimate is about 40% if applied to text.

DEFLATE *DEFLATE* [16] is a compression algorithm that joins LZ77 and Huffman. Compressed streams consist of sequences of blocks, each preceded by a 3-bit header, where the first bit is a flag saying whether that is the last block (**0** = false, **1** = true) and the other two express the encoding method for the block (the most used is **10**, corresponding to *dynamic Huffman encoding*, while **11** is reserved).

It consists of two steps:

1. Apply matching to find duplicate strings and replace them with backward references, obtaining the pairs

$(length, distance)$

where $length \in [3, 258]$ and $distance \in [1, 32768]$.

2. Replace the original symbols with new symbols having length inversely proportional to their frequency of use, according to Huffman encoding. Specifically, literal and length alphabets are merged into a single alphabet 0–285, and encoded in a tree that provides room for 288 symbols, as follows:

- 0–255 represent the possible literals;
- 256 denotes the end of the block;
- 257–285, combined with extra bits, express a match length of 3 to 258 bytes, where codes in each group from $256 + 4i + 1$ to $256 + 5i$, for $i = 0, \dots, 5$, denote 2^i lengths using i extra bits, and code 285 denotes length 258 (an extensional representation is provided in Table 2.1);
- 286–287 are reserved (not used).

Distance values between 1 and 32768 are encoded using 32 symbols as follows:

- Symbols 0 to 3 directly represent distances 1 to 4.
- Symbols $(2 \cdot i + 4)$ and $(2 \cdot i + 5)$, for $i = 0, \dots, 12$, denote distances from $(2^{i+2} + 1)$ to (2^{i+3}) as follows:

$(2 \cdot i + 4)$ denotes the base value $2^{i+2} + 1$ (and distances up to $3 \cdot 2^{i+1}$),

$(2 \cdot i + 5)$ denotes the base value $3 \cdot 2^{i+1} + 1$ (and distances up to 2^{i+3}),

and the actual distance is obtained by adding to the base value a displacement $d \in [0, 2^{i+1} - 1]$, expressed by $(i + 1)$ extra bits.

- Symbols 30 and 31 are reserved (not used).

These symbols are arranged in a distance tree, as well.

2.2 Non-structured Formats

By the attribute “non-structured”, in this section, we will refer to documents that do not explicitly organize the information they contain into structures that are significant from a geometrical and/or conceptual point of view.

Table 2.1 Correspondence between codes and lengths in the DEFLATE compression technique

Code	Extra bits	Length	Lengths per code
257		3	
⋮	0	⋮	1
264		10	
265		11–12	
⋮	1	⋮	2
268		17–18	
269		19–22	
⋮	2	⋮	4
272		31–34	
273		35–42	
⋮	3	⋮	8
276		59–66	
277		67–82	
⋮	4	⋮	16
280		115–130	
281		131–162	
⋮	5	⋮	32
284		227–257	
285	0	258	1

2.2.1 Plain Text

Plain text, usually denoted by the *TXT* extension, refers to a stream of characters represented according to some standard coding agreement. In particular, each character is represented as an integer (typically in binary or hexadecimal format²), to which the users assign a specific meaning (that can be identified with the symbol denoting that character). Usually, the characters to be represented are the letters of a particular writing system or *script* (the alphabet of a language, possibly extended with diacritic marks) plus punctuation marks and other useful symbols. In this format, the only semblance of a structure is given by the line separator, roughly indicating a kind of ‘break’ in the character flow.

This format is mainly intended for information exchange, not typography. Thus, each character represents the ‘essence’ of a symbol, abstracted from the many par-

²In the rest of this section, both notations will be used interchangeably, as needed.

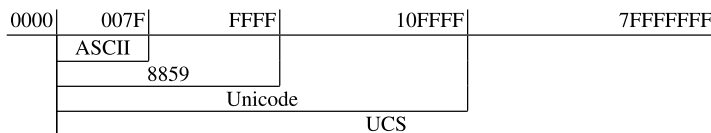


Fig. 2.2 Ranges of hexadecimal codes exploited by various standards for character encoding

ticular shapes by which that symbol can be graphically denoted. Rather, it should ensure complete accessibility, independent of the particular platform (software, operating system and hardware architecture) in use, as long as there is an agreement on the coding standard to be used. As a consequence, there is a need to define shared conventions on which basis a given value is uniformly interpreted by all users as the same character. Figure 2.2 depicts the ranges of hexadecimal codes exploited by various standards that will be discussed in the following.

ASCII Historically, the coding agreement that has established in the computer community, becoming a *de facto* standard, is the *ASCII* (American Standard Code for Information Interchange).³ It is based on the exploitation of 1 byte per character, leading to a total of 256 possible configurations (i.e., symbols that can be represented). In its original version, devised to support American texts (US-ASCII), it actually exploited only 7 bits, reserving the last bit for error checking purposes in data transmission, or leaving it unused. This allowed $2^7 = 128$ different configurations, to represent 95 printable characters plus some control codes. However, the worldwide spread of computer systems has subsequently called for an extension of this range to include further useful symbols as well. In particular, several kinds of codes were developed with the advent of the Internet, and for this reason their definition and assignment are ruled by the IANA (Internet Assigned Numbers Authority).

ISO Latin Many languages other than English often rely on scripts containing characters (accented letters, quotation marks, etc.) that are not provided for in ASCII. A first, straightforward solution was found in exploiting for coding purposes also the last bit of a character byte, which allowed 128 additional configurations/characters available. However, even the 256 total configurations obtained in this way were soon recognized to be insufficient to cover all Western writing systems. To accommodate this, several alternative exploitations of the new configurations coming from the use of the eighth bit were developed, leading to a family of standards known as ISO/IEC 8859. Specifically, 16 different extensions were defined, and labeled as the ISO/IEC 8859-*n* standards (with $n = 1, \dots, 16$ denoting

³Another coding standard in use during the ancient times of computing machinery, abandoned later on, was the *EBCDIC* (Extended Binary Coded Decimal Interchange Code), which started from the *BCD* (Binary Coded Decimal) binary representation of decimal digits, from $0000_2 = 0$ to $1001_2 = 9$, used by early computers for performing arithmetical operations, and extended it by putting before four additional bits. The decimal digits are characterized by an initial 1111_2 sequence, while the other configurations available allow defining various kinds of characters (alphabetic ones, punctuation marks, etc.).

the specific extension). Among them, the most used are the sets of Latin characters (*ISO Latin*), and specifically the ISO-8859-1 code. Obviously, the alternative sets of codes are incompatible with each other: the same (extended) configuration corresponds to different characters in different standards of the family. It should be noted that only printable characters are specified by such codes, leaving the remaining configurations unspecified and free for use as control characters.

UNICODE Although the one-byte-per-character setting is desirable for many reasons (efficiency in terms of memory, easy mapping, computer architecture compliance, etc.), it is not sufficient to effectively cover the whole set of languages and writing systems in the world. In order to collect in a unified set the different codes in the ISO/IEC 8859 family, and to allow the inclusion of still more scripts, avoiding incompatibility problems when switching from one to another, the *Unicode* [24] and *UCS* (Universal Character Set, also known as ISO/IEC 10646) [6] standards were developed, and later converged towards joint development. Here, each character is given a unique name and is represented as an abstract integer (called *code point*), usually referenced as ‘U+’ followed by its hexadecimal value.

Unicode defines a codespace of 1,114,112 potential code points in the range 000000_{16} – $10FFFFFF_{16}$, logically divided into 17 *planes* (00_{16} – 10_{16}), made up of $2^{16} = 65,536$ (0000_{16} – $FFFF_{16}$) code points each:

Plane 0 (0000–FFFF) *Basic Multilingual Plane (BMP)*

Plane 1 (10000–1FFFF) *Supplementary Multilingual Plane (SMP)*

Plane 2 (20000–2FFFF) *Supplementary Ideographic Plane (SIP)*

Planes 3 to 13 (30000–DFFFF) *unassigned*

Plane 14 (E0000–EFFFF) *Supplementary Special-purpose Plane (SSP)*

Planes 15 (F0000–FFFFF) **and 16** (100000–10FFFF) *Private Use Area (PUA)* – reserved

Only 100,713 code points are currently exploited (as of Version 5.1, April 2008, covering 75 scripts). Ranges of code points have been reserved for every current and ancient writing systems, already known or still to be discovered. The BMP, in particular, is devoted to support the unification of prior character sets as well as characters for writing systems in current use, and contains most of the character assignments so far. Code points in the BMP are denoted using just four digits (‘U+nnnn’); for code points outside the BMP, five or six digits are used, as required. ISO/IEC 8859 characters all belong to the BMP, and hence are mapped on their Unicode/UCS counterparts via a U+nnnn notation.

UTF Code points are implemented using a technique called *UTF*. Actually, different kinds of UTF exist, some using configurations of bits of fixed length and others using variable-length encoding:

UTF-8 sequences of one to six 8-bit code values

UTF-16 sequences of one or two 16-bit code values

UTF-32 or **UTF-4** fixed-length 4-byte code values

UTF-2 fixed-length 2-byte code values (now replaced by UTF-8)

UTF-8 is the only platform-independent UTF. Conversely, UTF-16 and UTF-32 are platform-dependent concerning byte ordering, and are also incompatible with ASCII, which means that Unicode-aware programs are needed to handle them, even in cases when the file contains only ASCII characters.⁴ For these reasons, 8-bit encodings (ASCII, ISO-8859-1, or UTF-8) are usually exploited for representing text even on platforms that are natively based on other formats.

Table 2.2 shows the 4-byte fixed-length and UTF-8 variable length representations of remarkable code points, while Table 2.3 reports a comparison of UTF-8 and UTF-16 for interesting ranges of code points.

UTF-8 adopts a segment-based management: a subset of frequent characters is represented using fewer bits, while special bit sequences in shorter configurations are used to indicate that the character representation takes more bits. Although this adds redundancy to the coded text, advantages outperform disadvantages (and, in any case, compression is not an aim of Unicode). For instance, such a variable-length solution allows saving memory in all the many cases in which the basic Latin script, without accented letters and typographic punctuation, is sufficient for the user's purposes (e.g., programming). It exploits up to four bytes to encode Unicode values, ranging 0–10FFFF; for the ISO/IEC 10646 standard, it can exploit even five or six bytes, to allow encoding values up to U+7FFFFFFF. The rules for obtaining UTF-8 codes are as follows:

1. *Single-byte codes start with 0 as the most significant bit.* This leaves 7 bits available for representing the actual character.
2. *In multi-byte codes, the number of consecutive 1's in the most significant bits of the first byte, before a 0 bit is met, denotes the number of bytes that make-up the multi-byte code; subsequent bytes of the multi-byte code start with 10 in the two most significant bits.* For a multi-byte code made up of n bytes, this leaves $(7 - n) + 6 \cdot (n - 1)$ bits available for representing the actual character.

This ensures that no sequence of bytes corresponding to a character is ever contained in a longer sequence representing another character, and allows performing string matching in a text file using a byte-wise comparison (which is a significant help and allows for less complex algorithms). Additionally, if one or more bytes are lost because of transmission errors, decoding can still be synchronized again on the next character, this way limiting data loss.

UTF-8 is compliant to ISO/IEC 8859-1 and fully backward compatible to ASCII (and, additionally, non-ASCII UTF-8 characters are just ignored by legacy ASCII-based programs). Indeed, due to rule 1, UTF-8 represents values 0–127 (00_{16} – $7F_{16}$) using a single byte with the leftmost bit at 0, which is exactly the same representation as in ASCII (and hence an ASCII file and its UTF-8 counterpart are identical). As to ISO/IEC 8859-1, since it fully exploits 8 bits, it goes from 0 to 255 (00_{16} – FF_{16}). Its lower 128 characters are just as ASCII, and fall, as said, in the 1-byte

⁴As a trivial example of how tricky UTF-16 can be: usual C string handling cannot be applied because it would consider as string terminators the many 00000000 byte configurations in UTF-16 codes.

Table 2.3 UTF-8 and UTF-16 encodings for noteworthy ranges of code points. 0/1s denote fixed bit values, while x's denote bit positions available for the actual value representation. For short, in UTF-8, the notation $(B) \times N$ stands for N bytes of the kind B

Range (hex)	UTF-16	UTF-8
000000–00007F	00000000 0xxxxxxx	0xxxxxxx
000080–0007FF	00000xxx xxxxxxxx	110xxxxx + 10xxxxxx
000800–00FFFF	xxxxxxx xxxxxxxx	1110xxxx + (10xxxxxx) \times 2
010000–10FFFF	110110xx xxxxxxxx 110111xx xxxxxxxx	11110xxx + (10xxxxxx) \times 3
00110000–001FFFFF		11110xxx + (10xxxxxx) \times 3
00200000–003FFFFF		111110xx + (10xxxxxx) \times 4
04000000–7FFFFFFF		1111110x + (10xxxxxx) \times 5

representation, while its upper 128 characters (those extending ASCII, going from 128 to 255, i.e., 80_{16} – FF_{16}) fall in the 2-byte representation (080_{16} – $7FF_{16}$, or 128–2047), thus they will be represented as 110000xx 10xxxxxx.

The rules for obtaining UTF-16 codes are as follows:

- For code points in the BMP (0000_{16} – $FFFF_{16}$), 16 bits (i.e., 2 bytes) are sufficient.
- To represent code-points over the BMP (010000_{16} – $10FFFF_{16}$), *surrogate pairs* (pairs of 16-bit words, each called a *surrogate*, to be considered as a single entity) are exploited. The first and second surrogate are denoted, respectively, by the bit sequences 110110 and 110111 in the first six positions, which avoids ambiguities between them and leaves $16 - 6 = 10$ available bits each for the actual value representation. First 10000_{16} is subtracted from the code point value, in order to obtain a value ranging in 00000_{16} – $FFFFF_{16}$, that can be represented by 20 bits. Such resulting 20 bits are split into two subsequences made up of 10 bits each, assigned to the trailing 10 bits of the first and second surrogate, respectively.

This means that the first surrogate will take on values in the range $D800_{16}$ – $DBFF_{16}$, and the second surrogate will take on values in the range $DC00_{16}$ – $DFFF_{16}$.

Example 2.4 (UTF-16 representation of a sample code point) The character at code point U+10FF3A is transformed by the subtraction into

$$FFFA_{16} = 11111111111100111010_2$$

The first ten bits (1111111111) are placed in the first surrogate, that becomes $110110111111111_2 = DBFF_{16}$, and the second ten bits 1100111010 are placed in the second surrogate, that becomes $110111100111010_2 = DF3A_{16}$; overall, the original code point is represented as the sequence (DBFF DF3A)₁₆.

Of course, not to confuse a surrogate with a single 16-bit character, Unicode (and thus ISO/IEC 10646) are bound not to assign characters to any of the code points in the U+D800–U+DFFF range.

2.2.2 Images

Image components play a very important role in many kinds of documents because they leverage on the human perception capabilities to compactly and immediately represent large amounts of information that even long textual descriptions could not satisfactorily express. While text is a type of linear and discrete information, visual information, however, is inherently multi-dimensional (2D still images, that will be the area of interest of this book, are characterized along three dimensions: two refer to space and one to color) and continuous. Thus, while the transposition of the former into a computer representation is straightforward, being both characterized by the same features, the translation of the latter posed severe problems, and raised the need for formats that could find a good trade-off among the amount of information to be preserved, the memory space demand and the manipulation processes to be supported. To make the whole thing even more difficult, all these requirements may have different relevance depending on the aims for which the image is produced.

Color Spaces

A *color space* is a combination of a *color model*, i.e., a mathematical abstract model that allows representing colors in numeric form, and of a suitable mapping function of this model onto perceived colors. Such a model determines the colors that can be represented, as a combination of a set of basic parameters called *channels*.⁵ Each channel takes on values in a range, that in the following we will assume to be $[0, 1]$ unless otherwise stated. Of course, practical storing and transmission of these values in computer systems may suggest more comfortable byte-based representations, usually as discrete (integer) values in the $[0, N - 1]$ interval (where $N = 2^k$ with k the number of bits reserved for the representation of a channel). In such a case, a value $x \in [0, 1]$ can be suitably scaled to an $n \in [0, N - 1]$ by the following formula:

$$n = \min(\text{round}(N \cdot x), N - 1)$$

and back from n to x with a trivial proportion. A typical value adopted for N , that is considered a good trade-off between space requirements, ease of manipulation and quality of the representation, is $2^8 = 256$, which yields a channel range $[0, 255]$. Indeed, $k = 8$ bits, or a byte, per channel ensures straightforward compliance to traditional computer science memory organization and measurement. It is also usual that a color is compactly identified as a single integer obtained by juxtaposing the corresponding channel values as consecutive bits, and hence adding or subtracting given amounts to such a compound value provides results that cannot be foreseen from a human perception viewpoint.

⁵An indication that one of the main interests towards images is their transmission.

Table 2.4 The main colors as represented in coordinates of the different color spaces

Color	R	G	B	C	M	Y	Y	U	V	H	S	V L
K	0	0	0	1	1	1	0	0.5	0.5	–	0	0 0
R	1	0	0	0	1	1	0.3	0.33	1	0	1	1 0.5
G	0	1	0	1	0	1	0.58	0.17	0.08	1/3	1	1 0.5
B	0	0	1	1	1	0	0.11	1	0.42	2/3	1	1 0.5
C	0	1	1	1	0	0	0.7	0.67	0	1/2	1	1 0.5
M	1	0	1	0	1	0	0.41	0.83	0.92	5/6	1	1 0.5
Y	1	1	0	0	0	1	1	0	0.58	1/6	1	1 0.5
W	1	1	1	0	0	0	1	0.5	0.5	–	0	1 1
Gray	x	x	x	x	x	x	x	0.5	0.5	–	0	x x

RGB The most widespread color space is *RGB* (or *three-color*), acronym of the colors on which it is based (Red, Green and Blue). It is an additive color system, meaning that any color is obtained by adding to black a mix of lights carrying different amounts of the base colors. The intensity of each color represents its shade and brightness. The three primary colors represent the dimensions along which any visible color is defined, and hence the whole space consists of a cube of unitary side. Each color is identified as a triple (R, G, B) that locates a point in such a space (and *vice-versa*). In particular, the triples that represent the base colors correspond to the eight corners of the cube (see Table 2.4), while gray levels (corresponding to triples in which $R = G = B$) are in the main diagonal of such a cube, going from Black (usually denoted by K) to White (W). Note that this color space is *non-linear*, in the sense that its dimensions have no direct mapping to the typical dimensions underlying human perception. As a consequence, it is not foreseeable what will be the result of acting in a given way on each channel value because changing a single component changes at once many different parameters such as color tone, saturation and brightness.

The success of RGB is due to the fact that it underlies most analog and digital representation and transmission technologies, and is supported by many devices, from old CRT (Cathode Ray Tube) monitors to current LCD (Liquid Crystal Display) and LED (Light-Emitting Diode) screens, from scanners to cameras.

YUV/YC_bC_r The *YUV* color space has its main feature in the Y component that denotes the *luminance* channel (i.e., the sum of primary colors), and corresponds to the overall intensity of light carried by a color. In practical terms, the luminance information alone yields grayscale colors, and hence is sufficient to display images on black&white screens. The values of the colors can be derived as the difference between the chrominance channels U (difference from blue) and V (difference from red). Green is obtained by subtracting from the luminance signal the signals transmitted for red and blue. YC_bC_r is the international standardization of YUV.

YUV was originally developed to switch from a color RGB signal to a black and white one because the same signal could be straightforwardly exploited both by

color receivers (using all three channels) and by black&white receivers (using only the *Y* channel).

CMY(K) *CMY* is a color model named after the acronym of the basic colors that it exploits: Cyan, Magenta, and Yellow. It is a subtractive system, because starting from white (such as a sheet to be printed), light is progressively subtracted along the basic components to determine each color, up to the total subtraction that yields black.

Hence, in theory a separate black component is not necessary in *CMY*, because black can be obtained as the result of mixing the maximum level of the three basic colors. However, in practice the result obtained in this way is not exactly perceived as black by the human eye. For this reason, often an additional *K* channel (where *K* stands for black, also called *Key color*) is specifically exploited for the gray-level component of the colors, this way obtaining the *CMYK* (or *four-color*) color space.

CMY(K) is suitable for use in typography, and indeed its basic colors correspond to the colors of the inks used in color printers.

HSV/HSB and HLS The *HSV* (acronym of Hue Saturation Value) color space (sometimes referred to as *HSB*, where *B* stands for Brightness) is a three-dimensional space where the vertical axis represents the brightness, the distance from such an axis denotes saturation, and the angle from the horizontal is the hue. This yields a cylinder having black on the whole lower base (but conventionally placed in its center, which is the origin of the axes), and a regular hexagon inscribed in the upper base circle, whose corners correspond to the *R*, *Y*, *G*, *C*, *B*, *M* basic colors. White stands in the center of the upper base, and gray levels lie along the vertical. With black being in the whole base, the space is again non-linear.

HLS (short for Hue Luminance Saturation) is very similar to *HSV*: the vertical axis is still brightness (here called Luminance), the angle is still Hue and the distance (radius) is again Saturation, but the latter are defined here in a different way. In particular, black is again on the whole lower base (although conventionally placed in its center, which is the origin of the axes), but white now takes the whole upper base (although conventionally placed in its center), and the regular hexagon whose corners are the basic colors cuts the cylinder in the middle of its height. It is again non-linear, because black takes the whole lower base, and white takes the whole upper base.

Comparison among Color Spaces Table 2.4 compares the representations of main colors in different spaces. The color distribution is uniform in the *RGB* cube, while in *HSV* it is more dense towards the upper base and goes becoming more rare towards the lower base (for which reason it is often represented as an upside-down pyramid), and is more dense in the middle of the *HLS* cylinder, becoming more rare towards both its bases (hence the representation as a double pyramid, linked by their bases).

Raster Graphics

The term *raster* refers to an image representation made up of dots, called *pixels* (a contraction of the words ‘picture elements’), stored in a matrix called *bitmap*. It is the result of a spatial discretization of the image, as if a regular grid, called *sampling grid*, were superimposed to an analog image: then, the portion of image that falls in each of the grid cells is mapped onto a pixel that approximately represents it as just one value. If also the values a pixel can take on are discrete, the original image is completely discretized. The number of possible values a pixel can take on is called the *density* (or *depth*) of the image. Each value represents a color, and can range from a binary distinction between black and white, to a scale of gray levels, to a given number of colors. The *resolution* corresponds to the number of (horizontal/vertical) grid meshes (pixels) that fit in a given linear distance (usually an *inch*, i.e., 2.54 cm); clearly, narrower meshes provide a closer approximation of the original image. In addition to using a given color space, some computer formats for image representation provide an additional (optional) *alpha channel* that expresses the degree of transparency/opacity of a pixel with respect to the background on which the image is to be displayed. Such a degree is denoted by means of a numeric value (whose range depends on the format in use, but is often the same as the other color space channels).

The amount of memory space needed to store the information concerning an image thus depends on both resolution and density, as reported in Table 2.5 along with other representation-related parameters. Indeed, a single bit per pixel is enough to distinguish among black and white; 4 bits (16 levels) can be used for low-quality gray-level or very-low-quality color images; 8 bits (and hence one byte) per pixel are fine for good-quality gray-level (using a single luminance channel) or for low-quality color images; three 8-bit channels (e.g., RGB), for a total of 3 bytes, allow expressing more than 16 million colors that are considered a sufficient approximation of what can be perceived by the eye (whence the term *true color* for this density).

There are many formats to save an image on file; the choice depends on several factors, among which the allowed density, the use of compression or not, and, if

Table 2.5 Indicative space needed to store a non-compressed raster image having width *W* and height *H* for different density values, plus other parameters

Density	2	16	256	16 mil.
Usual exploitation	(black&white)	gray-level or color		true color
Bits/pixel	1	4	8	24
Image size (in bytes)	$W \cdot H / 8$	$W \cdot H / 2$	$W \cdot H$	$W \cdot H \cdot 3$
Usual number of channels	1	1	1	3

Table 2.6 Bitmap file header

Byte	Name	stdval	Description
1–2	bfType	19778	The characters ‘BM’ to indicate it is a BMP file
3–6	bfSize		File size in bytes
7–8	bfReserved1	0	Always 0
9–10	bfReserved2	0	Always 0
11–14	bfOffBits	1078	Offset from the file start to the first data byte (the beginning of the pixel map)

used, its being information lossy or lossless,⁶ the algorithm and the compression ratio.

BMP (BitMaP) Introduced in 1990 by Microsoft for Windows 3.0, not patented, the BMP format soon gained wide acceptance by graphic programs. It allows depths of 1, 4, 8, 16, 24 or 32 bits/pixel. Although provided for the cases of 16 and 256 colors, compression is usually not exploited due to the inefficiency of the RLE (lossless) algorithm. Thus, the most used (uncompressed) version has a representation on disk similar to its RAM counterpart. Even if this improves read and write speed, because the processor is not in charge of thoroughly processing the data contained in the file, the drawback is that the space needed to represent an image is quite large, which prevents the use of this format on the Web. Another shortcoming is the fact that, in version 3 (the most commonly used), differently from versions 4 and 5, the alpha channel is not provided and personalized color spaces cannot be defined. The latest version allows exploiting a color profile taken from an external file, and embedding JPEG and PNG images (to be introduced later in this section).

An image in BMP format is made up as follows:

- Bitmap file header** Identifies the file as a BMP-encoded one (see Table 2.6).
- Bitmap information header** Reports the size in pixels of the image, the number of colors used (referred to the device on which the bitmap was created), and the horizontal and vertical resolution of the output device that, together with the width and height in pixels, determine the print size of the image in true size; see Table 2.7.
- Color palette** An array (used only for depth 1, 4 or 8) with as many elements as the number of colors used in the image, each represented by a RGBQUAD, a 4-byte structure organized as follows:

Byte	1	2	3	4
Name	rgbBlue	rgbGreen	rgbRed	rgbReserved
Description	Amount of blue	Amount of green	Amount of red	0 (unused)

⁶An image compressed lossily, if repeatedly saved, will tend to lose quality, up to not being able to recognize its content anymore.

Table 2.7 Bitmap information header

Byte	Name	stdval	Description
15–18	biSize	40	Size of the Bitmap information header (in bytes)
19–22	biWidth	100	Image width (in pixels)
23–26	biHeight	100	Absolute value = image height (in pixels); sign = scan direction of the lines in the pixel map: + bottom-up (most common variant) – top-down
27–28	biPlanes	1	Number of planes of the device
29–30	biBitCount	8	Number of bits per pixel
31–34	biCompression	0	Pixel map compression: 0 (BI_RGB) not compressed 1 (BI_RLE8) compressed using RLE. Valid only for biBitCount = 8 and biHeight > 0 2 (BI_RLE4) compressed using RLE. Valid only for biBitCount = 4 and biHeight > 0 3 (BI_BITFIELDS) not compressed and encoded according to personalized color masks. Valid only for biBitCount ∈ {16, 32}; unusual 4 (BI_JPEG) the bitmap embeds a JPEG image (in version 5) 5 (BI_PNG) the bitmap embeds a PNG image (in version 5)
35–38	biSizeImage	0	Size of the pixel map buffer (in bytes). Can be 0 when biCompression = BI_RGB
39–42	biXPelsPerMeter	0	Horizontal resolution of the output device (in pixels/meter); 0 if unspecified.
43–46	biYPelsPerMeter	0	Vertical resolution of the output device (in pixels/meter); 0 if unspecified
47–50	biClrUsed	0	Number of colors used in the bitmap if biBitCount = 1: 0 if biBitCount ∈ {4, 8}: number of entries actually used in the color palette; 0 indicates the maximum (16 or 256) else: number of entries in the color palette (0 meaning no palette). For depths greater than 8 bits/pixel, the palette is not needed, but it can optimize the image representation
51–54	biClrImportant	0	if biBitCount ∈ {1, 4, 8}: number of colors used in the image; 0 indicates all colors in the palette else if a palette exists and contains all colors used in the image: number of colors else: 0

Image pixels An array of bytes containing the data that make up the actual image. It is organized by image rows (called *scanlines*), usually stored in the file bottom-up, each with the corresponding pixels from left to right (thus the first pixel is the bottom-left one in the image, and the last is the top-right one). Each pixel consists of a color, expressed as an index in the palette (for depth 1, 4 or 8) or directly as its RGB chromatic components, one after the other (for larger depth values). For instance, in two-color (usually, but not necessarily, black&white) images the palette contains two RGBQUADs, and each bit in the array represents a pixel: 0 indicates the former color in the palette, while 1 indicates the latter. The length in bytes of each scanline thus depends on the number of colors, format and size of the bitmap; in any case, it must be a multiple of 4 (groups of 32 bits), otherwise NUL bytes are added until such a requirement is fulfilled. In version 5 this structure can also embed JPG or PNG images.

The overall size in bytes of an image will be:

$$54 + 4 \cdot \left(u(15 - b) \cdot 2^b + h \cdot \left\lceil \frac{w \cdot b}{32} \right\rceil \right),$$

where 54 is the size of the first two structures, b is the depth in bits/pixel, and h and w denote, respectively, the height and width of the image in pixels. 2^b yields the size in bytes of the palette, and $u(x)$ denotes the Heaviside unit function: for depth values between 1 and 8 it yields 1; for depth values greater or equal to 16 it yields 0, and hence no space is allocated for the palette. Value 32 as a denominator of the ceiling function is exploited to obtain multiples of 4 bytes, as required by the specifications.

GIF (Graphics Interchange Format) Released in 1987 by Compuserve for image transmission on the Internet, *GIF* [1] is today supported by all browsers, and widely exploited in Web pages thanks to its useful features, among which fast displaying, efficiency, the availability of transparency, the possibility to create short animations that include several images and to have a progressive rendering. It allows using at most 256 colors in each image, chosen from a palette called *Color Table*, which makes it not suitable for photographic (halftone) images. A binary alpha channel is supported. It exploits the LZW compression algorithm, that until 2003 in America (and until 2004 in the rest of the world) was patented, for which reason whoever wrote software that generated GIF images had to pay a fee to CompuServe and Unisys.

GIF is organized into blocks and extensions, possibly made up of sub-blocks and belonging to three categories:

Control blocks Image-processing information and hardware-setting parameters.

Header Takes the first 6 bytes: the first three are characters that denote the format ('GIF'), the other three specify the version ('87a' or '89a').

Logical Screen Descriptor Always present next to the header, contains global information on image rendering according to the structure reported in Table 2.8. The *logical screen* denotes the area (the monitor, a window, etc.) where the image is to be displayed.

Table 2.8 Organization of a GIF logical screen descriptor

Byte	Name	Type	Description
0–1	LogicalScreenWidth	Unsigned	Horizontal coordinate in the logical screen where the top-left corner of the image is to be displayed
2–3	LogicalScreenHeight	Unsigned	Vertical coordinate in the logical screen where the top-left corner of the image is to be displayed
4	Packed field		Specifies: <i>g</i> a flag (usually true) that indicates the presence of a global palette (<i>Global Color Table</i>), used by default for all images that do not have a local one <i>k</i> the color resolution, a 3-bit integer meaning that the palette colors are chosen from an RGB space defined on $k + 1$ bits per channel <i>s</i> a flag that indicates whether colors in the palette are ordered by decreasing importance <i>i</i> the palette size, a 3-bit integer meaning that $i + 1$ bits are used for the index (and hence at most 2^{i+1} entries are allowed)
5	BackgroundColorIndex	Byte	The alpha channel, denoted as the index of the palette color that, during visualization, is replaced by the background
6	PixelAspectRatio	Byte	The width/height ratio for pixels, ranging from 4:1 to 1:4, approximated in 1/64th increments 0 no aspect ratio information [1..255] ($PixelAspectRatio + 15$)/64

Color Table A color palette that can be local (valid only for the image immediately following it) or global (valid for all images that do not have a local one). Both kinds have the same structure that consists of a sequence of $3 \cdot 2^{i+1}$ bytes representing RGB color triplets.

Graphic Control Extension Controls the visualization of the immediately subsequent image, such as what to do after the image has been displayed (e.g., freezing the image, or going back to the background or to what was in place before), whether user input is needed to continue processing, the visualization delay (in 1/100th of second) before continuing processing (important for animations), local color table and transparency settings (as in the Logical Screen Descriptor). Animations include one such block for each frame that makes up the animation.

Trailer Indicates the end of the GIF file.

Graphic-Rendering blocks Information needed to render an image.

Image Descriptor Contains the actual compressed image (in the *Table Based Image Data* sub-block), plus information concerning its size and position in the logical screen (in pixels), the local color table (as in the Logical Screen Descrip-

Table 2.9 Header block of a TIFF file

Byte	Description
0–1	Two characters denoting the byte ordering used in the file: II <i>little-endian</i> (from the least to the most significant bit), used by Microsoft MM <i>big-endian</i> (from the most to the least significant bit), used by the Macintosh
2–3	The number 42 (denoting the format TIFF)
4–7	The offset in byte of the first IFD

tor) and the presence of *interlacing* (i.e., the possibility to display it in stages by progressively refining the representation).
Plain Text Extension Allows rendering text, encoded as 7-bit ASCII, as images on the screen. Defines the features of grids in which each character is rendered as a raster image. Not all interpreters support this extension.

Special Purpose blocks Information not affecting image processing.

Comment Extension Contains comments that are not going to be displayed.
Application Extension Contains data that can be exploited by specific programs to add special effects or particular image manipulations.

The image is compressed using the LZW algorithm, where the alphabet is known: it is the number of possible pixel values, reported in the header. Actually, GIF applies a modification to the LZW algorithm to handle cases in which the initially defined size of the compression codes (N bits per pixel) turns out to be insufficient and must be extended dynamically. This is obtained by adding to A two additional meta-symbols: C (clear, used to tell to the encoder to re-initialize the string table and to reset the compression size to $N + 1$ for having more codes available) and E (end of data).

TIFF (Tagged Image File Format) Developed by Microsoft and Aldus (that subsequently joined Adobe, which to date holds the patent), *TIFF* [2] is currently the most used, flexible and reliable technique for storing bitmap images in black&white, gray-scale, color scale, in RGB, CMYK, YC_bC_r representation. As a drawback, the file size is quite large. There are no limits to the size in pixel that an image can reach, nor to the depth in bits. A TIFF file can be saved with or without compression, using the LZW or the Huffman method. It can embed meta-information (the most common concerns resolution, compression, contour trace, color model, ICC profile) in memory locations called *tags*. Extensions of the format can be created by registering new tags with Adobe, but the possibility of adding new functionality causes incompatibility between graphic programs and lack of support on the browsers' side. It is supported by the most common operating systems, and two versions of it exist, one for Windows and one for Macintosh, that differ because of the byte ordering.

The file structure includes a data block (*header*) organized as in Table 2.9, plus one or more blocks called *Image File Directories* (IFD), each of which (supposing it starts at byte I) is structured as follows:

Table 2.10 Structure of a TIFF Image File Directory entry. Note that each field is a one-dimensional array, containing the specified number of values

Byte	Description
0–1	Tag that identifies the field
2–3	Type: 1. BYTE, 8-bit unsigned integer 2. ASCII, 7-bit value 3. SHORT, 2-byte unsigned integer 4. LONG, 4-byte unsigned integer 5. RATIONAL, made up of 2 LONGs (numerator and denominator) 6. SBYTE, 8-bit integer in two’s complement 7. UNDEFINED, byte that may contain everything 8. SSHORT, 2-byte integer in two’s complement 9. SLONG, 4-byte integer in two’s complement 10. SRATIONAL, made up of 2 SLONGs (numerator and denominator) 11. FLOAT, in single precision (4 bytes) 12. DOUBLE, in double precision (8 bytes)
4–7	Number of values of the indicated type
8–11	Offset to which the value is placed (or the value itself if it can be represented in 4 bytes)

- Bytes $I, I + 1$: the number of directory entries in the IFD (say N)
- Bytes from $I + 2 + 12 \cdot (k - 1)$ to $I + 2 + 12 \cdot k - 1$: the k -th directory entry (made up of 12 bytes and organized as in Table 2.10), for $1 \leq k \leq N$
- Bytes from $I + 2 + 12 \cdot N$ to $I + 2 + 12 \cdot N + 3$: the address of the next IFD (0 if it does not exist).

Each TIFF file must have at least one IFD that, in turn, must contain at least one entry. The *Tag* field in the entry expresses as a numeric value which parameter is being defined (e.g., 256 = ImageWidth, 257 = ImageLength, 258 = BitsPerSample, 259 = Compression, etc.). Different kinds of images require different parameters, and a parameter may take on different values in different kinds of images.

Images can also be stored in frames, which allows for a quick access to images having a large size, and can be split into several ‘pages’ (e.g., all pages that make up a single document can be collected into a single file).

JPEG (Joint Photographic Experts Group) *JPEG* [15, 19], named after the acronym of the group that in 1992 defined its standard, aims at significantly reducing the size of raster images, mainly of halftone pictures such as photographs, at the cost of a lower quality in enlargements. Indeed, it provides lossy compression and true color. It is ideal for efficient transmission and exploitation of images on the Web, even in the case of photographs including many details, which makes it the most widely known and spread raster format. However, it is not suitable for images that are to be modified because each time the image is saved an additional compression is applied, and thus increasingly more information will be lost. In such

a case, it is wise to save intermediate stages of the artifact in a lossless format. Meta-information cannot be embedded in the file.

The dramatic file size reduction with reasonable loss in terms of quality is obtained at the expenses of information that, in any case, the eye cannot perceive, or perceives negligibly, according to studies in human physiology. JPEG encoding of an image is very complex, and its thorough discussion is out of the scope of this book. However, a high-level, rough survey of its most important phases can be useful to give the reader an idea of the involved techniques.

1. Switch from the RGB color space channels to the YUV ones. This singles out luminance, to which the eye is more sensible than it is to colors. Thus, luminance (Y) can be preserved in the next step, while losing in chrominance.
2. Reduce chrominance (U , V) components by subsampling⁷ (available factors are 4:4:4, 4:2:2, 4:2:0). This replaces 2×1 , 2×2 or larger pixel blocks by a single value equal to the average of their components, and already reduces by 50–60% the image size.
3. Split each channel in blocks of 8×8 pixels, and transpose each value to an interval centered around the zero.
4. Apply to each such block the *Discrete Cosine Transform (DCT)* that transposes the image into a *frequency-domain* representation. The forward formula is:

$$F(u, v) = \frac{1}{4} C(u) C(v) \left[\sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right]$$

while the inverse formula, to go back to the original image, is:

$$f(x, y) = \frac{1}{4} \left[\sum_{u=0}^7 \sum_{v=0}^7 F(u, v) C(u) C(v) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right],$$

where $f(x, y)$ is the original pixel, $F(u, v)$ is the DCT coefficient and $C(u)$, $V(u)$ are normalization factors. The resulting 8×8 matrix, whose values are rounded to the closest integer, aggregates most of the signal in the top-left corner, called DC coefficient (that is the average of the image luminance), while the other cells are called AC coefficients, as represented in Fig. 2.3 on the left.

5. Divide each DCT coefficient by the corresponding value (between 0 and 1) in an 8×8 *quantization table*. This amplifies the effect of the previous step. Since the eye can note small differences in brightness over a broad area, but cannot distinguish in detail high frequency brightness variations, high frequencies can be cut off (using lower values in the quantization table, while higher values are

⁷The *subsampling* scheme is commonly expressed as an $R : f : s$ code that refers to a conceptual region having height of 2 rows (pixels), where:

R width of the conceptual region (*horizontal sampling reference*), usually 4;

f number of chrominance samples in the first row of R pixels;

s number of (additional) chrominance samples in the second row of R pixels.

<i>DC</i>	<i>AC</i> ₀₁	<i>AC</i> ₀₂	<i>AC</i> ₀₃	<i>AC</i> ₀₄	<i>AC</i> ₀₅	<i>AC</i> ₀₆	<i>AC</i> ₀₇	1	2	6	7	15	16	28	29
<i>AC</i> ₁₀	<i>AC</i> ₁₁	<i>AC</i> ₁₂	<i>AC</i> ₁₃	<i>AC</i> ₁₄	<i>AC</i> ₁₅	<i>AC</i> ₁₆	<i>AC</i> ₁₇	3	5	8	14	17	27	30	43
<i>AC</i> ₂₀	<i>AC</i> ₂₁	<i>AC</i> ₂₂	<i>AC</i> ₂₃	<i>AC</i> ₂₄	<i>AC</i> ₂₅	<i>AC</i> ₂₆	<i>AC</i> ₂₇	4	9	13	18	26	31	42	44
<i>AC</i> ₃₀	<i>AC</i> ₃₁	<i>AC</i> ₃₂	<i>AC</i> ₃₃	<i>AC</i> ₃₄	<i>AC</i> ₃₅	<i>AC</i> ₃₆	<i>AC</i> ₃₇	10	12	19	25	32	41	45	54
<i>AC</i> ₄₀	<i>AC</i> ₄₁	<i>AC</i> ₄₂	<i>AC</i> ₄₃	<i>AC</i> ₄₄	<i>AC</i> ₄₅	<i>AC</i> ₄₆	<i>AC</i> ₄₇	11	20	24	33	40	46	53	55
<i>AC</i> ₅₀	<i>AC</i> ₅₁	<i>AC</i> ₅₂	<i>AC</i> ₅₃	<i>AC</i> ₅₄	<i>AC</i> ₅₅	<i>AC</i> ₅₆	<i>AC</i> ₅₇	21	23	34	39	47	52	56	61
<i>AC</i> ₆₀	<i>AC</i> ₆₁	<i>AC</i> ₆₂	<i>AC</i> ₆₃	<i>AC</i> ₆₄	<i>AC</i> ₆₅	<i>AC</i> ₆₆	<i>AC</i> ₆₇	22	35	38	48	51	57	60	62
<i>AC</i> ₇₀	<i>AC</i> ₇₁	<i>AC</i> ₇₂	<i>AC</i> ₇₃	<i>AC</i> ₇₄	<i>AC</i> ₇₅	<i>AC</i> ₇₆	<i>AC</i> ₇₇	36	37	49	50	58	59	63	64

Fig. 2.3 Schema of a JPEG DCT coefficients (on the *left*), and sequence of the corresponding zigzag traversal (on the *right*)

used for exalting low frequencies). This is the most information-lossy step. Each transformed block represents a frequency spectrum.

6. Rearrange AC cells in a 64-elements vector following a zigzag route (as in the schema on the right in Fig. 2.3), which increases the chance that similar cells become adjacent.
7. Perform final lossless *entropy encoding*, exploiting several algorithms:
 - RLE compression on the AC components, creating pairs (*skip*, *value*) where *skip* is the number of values equal to 0 and *value* is the next value different than zero. Since the array resulting from the zigzag reading contains many consecutive 0 values, this method saves significant space.
 - *Differential Pulse Code Modulation (DPCM)* compression on the DC component: the DC component of the *i*th block is encoded as the difference with respect to the preceding block ($DC_i - DC_{i-1}$); indeed, it turned out that there exists a statistical relationship between DC components of consecutive blocks.

Example 2.5 (Compression in JPEG) The sequence 150, 147, 153, 145, 152, 160 is stored by DPCM as the value 150 followed by the differences with the remaining values: -3, 6, -8, 7, 8.

- Huffman encoding for the final data. In this way, a further compression of the initial data is obtained, and, as a consequence, a further reduction of the JPEG image size.

A JPEG file is made up of *segments*, each started by a 2-byte *marker* where the first byte is always FF_{16} and the second denotes the type of segment. If any, the third and fourth bytes indicate the length of the data. In entropy-coded data (only), immediately following any FF_{16} byte, a 00_{16} byte is inserted by the encoder, to distinguish it from a marker. Decoders just skip this 00_{16} byte (a technique called *byte stuffing*).

PNG (Portable Network Graphics) *PNG* is an open and free format [7], created by some independent developers as an alternative to the GIF for compressed images, and approved by W3C in 1996. It has been accepted as ISO/IEC 15948:2003 standard. A motivation for it came from GIF being patented, and from the purpose

(announced in 1995) of its owners (CompuServe and Unisys) to impose a fee to third parties that included GIF encoders in their software. Thus, many of its features are similar to GIF, as reported in the following list:

Compression Mandatory in PNG, exploits the information lossless *Zlib* algorithm⁸ [17], that yields results in general 20% better than GIF, and can be significantly improved by using filters that suitably rearrange the data that make up the image.

Error check The *CRC-32* (32-bits *Cyclic Redundancy Check*) system associates check values to each data block, and is able to immediately identify any corruption in the information saved or transmitted through the Internet.

Color Full 24-bit RGB (true color) mode is supported as a range of colors for images.

Alpha channel A transparency degree ranging over 254 levels of opacity is allowed.

Interlacing 1/64th of the data is sufficient to obtain the first, rough visualization of the image.

Gamma correction Allows, although approximately, to balance the differences in visualization of images on different devices.

Thus, compared to GIF, PNG improves performance (interlacing is much faster, compression rates are much higher) and effectiveness (the number of colors that can be represented is not limited to a maximum of 256, the alpha channel is not limited to a binary choice between fully transparent and completely opaque), but does not provide support for animated images (although a new format, *Multiple-image Network Graphics* or *MNG*, has been defined to overcome this lack).

Encoding an image in PNG format is obtained through the following steps:

1. **Pass extraction:** in order to obtain a progressive visualization, the pixels in a PNG image can be grouped in a series of small images, called *reduced images* or *passes*.
2. **Serialization** by scanline, top-down among scanlines and left-to-right within scanlines.
3. **Filtering** of each scanline, using one of the available filters.
4. **Compression** of each filtered scanline.
5. **Chunking:** the compressed image is split into packets of conventional size called *chunks*, to which an error checking code is attached.
6. Creation of the **Datastream** in which chunks are inserted.

Textual descriptions and other auxiliary information, that is not to be exploited during decompression, can be embedded in the file: a short description of the image, the background color, the chromatic range, the ICC profile of the color space, the image histograms, the date of last modification and the transparency (if not found in the file). The file size is larger than JPEG, but compression is lossless.

⁸A variant of the *LZ77*, developed by J.-L. Gailly for the compression part (used in zip and gzip) and by M. Adler for the decompression part (used in gzip and unzip), and almost always exploited nowadays in ZIP compression. It can be optimized for specific types of data.

DjVu (DejaVu) *DjVu* (“déjà vu”) [8] was being developed since 1996 at AT&T Labs, and first released in 1999. It is intended to tackle the main problems related to document digitization, preservation and transmission. Its main motivation is that the huge number of legacy paper documents to be preserved cannot be cheaply and quickly re-written in a natively digital format, and sometimes it is not desirable either because (part of) their intrinsic value comes from their original visual aspect. Thus, the only technology that can be sensibly applied is scanning, whose drawback is that the image representation of color documents requires significant amounts of space. Usual ways to deal with this problem are lossy compression, as in JPEG, or color depth reduction to gray-level or black&white. However, sometimes the lower definition due to compression is unacceptable, and color cannot be stripped off without making the document meaningless. In these cases, the file size seriously affects its transmittability which, in turn, prevents its embedding into Web pages and hence severely limits access to it. DjVu solves all these problems at once by preserving the original aspect of the document, without losing in quality on sensible components (such as text), and keeping the size of the outcome within limits that significantly outperform JPEG, GIF and PDF. To give an idea, a 300 dpi scanned full color A4 page would take 25 MB that can be represented in less than 100 kB.

The key for this optimal solution lies in the selective application of different types and rates of compression to different kinds of components: text and drawings, usually being more important to the human reader, are saved in high quality (to appear sharp and well-defined), while stronger compression can be applied to halftone pictures in order to preserve their overall appearance at a sufficient tradeoff between quality and space. It is also a progressive format: most important components (text ones) are displayed first and quickly, and the others (pictures and then background) are added afterwards as long as the corresponding data is gained. A DjVu document may contain multiple (images of) pages, placed in a single file or in multiple sources: the former solution is more comfortable for handling, but involves a serialization for their exploitation (a page cannot be displayed until all the previous ones have been loaded), while the latter allows for quicker response and selective transmission. Additional useful features that a DjVu file can contain is meta-data (e.g., hyperlinks) in the form of annotations, a hidden text layer that allows reading plain text corresponding to what is displayed, and pre-computed thumbnails of the pages that can be immediately available to the interpreter.

Several types of compression are provided by DjVu. The *JB2* data compression model is exploited for binary (black&white) images, where white can be considered as being the background and black as the significant content (usually text and drawings). It is a bitonal technique that leverages the repetition of nearly identical shapes to obtain high compression rates, and essentially stores a dictionary of prototypical shapes and represents each actual shape as its difference from one of the prototypes. *IW44* wavelet representation is exploited for pictures. Other compression methods are available as well.

According to its content type, a DjVu page consists of an image in one of the following formats:

Table 2.11 Structure of a DjVu chunk

Field	Type	Description
ID	Byte [4]	An ID describing the use of the chunk as a string, e.g., FORM container chunk FORM:DJVM multi-page document FORM:DJVU single-page document FORM:DJVI shared data (e.g., the shape dictionary) FORM:THUM embedded thumbnails chunks Djbz shared shape table Sjbz mask data (BZZ-compressed JB2 bi-tonal) FG44 foreground data (IW44-encoded) BG44 background data (IW44-encoded) TH44 thumbnails data (IW44-encoded) FGbz color data for the shapes (JB2) BGjp background image (JPEG-encoded) FGjp foreground image (JPEG-encoded)
Length	32-bit integer	The length of the chunk data (in Big Endian byte ordering)
Data	Byte [Length]	The chunk data

Photo used for color or gray-level photographic images, compressed using IW44.
Bi-level used for black&white images, handled using JB2.
Compound used for pages that mix text and pictures. The background and foreground are identified, separated and represented in two different layers. A third layer (called *mask layer*) is used to distinguish between background and foreground. The background contains paper texture (to preserve the original look-and-feel of the document) and pictures, while the foreground contains text and drawings. The foreground is represented as a bi-level image, called *foreground mask*, encoded in JB2 where black denotes the foreground pixels and white the background ones. Foreground colors can be represented by specifying the color of each foreground shape separately, or as a small image, sub-sampled with a factor ranging from 1 to 12 (usually 12: smaller values do not significantly increase quality), that is scaled up to the original size of the document and drawn on the background image. The background image is obtained through progressive IW44 refinements, scaled down with a sub-sampling factor between 1 and 12 (usually 3, smaller if high-quality pictures are to be represented, higher if no pictures are present).

DjVu files are represented in UTF-8 encoded Unicode, and identified by a header having hexadecimal values 41 54 26 54 in the first four bytes. They are structured in a variable number of *chunks*, that contain different types of information according to the field structure described in Table 2.11. An external FORM chunk contains all the others, with no further nesting. Each chunk must begin on an even byte boundary, including an initial 00 padding byte to ensure this in case it does not hold (as in the beginning of the file). Chunks of unknown type are simply ignored.

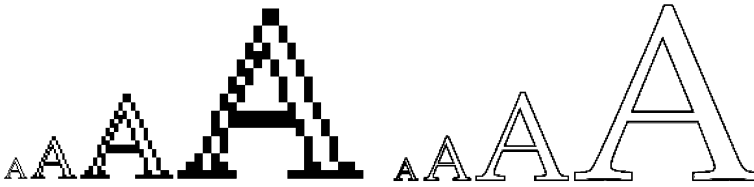


Fig. 2.4 Comparison between enlargements in a raster format (on the *left*) and in a vector one (on the *right*)

Vector Graphic

Vector Graphic is a technique to describe images exploiting mathematical/geometrical primitives, such as points, lines, curves and polygons. Further information can be attached to each element as well (e.g., color). Compared to raster formats, in which images are described as a grid of syntactically unrelated colored points, the advantages of vectorial images are: better quality (in particular, when zooming or up-scaling them—see Fig. 2.4), limited memory requirements, easy modification (each component can be managed separately and independently of the others by acting on its descriptive primitives), progressive rendering facility, and easier analysis (the description language might allow applying logical deduction and other kinds of inference to identify desired shapes). On the other hand, it is very hard to describe photographs (because the details to be translated into mathematical primitives would be a huge number). Moreover, very detailed images, such as architecture and engineering projects, might require the processor to perform significantly heavier calculations to display them.

SVG (Scalable Vector Graphic) Produced by W3C, SVG [25] dates back to 1999. In addition to vectorial shapes, it allows embedding raster graphics and text, and attaching information to any component (e.g., name, available colors, relationships to other objects). It is suitable for high-complexity images, such as architecture and engineering projects (it allows scaling up or down an image at any ratio), or 3D animations. The main disadvantage that can, however, be tackled by means of suitable tools is given by the grammar complexity. All Internet browsers, except Internet Explorer, support SVG. Although conceived as a vector graphics markup language, it may also act as a page description language (PDL), like PDF, since all the functionality required to place each element in a precise location on the page is provided.

Everything in SVG is a graphic element (a list is provided in Table 2.12) that can be a generic shape, a text or a reference to another graphic element. The features include nested transformations, clipping paths, alpha masks, filter effects, template objects and extensibility. Among the most relevant are:

Shapes Basic Shapes (straight-lines, polylines, closed polygons, circles and ellipses, rectangles possibly with rounded corners) and Paths (simple or compound shape outlines drawn with curved or straight lines) can be drawn. Ends of lines or

Table 2.12 Graphical elements available in SVG

Element	Attributes	Description
SVG	x, y, width, height, allowZoomAndPan	The most external graphical element that contains all other elements and has the specified <i>width</i> , <i>height</i> and (x, y) position (useful for displaying the image inside a Web page). The <i>allowZoomAndPan</i> flag concerns the possibility to zoom and pan the image
Rect	x, y, width, height	Rectangle having the specified <i>width</i> and <i>height</i> . Optionally, the horizontal and vertical radii (<i>rx</i> , <i>ry</i>) of ellipses used to smooth the angles can be specified
Circle	cx, cy, r	Circle centered at (cx, cy) and having radius <i>r</i>
Ellipse	cx, cy, rx, ry	Ellipse centered at (cx, cy) and having horizontal and vertical radii <i>rx</i> and <i>ry</i> , respectively
Line	x1, y1, x2, y2	Straight line having extremes at (x1, y1) and (x2, y2)
Polyline	List of points	The sequence of points—pairs of (x, y) coordinates—that make up the curve, separated by commas
Polygon	List of points	A polyline automatically closed, whose points are separated by commas or blanks
Text	string, x, y	A string to be displayed
Image	x, y, width, height, xlink:href	Allows embedding a PNG or JPEG raster image, and also textual descriptions (useful to classify a document containing also images)

vertices of polygons can be represented by symbols called *markers* (e.g., arrow-heads).

Text Characters are represented in Unicode expressed as in XML. Text can flow bidirectionally (left-to-right and right-to-left), vertically or along curved paths. Fonts can reference either external font files, such as system fonts, or *SVG fonts*, whose glyphs are defined in SVG. The latter avoid problems in case of missing font files on the machine where the image is displayed.

Colors Painting allows filling and/or outlining shapes using a color, a gradient or a pattern. Fills can have various degrees of transparency. Colors are specified using symbolic names, hexadecimal values preceded by #, decimal or percentage RGB triples like *rgb*(·, ·, ·). (Color or transparency) gradients can be linear or radial, and may involve any number of colors as well as repeats. Patterns are based on predefined raster or vector graphic objects, possibly repeated in any direction. Gradients and patterns can be animated. Clipping, Masking and Composition allow using graphic elements for defining inside/outside regions that can be painted independently. Different levels of opacity in clipping paths and masks are blended to obtain the color and opacity of every image pixel.

Effects Animations can be continuous, loop and repeat. Interactivity is ensured through hyperlinks or association of image elements (including animations) to (mouse-, keyboard- or image-related) events that, if caught and handled by scripting languages, may trigger various kinds of actions.

Metadata According to the W3C's Semantic Web initiative, the Dublin Core (e.g., title, creator/author, subject, description, etc.—see Sect. 5.4.2) or other metadata schemes can be used, plus elements where authors can provide further plain-text descriptions to help indexing, search and retrieval.

SVG sources are pure XML and support DOM [27] (see Sect. 5.1.2). Thus, their content is suitable for other kinds of processing than just visualization: using XSL transformations, uninteresting components can be filtered out, and embedded metadata can be displayed or textually described or reproduced by means of a speech synthesizer. The use of CSSs makes graphical formatting and page layout simple and efficient: their modification is obtained by simply changing the style sheet, without accessing the source code of the document. The files contain many repeated text strings, which makes the use of compression techniques particularly effective. Specifically, using the *gzip* technique on SVG images yields the *SVGZ* format, whose space reduction reaches up to 20% of the original size.

2.3 Layout-Based Formats

This section deals with those formats, often referred to as *Page Description Languages (PDLs)* that are structured in the perspective of document displaying. They focus on the visual/geometrical aspect of documents, by specifying the position in the page of each component thereof. The most widespread formats in this category, thanks to their portability and universality (independence on platform and on the software exploited to create and handle the documents), are the PostScript and the PDF, which will be introduced in the next sections.

PS (PostScript) PostScript (*PS*) [22] is a real programming language (it even allows writing structured programs) for the description and interpretation of pages, developed in 1982 by Adobe Systems. Originally intended for controlling printer devices, and indeed widely exploited in typography, it has been subsequently used for the description of pages and images. It allows providing a detailed description of the printing process of a document, and is characterized by a representation that is independent from the devices on which the pages are to be displayed. Also text characters are considered as graphic elements in PS, which prevents text in a document from being read as such, and hence copied and pasted. Documents generated in this format can be virtually 'printed' on file using suitable drivers; when such files are later interpreted, the original document is perfectly reproduced on different kinds of devices. It has the advantage of not bearing viruses. One of the most fa-

mous PS interpreters is *GhostScript*⁹ that provides a prompt to enter commands and a ‘device’ on which graphic elements are drawn.

PS exploits postfix notation and is based on an operator stack, which makes easier command interpretation although seriously affecting code readability. Many kinds of objects can be represented: characters, geometrical shapes, and (color, grayscale or black&white) raster images. They are built using 2D graphic operators, and placed in a specified position in the page. Objects handled by PS are divided into two categories:

simple Boolean, fontID, Integer, Mark, Name, Null, Operator, Real, Save
compound Array, Condition, Dictionary, File, Gstate, Lock, packedarray, String, Procedure.

Any object in PS can be ‘executed’: execution of some types of objects (e.g., Integers) pushes them on the stack, while execution of other types of objects (e.g., Operators) triggers actions that, usually, consume objects in the stack.

Example 2.6 Sample computation of $5 + 3 = 8$ in GhostScript:

GS> GS>5 3 GS<2> GS<2>add GS<1> GS<1>pstack 8 GS<1>	GhostScript interpreter ready Integer objects 5 and 3 are ‘executed’, i.e., pushed on the stack The prompt indicates that two objects are present in the stack The Operator object <code>add</code> extracts two elements from the operators stack (in case of a stack containing fewer than two objects, the execution would terminate issuing an error), sums them and inserts the result on the stack The prompt says that only one object is present in the stack The <code>pstack</code> Operator displays the stack content (in this case, 8) Still one object in the stack, interpreter ready
--	--

PS allows defining procedures, to be called/run subsequently, by enclosing a sequence of commands in curly brackets and assigning them an identifier (of type Name) through the `def` operator.¹⁰ When it is called, the interpreter runs in sequence the objects in curly brackets.

Example 2.7 Sample definition of a single `printSum` procedure in GhostScript, that includes the sum and print operations:

⁹An open-source project (<http://pages.cs.wisc.edu/~ghost/>) that does not directly handle PS and PDF formats, this way being able to handle some differences in the various versions or slangs of such formats. An associated viewer for PS files, called *GSview*, is also maintained in the project.

¹⁰In this section, PostScript code and operators will be denoted using a teletype font. Operators that can be used with several numbers of parameters are disambiguated by appending the number *n* of parameters in the form `operator/n`.

GS>/printSum {add pstack} def	procedure definition
GS>5 3	push of two Integers on the stack
GS<2>printSum	two objects in the stack; procedure call
8	result
GS<1>	one object (the result) in the stack

The interpreter reads commands that define objects (such as characters, lines and images). Each command runs an associated parametric graphic procedure that, based on its parameters, suitably places *marks* on a *virtual page*. The virtual page is distinct from (it is just a representation in temporary storage of) the physical device (printer or display). The interpretation of a PS document is based on a so-called *painting model* that exploits a ‘current page’ in which it progressively adds the marks (a mark could overshadow the previous ones). When all information for the current page has been read (and hence the page is complete), `showpage` causes the interpreter to call the page printing procedure: all marks in the virtual page are rendered on the output, i.e., transformed into actual drawings on the device, and the current page becomes again empty, ready for another description.

Dictionaries are objects in which a key-value list can be defined. They can be used as a kind of variables or to store procedures. All pre-defined PS operators correspond to procedures defined in the read-only dictionary **systemdict**. The interpreter maintains a separate stack of dictionaries, whose top element is called *current dictionary*. At runtime, whenever a name that does not correspond to a simple type is referenced, a look-up for it is started from the current dictionary down through the dictionary stack. For instance, groups of at most 256 text characters (*character sets*) are represented by programs that define a dictionary (*Font Dictionary*). Such a dictionary contains an **Encoding** vector whose 256 elements are names, each corresponding to a drawing procedure for a character. Each character is denoted by an integer between 0 and 255, used as an index to access the Encoding vector of the character set currently in use. When referenced, the name is extracted and the corresponding drawing procedure is run. Each character set defines different names for its characters. ASCII characters are defined in the **StandardEncoding** vector.

A *path* is a sequence of points, lines and curves, possibly connected to each other, that describe geometrical shapes, trajectories or areas of any kind. A path is made up of one or more sub-paths (straight and curved segments connected to one another). A new path is started by calling `newpath`, always followed by `moveto`, that adds a new non-consecutive sub-path to a previous sub-path (if any). The path can be open or closed (the latter obtained using `closepath`).

The reference system to locate any point in the page consists by default of an ideally infinite Cartesian plane whose origin is placed in the bottom-left corner of the page, and whose horizontal (*x*) and vertical (*y*) axes grow, respectively, towards the right and up-wise. Points are denoted by real-valued coordinates measured in *dots* (corresponding to 1/72th of inch, as defined in the print industry). Independence on the particular device in use is obtained by considering two coordinate systems: one referring to the user (*user space*) and one referring to the output device (*device space*) on which the page will be subsequently displayed. The former is independent of the latter, whose origin can be placed at any point in the page to fit different

printing modes and resolutions and several kinds of supports that can be exploited for visualization. To define the device space, it suffices to indicate the axes origin (by default in the bottom-left with respect to the support on which it will be displayed), their orientation and the unit of measure. The interpreter automatically, and often implicitly, converts the user space coordinates into device space ones. The default settings of the user space can be changed by applying operators that can, e.g., rotate the page or translate the coordinates. Switching from a pair of coordinates (x, y) in the current user space to a new pair (x', y') in the device space is obtained through the following linear equations:

$$x' = ax + cy + t_x, \quad y' = bx + dy + t_y.$$

The coefficients for the transformations are defined by a 3×3 matrix,¹¹ called *Current Transformation Matrix (CTM)*, always present and equal to

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix},$$

and represented as an array of 6 elements $[a \ b \ c \ d \ t_x \ t_y]$ (where the last column, being fixed, is omitted). The possibility of modifying the matrix to distort and move the user system is often exploited also to efficiently manage recurring objects in the document: each object is defined just once using a separate reference system (whose origin is usually placed in the bottom-left corner of the element to be drawn), and then it is drawn multiple times in different places of the page by simply moving the reference system each time it appears. Printing drivers that thoroughly use this technique will produce more compact documents.

The *Graphic State* is the framework in which the operators (implicitly or explicitly) act. It consists of a structure that contains various graphic parameter settings (see Table 2.13), including color, font type, line thickness, the *current path* and the CTM. It is organized LIFO, which complies with the structure in which objects are typically stored (generally independent on each other and nested at various level of depth). It contains objects, but it is not an object itself, and hence it cannot be directly accessed by programs. It can be handled only using two operators that allow changing the internal graphic state without modifying those around it:

- `gstate` pushes in a stack the whole graphic state
- `grestore` pops from the stack the values of a graphic state

Operators are grouped into 7 main categories:

Graphic state operators handle the graphic state.

Coordinate system and matrix operators handle the CTM by combining translations, rotations, reflections, inclinations and scale reductions/enlargements.

¹¹The most common operations that modify the matrix are *translation* of the axes origin, *rotation* of the system of Cartesian axes by a given angle, *scaling* that independently changes the unit of measure of the axes, and *concatenation* that applies a linear transformation to the coordinate system.

Table 2.13 PostScript graphic state parameters

Parameter	Type	Description
Device-independent		
CTM	array	Current transformation matrix
Position	2 numbers	Coordinates of the current point in the user space (initially undefined)
Path	(internal)	Current path, that is, the implicit parameter of some path operators (initially empty)
Clipping path	(internal)	Defines the borders of the area in which the output can be cut (initially the entire page)
Clipping path stack	(internal)	Stores the clipping paths saved through <code>clipsave</code> and not yet returned by <code>cliprestore</code> (Type 3)
Color space	array	Color space in which the values are to be interpreted (initially <i>DeviceGray</i>)
Color	(several)	Varies according to the specified color space (initially black)
Font	dictionary	Contains the set of graphical shapes to represent the characters in a font style
Line width	number	Thickness exploited by lines (initially 1.0)
Line cap	integer	Shape of lines end (initially square)
Line join	integer	Shape of conjunctions of segments
Miter limit	number	Maximum length of a miter line for <code>stroke</code>
Dash pattern	array and numbers	Style for drawing lines by <code>stroke</code>
Stroke adjustment	boolean	Defines whether resolution is to be compensated in case of too thin a thickness (Type 2)
Device-dependent		
Color rendering	dictionary	Collection of parameters to transform CIE-based colors into values suitable for the device color (Type 2)
Overprint	boolean	Specifies whether the underlying area is to be overwritten during printing (Type 2)
Black generation	procedure	Computes the amount of black to be used when converting from RGB to CMYK (Type 2)
Undercolor removal	procedure	Based on the quantity of black used by the <code>black generation</code> procedure computes the amount of the other colors to be used (Type 2)
Transfer	procedure	Correction in case of transfer of pages on particular devices
halftone	(several)	Defines a screen for rendering gray levels and colors
Flatness	numbers	Precision for curve rendering on output devices
Smoothness	numbers	Precision for gradient rendering on output devices (Type 3)
Device	(internal)	Internal structure that represents the current state of the output device

Table 2.14 PostScript graphic procedures

<code>erasepage</code>	paints the page in white
<code>showpage</code>	prints the page on the physical device
<code>fill</code>	fills the current path with the current color (if it denotes a closed line)
<code>eofill</code>	fills with the current color the internal part (as defined by the ‘even–odd rule’) of the ‘current path’
<code>stroke</code>	draws a line along the points in the current path
<code>ufill</code>	fills with the current color a path given in input (called <i>userpath</i>)
<code>ueofill</code>	fills with the current color the internal part (as defined by the ‘even–odd rule’) of a ‘userpath’ given in input
<code>ustroke</code>	draws a line along the points in a <i>userpath</i> given in input
<code>rectfill</code>	fills with the current color a rectangle defined in input
<code>rectstroke</code>	draws the contour of a rectangle defined in input
<code>image</code>	draws a raster image
<code>colorimage</code>	draws a color raster image
<code>imagemask</code>	uses a raster image as a mask to locate zones to be filled with the current color
<code>show</code>	prints on the page the characters in a string
<code>ashow</code>	prints on the page the characters in a string by spacing them of the number of points given in input
<code>kshow</code>	runs the procedure defined in input while printing on the page the characters of a string
<code>widthshow</code>	as <code>show</code> , but modifies character width and height (and spacing accordingly)
<code>awidthshow</code>	combines the effects of <code>ashow</code> and <code>widthshow</code>
<code>xshow</code>	prints the characters in a string on the page using as width of each the values defined in a vector in input
<code>yshow</code>	prints the characters in a string on the page using as height of each the values defined in a vector in input
<code>xyshow</code>	combines the effects of <code>xshow</code> and <code>yshow</code>
<code>glyphshow</code>	prints a character identified by a name (associated to a draw procedure)
<code>cshow</code>	prints the characters in a string using a drawing procedure defined in input

Path construction operators are the only way to modify the current path to be added to the graphic state that describes the shapes using the parameters reported in the CTM. They do not place marks on the page (a job of the painting operators).

Painting operators draw on output, by referring to the information contained in the graphic state, graphic elements (shapes, lines and raster images) that have been placed in the current path and handle the sampled images. The clipping path contained in the graphic state limits the region of the page that is affected by the painting operators, the only that will be displayed in output. Everything that can be drawn in PS on a device goes through a graphic procedure (a list is provided in Table 2.14).

Glyph and Font operators draw *glyph* characters, also using path and painting operators.

Fig. 2.5 Ordering of samples' scanning in PS images

h	(h-1)w	(h-1)w+1		hw-1
h-1	⋮	⋮		⋮
2	w	w+1		2w-1
1	0	1		w-1
0	1	2	w-1	w

Device setup operators define an association between raster memory and physical output on which the image is to be displayed.

Output operators, having completed the image description, transmit the page to the output.

Operators exploit both implicit (among which the current **path**, **color**, **line width** and **font**) and explicit parameters. Parameters are always read-only to avoid errors of inconsistency due to modifications not updated. Some have biases on the type or range their values must belong to. Numeric values are always stored as real numbers, and are forced to fall in the required ranges, independently of the initial specifications. Lastly, the current path, clipping path and the device parameters are objects internal to the graphic state, and for this reason cannot be accessed by programs.

Raster images are regarded as rectangles of $h \times w$ units in a Cartesian reference system having its origin in the bottom-left vertex. They are represented as *sampled images*, i.e., as sequences of sampled arrays, where the samples contain color-related information and the arrays are obtained by linewise scanning the image from the origin until the top-right point, as represented in Fig. 2.5.

To represent an image, some (inter-related) parameters must be specified, that will be implicitly or explicitly used by the operators:

- The format of the source image: width (number of columns) and height (number of rows), number of components per sample and number of bits per component.
- The capacity in bits of the source image, given by the product

$$height \times width \times components \times bits/component.$$

- The correspondence between user space coordinates and external coordinate system, to define the region in which the image will be inserted.
- A mapping between the values of the source image components and the respective values in the current color space.
- The stream of data that make up the image samples.

Thanks to the independence from the output devices on which the documents will be displayed, properties such as resolution, scan ordering of samples, orientation of image and others are to be intended as referred to the images and not to the

devices, although the actual resolution that can be obtained obviously depends on the properties of the latter.

Three *levels* of PS exist, having increasing expressive power. As to image handling, the difference can be summarized as follows:

1. Supports almost only images defined in a DeviceGray color space (graylevel). It is rendered on output using `image/5` that processes the image data coming only from specific procedures and not directly from files or strings. The number of bits per component ranges only between 1 and 8. Some level 1 interpreters handle images with three or four components per value, by using `colorimage`.
2. Adds to the features of level 1 `image/1` (whose parameter is an image dictionary in which much more information can be inserted to define more precisely the image features) the possibility of using 12 bits per component and using files and strings as image data sources.
3. Adds to `imagemask` of previous levels two more operators for color masking.

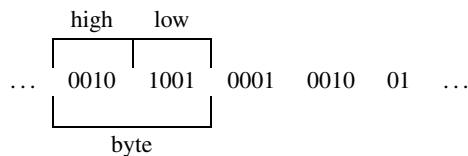
The number of components per color varies according to the Device Space in use (1 component for DeviceGray, 3 for DeviceRGB, etc.) that, in turn, depends on the operator used:

- `image/5` exploits only DeviceGray;
- `colorimage` refers to the value taken by the **ncomp** parameter: DeviceGray if it is 1, DeviceRGB if it is 3, and DeviceCMYK if it is 4;
- `image/1` refers the current color space.

In turn, each component is made up of $n \in \{1, 2, 4, 8, 12\}$ bits that can represent 2^n (i.e., from 2 to 4096) values interpreted as integers in the range $[0, 2^n - 1]$. Image data (*samples*) are stored as byte streams that are split in units, starting from the most significant bit, based on the number of bits/component, so that each unit encodes a value for a color component. The encoded values are interpreted in two ways to obtain the corresponding color:

- `colorimage` and `image/5` map the integer range $[0, 2^n - 1]$ onto $[0.0, 1.0]$;
- `image/1` uses the **Decode** parameter of the corresponding dictionary.

Halftoning techniques are used to approximate color values of components. The stream length must be a multiple of 8 bits (if it is not, padding bits are introduced, that will be ignored by the interpreter during the decoding phase). The bytes can be interpreted as 8-bit integers, starting from the high order bit:



The byte stream is passed to the interpreter by means of files, strings or procedures (depending on the level). Its organization may vary when many components per sample are provided: for a single data source, the component values (e.g., red/green/blue for the DeviceRGB color space) are adjacent for each sample; in

a multiple data source, conversely, there are first all values for the red component, followed by all values of the green one and lastly by the blue ones, and components can be distributed over different sources.

The CTM to modify the coordinate system is provided as a separate operator in the case of `image/5`, or, in the case of `image/1`, as a dictionary that contains the parameters needed to render the image on the page. It allows using any color space, defining an encoding for the sample values different from the standard one, inserting an interpolation among samples and making explicit particular values to mask a color.

Different kinds of dictionaries can be chosen by specifying the **ImageType** parameter inside the dictionary itself. If the value is 1, an image will be represented in an opaque rectangle; if it is 3 or 4 (valid in level 3 language), different levels of color masking can be used, so that new marks added to the page do not to completely overwrite those previously placed in the same area of the page. The **Decode** parameter defines a linear mapping of the integer values of color components in order to exploit them as parameters of `setcolor` in the color space. It is an array of pairs that represent the minimum and maximum value for such a mapping, as shown in Table 2.15. The output value is

$$c = D_{\min} + \left(i \cdot \frac{D_{\max} - D_{\min}}{2^n - 1} \right),$$

where $n = \text{BitsPerComponent}$, i is the input value and D_{\min} , D_{\max} are the values in the array. It yields $c = D_{\min}$ for $i = 0$, $c = D_{\max}$ for $i = 2^n - 1$ and intermediate values among these extremes for the other values of i .

Example 2.8 An excerpt of code referred to an image using type 1 dictionary:

<code>/DeviceRGB setcolorspace</code>	color space identification
<code>45 140 translate</code>	place the image in the bottom-left corner
<code>132 132 scale</code>	scale the image to a 132×132 unit square
<code><<</code>	beginning of the dictionary
<code> /ImageType 1</code>	dictionary identification
<code> /Width 256</code>	width
<code> /Height 256</code>	height
<code> /BitsPerComponent 8</code>	bits per component
<code> /Decode [0 1 0 1 0 1]</code>	array for color coding
<code> /ImageMatrix [256 0 0 -256 0 256]</code>	CTM array
<code> /DataSource /ASCIISHexDecode filter</code>	source data obtained by a filter in hexadecimal values
<code>>></code>	end of dictionary
<code>image</code>	
<code>...</code>	stream of hexadecimal values representing the $256 \times 256 = 65536$ image samples
<code>></code>	mark for the end of source data

Table 2.15 Configuration of the decode array according to the most frequent color spaces to be mapped

Color space	Decode array
DeviceGray	[0 1]
DeviceRGB	[0 1 0 1 0 1]
DeviceCMYK	[0 1 0 1 0 1 0 1]
CIEBasedABC	[0 1 0 1 0 1]
DeviceN	[0 1 0 1 ... 0 1] with N pairs of [0 1]

Table 2.16 Type 1 dictionary parameters for PostScript

Parameter	Type	Optional	Value
ImageType	integer	No	1 (the dictionary, and hence image, type)
Width	integer	No	Width in samples
Height	integer	No	Height in samples
ImageMatrix	array	No	Six numbers that define the transformation from user space to image space
MultipleDataSources	boolean	Yes	If true the samples are provided through different sources, otherwise (default) are packed in the same data stream
DataSource	(various)	No	The source from which the data stream is to be drawn. If MultipleDataSources is true, it is an array of as many components as in the color space, otherwise it is a single file, procedure or string
BitsPerComponent	integer	No	Number of bits used to represent each color component (1, 2, 4, 8 or 12)
Decode	array	No	Values describe how the image sample is to be mapped in the range of values of the proper color space
Interpolate	boolean	Yes	Denotes the presence of interpolation in the image visualization

Type 3 dictionaries allow exploiting an *explicit mask*, i.e., specifying an area of the image to be masked (in such a case, it is not required that the image and the mask have the same resolution, but they must have the same position in the page). It uses two more sub-dictionaries, the *image data dictionary* (**DataDict**) and the *mask dictionary* (**MaskDict**), that are similar to a Type 1 dictionary, except for some restrictions applied to the parameters. The parameters for Type 1 dictionaries are reported in Table 2.16, while those for Type 3 dictionaries can be found in Table 2.17.

A Type 4 dictionary, instead of an area to be masked, identifies a range of colors to be used as a mask (a technique called *color key masking*) that are not displayed. It does not use sub-dictionaries, and differs from Type 1 because of the presence of pa-

Table 2.17 Type 3 dictionary parameters for PostScript

Parameter	Type	OptionalValue	
ImageType	integer	No	3 (code that identifies the dictionary type)
DataDict	dictionary	No	Reference to a type 1 dictionary modified to contain information on the image
MaskDict	dictionary	No	Reference to a type 1 dictionary modified to contain masking information
InterleaveType	integer	No	Code that identifies the kind of organization of the image and of the mask. Possible values: 1. Image and mask samples interleaved by sample and contained in the same data source 2. Image and mask samples interleaved by row and contained in the same data source 3. Image and mask contained in different data sources

Table 2.18 Type 4 dictionary parameters for PostScript

Parameter	Type	Optional	Value
ImageType	Integer	No	4 (code identifying the dictionary type)
MaskColor	array	No	Integers that specify the colors to be masked; its size varies from n to $2n$ values, where n is the number of components per color
Width	integer	No	Width of the image in samples
Height	integer	No	Height of the image in samples
ImageMatrix	array	No	6 numbers that define the transformation from user space to image space
MultipleDataSources	boolean	Yes	Specifies whether the source is single or multiple
DataSource	(various)	No	The source from which the data stream is to be drawn. If MultipleDataSources is true, it is an array with as many components as in the color space, otherwise it is a single file, procedure or string
BitsPerComponent	integer	No	Number of bits used to represent each color component. Allowed values are 1, 2, 4, 8 and 12
Decode	array	No	Values that describe how the sample image is to be mapped in the range of values of the proper color space
Interpolate	boolean	Yes	Presence of interpolation in the image visualization

parameter **MaskColor** that denotes the range of colors to be masked. The parameters for Type 4 dictionaries are reported in [Table 2.18](#).

PDF (Portable Document Format) *PDF* [14] is an evolution of PS developed by Adobe, and in some respects represents a slang thereof. However, differently from PS, it is a document presentation format rather than a programming language. This means that there is no need for an interpreter to be able to write and display documents, but it is sufficient to read the descriptions included in the PDF file itself. Its specifications are public domain; it is compatible to any printer, flexible (it allows character replacement and the insertion of links, bookmarks and notes) and readable in third-party applications through suitable plug-ins. Differently from PS, text displayed on screen is internally represented using character codes, and hence can be copied and pasted in other applications. Thanks also to this, the size of a PDF document is much smaller than the corresponding PS counterpart (often about 1/10th of it).

PDF provides for color space components that use 1, 2, 4, 8 or 16 bits (i.e., 2, 4, 16, 256 or 65536 values, respectively), which is another difference with respect to PS. To represent a sampled image, it is necessary to provide in the dictionary of operator `Xobject` the parameters height, width, number of bits per component and color space (from which the number of components needed is inferred). Color components are interleaved by sample (e.g., in the case of the DeviceRGB color space, the three components—red, green and blue—of a sample are followed by the three components of the next sample in the same order, and so on).

Each image has its own coordinate system, called *image space*, similar to that of PS (the image is split in a grid of $h \times w$ samples, whose Cartesian system has the x axis oriented towards the right and the y axis up), but with the origin in the top-left corner. The scan of samples in the grid starts from the origin and goes on horizontally, linewise. The correspondence between user space and image space is fixed and can be described by the matrix

$$\begin{bmatrix} \frac{1}{w} & 0 & 0 & -\frac{1}{h} & 0 & 1 \end{bmatrix},$$

where the user space coordinate $(0, 0)$ corresponds to the image space coordinate $(0, h)$. The code to modify the transformation matrix is enclosed between the `q` and `Q` commands, that save and restore the image graphic state, respectively; command `cm` modifies the matrix; command `Do` draws the image on the device.

Example 2.9 Sample code to draw an image ‘sampleimage’:

<pre>q 1 0 0 1 100 200 cm 0.707 0.707 -0.707 0.707 0 0 cm 150 0 0 80 0 0 cm /sampleimage Do Q</pre>	<pre>saves the graphic state translation rotation scaling image drawing graphic state restoration</pre>
---	---

PDF provides two methods of different expressive power to describe and handle images:

XObject is used for any kind of image. In Xobject images, the data stream is made up of two parts: a dictionary (*Image Dictionary*) containing image-related information and a data container that encloses the image source data. The dictionary contains approximately 20 parameters, by which all the features needed to display the image on a device can be handled. Many parameters are inter-related, and their values must be consistent not to raise an error. Different operators could handle it in different ways: Table 2.19 reports some of the main parameters, in the form suitable for use by the Do operator.

Example 2.10 Sample code for an Xobject image:

```
...
22 0 obj
<< /Type /XObject
  /SubType /Image
  /Width 256
  /Height 256
  /ColorSpace /DeviceGray
  /BitPerComponent 8
  /Length 83183
>>
stream
aclkjlkj5kjlciIÃ ixinxosaxjhaf ... source data stream (65536 samples)
endstream
endobj
...
```

Inline allows handling only small-sized images, and applies restrictions to their properties. In an Inline object, all information needed to describe the image is embedded in a single data stream. It allows including, before the image source data stream, a short definition of the way in which the stream is to be interpreted (rather than creating an object, such as a dictionary, that embeds it). The object structure is defined by delimitation operators in the following form:

BI	beginning of the Inline object
...	parameters definition
ID	beginning of the source data stream
...	image source data stream
EI	end of the Inline object

Nested BIs and EIs are allowed, while ID must appear only between BI and EI, with at most a blank after it so that the first following character is the first character of the stream. The parameters that can be used between BI and EI are almost all the same as for XObject, and show different abbreviations and syntax, as reported in Table 2.20.

Table 2.19 Some PDF image dictionary parameters, in the form required by Do

Parameter	Type	Optional	Value
Type	name	Yes	Type of object this dictionary refers to (in this case, XObject)
Subtype	name	No	Type of XObject described by this dictionary (in this case, Image)
Width	integer	No	Width in samples
Height	integer	No	Height in samples
ColorSpace	name or array	No	Type of color space used
BitsPerComponent	integer	No	Number of bits used per color component
Intent	name	Yes	A color name to be used for the image rendering intent
ImageMask	boolean	Yes	Indicates whether the image is a mask. If it is true, BitsPerComponent must be equal to 1 and Mask and Colorspace must not appear
Mask	stream or array	Yes	If it is a stream, the mask to be applied to the image; if it is an array, the range of colors to be masked
Decode	array	Yes	Numbers that describe (using the same formula as in PS) how to map the image samples in the range of values suitable for the defined color space
Interpolate	boolean	Yes	Presence of interpolation
Alternates	array	Yes	Alternate dictionaries that can be used for the image
Name	name	No	Name by which the image to which the dictionary refers is referenced in the subdictionary XObject
Metadata	stream	Yes	A data stream that can be used to include further textual information
OC	dictionary	Yes	A further group of dictionaries in case additional information is needed to describe the image

Example 2.11 Sample code for an Inline image:

q	save graphic state
BI	beginning of Inline object
/W 256	width in samples
/H 256	height in samples
/CS /RGB	color space
/BPC 8	bits per component
/F [/A85 /LZW]	filters
ID	beginning of data stream
...slckiu7jncso8nlssjo98ciks ...	data stream
EI	end of Inline object
Q	restore graphic state

Table 2.20 PDF abbreviations synopsis

BI–EI operators parameters		Color spaces	
Internal name	Abbreviation	Internal name	Abbreviation
BitPerComponent	BPC	DeviceGray	G
ColorSpace	CS	DeviceRGB	RGB
Decode	D	DeviceCMYK	CMYK
DecodeParms	DP	Indexed	I
Filter	F	ASCIISHexDecode	AHx
Height	H	ASCIIS5Decode	A85
ImageMask	IM	LZWDecode	LZW
Intent	none	FlateDecode	Fl
Interpolate	I	RunLengthDecode	RL
Width	W	CCITTFaxDecode	CCF
		DCTDecode	DCT

A further way for describing an image in PDF is the **XObject form**. It is a stream that contains a sequence of graphic objects (such as paths, text and images) and can be considered as a model to be exploited several times in different pages of the same document. In addition to document description optimization, it helps in:

- Modeling pages (e.g., forms for bureaucratic documents);
- Describing logos to be displayed on each page background by using a mask to make them semi-transparent;
- Creating a kind of form called **group XObject**, useful to group different graphic elements and handle them as a single object;
- Creating a kind of form called **reference XObject** that can be used to transfer content from a PDF document to another.

2.4 Content-Oriented Formats

By the attribute ‘content-oriented’ we refer to formats that, when defining the document components, specify a set of conceptual properties thereof, instead of determining their position in the document. Thus, the visual aspect that the document gets after displaying is, in some sense, a consequence of its components’ properties. Such formats are usually described using declarative languages that allow specifying ‘what’ is to be obtained, rather than ‘how’ to obtain it (differently from procedural languages).

Several well-known formats belong to this category. Some that will not be thoroughly discussed in this book are:

L^AT_EX is used to specify the properties of the elements that make up a document from a typographic perspective [21], and is based on the **T_EX** language by

D. Knuth. This format requires a compiler to produce the actual document (typically, final output formats of the compiled version are PS and PDF), but ensures that the rendering is perfectly the same on any hardware/software platform. Different typographic styles can be defined, and the compiler acts as a typographer that, given a plain text manuscript annotated with indications on the role and desired appearance of the various passages, typesets it in order to obtain the most balanced layout possible according to the indicated style.

DOC is a binary, proprietary format developed by Microsoft for files produced by the Word word processing program of the Office suite [11]. Latest versions of such a suite exploit a new XML-based, open but patented format called **OOXML** (*Office Open XML*) that since August 15, 2008 has become an ISO standard (ISO/IEC DIS 29500), notwithstanding an open and free ISO standard for the same purposes (the OpenDocument Format, discussed below) already existed since May 1, 2006. In particular, for what concerns word processing, DOC has been replaced by **DOCX**. **RTF** (*Rich Text Format*) was born for allowing document interchange between different word processors. It was introduced by Microsoft with the aim of creating a standard for formatted text. It provides the same potential as the DOC format, but its specification is—at least in its original version—public domain. Most word processing programs can read and write in this format, but many of them add proprietary extensions, which results in limited portability.

2.4.1 Tag-Based Formats

A comfortable way to specify the properties of a document's components is using *tags* spread along the document itself, that express information concerning the document structure and content within the document itself by means of particular attributes of the components to which they are associated. In general, a very important distinction is made between two kinds of tags:

Logical tags that express the role played by the indicated component in the document;

Physical tags that directly specify the way in which the component is to be displayed.

Tags of the former kind do not specify any particular visual attribute for the component they denote: it is in charge of the interpreter properly and consistently displaying it, and different interpreters, or even the same interpreter at different times, could adopt different display attributes for a given logical tag. On the other hand, physical tags do not carry any information about the component role. Usually, the former are to be preferred to the latter, at least for a systematic identification of components because this would allow transposing the same document through different styles by just changing the interpreter and/or the attributes underlying the tag. Conversely, the latter can be used occasionally for describing how specific components are to be rendered, independently of the particular style in use.

The category of *tag-based* formats essentially includes two well-known Web-oriented languages: HTML and XML. Thousands of pages and many professional works have been written about these formats, and hence it would be pretentious (and out of our scope) to give here a complete presentation thereof. Nevertheless, a quick overview is useful to introduce their approach and compare it to those of the other formats presented in this book.

HTML (HyperText Markup Language) A *hypertext* is a set of texts interconnected by links that allow jumping from a given place in a document to (a specific place of) another document. The *World Wide Web* (WWW) is the most widespread and famous example of hypertext nowadays. *HTML* [3] is a tag-based language developed to support the creation and presentation of hypertextual information (such as Web documents) in a universal and hardware-independent way, and to allow the definition of a simple and straightforward interface for ‘browsing’ among hypermedia documents, with the only help of a pointing device (e.g., a mouse). It is an application of the *SGML* (*Structured Generalized Markup Language*, used in the editorial field), and represents the current standard for Web page description, whose definition is fixed by the W3C.¹²

HTML allows defining the format of a document (size, type and style of the characters, kind of text justification, etc.) and including hyperlinks to other documents or to multimedia objects (such as images, icons, videos, sounds) or even to other Internet services (FTP, Gopher, etc.), still keeping the (sometimes complex) commands underlying those links transparent to the user. Its main strengths lie in the possibility of including images in the documents and in its universality, flexibility, richness and compactness.

An HTML file is encoded in standard ASCII, that can be easily handled by any text editor (although dedicated editors improve readability). Special characters, reserved to HTML (<, >, &) or not provided by the ASCII, are expressed as *escape sequences* (sequences of characters that are displayed in the document as a single symbol), starting with an ampersand & and ending with an (optional) semi-colon ;. Their conversion is the task of the interpreter. A sample of escape sequences is reported in Table 2.21.

The visual presentation of the document is driven by structural and stylistic indications provided by specifying content- and formatting-related *directives* to an interpreter. Directives are interleaved with the text that represents the actual content, delimited by brackets (< and >) to be distinguished from it, and are made up of a code (called *tag*, a term sometimes improperly used to denote the whole directive), possibly followed by the specification of *attributes* (some are mandatory, some optional, depending on the particular tag) that modify their effect:

```
<TAG attribute="value" ... attribute="value">
```

¹²Some software applications that produce HTML documents exploit extensions not included in the official format definition, and hence part of their output represents semi-proprietary code that might not be properly displayed on some platforms.

Table 2.21 Some special characters and their escape sequences in HTML

Symbol	Escape	Note	Symbol	Escape	Symbol	Escape
<	<	less than	À	À	à	à
>	>	greater than	É	É	é	é
&	&	ampersand	È	È	è	è
©	©	copyright	Ì	Ì	ì	ì
...			Ò	Ò	ò	ò
			Ù	Ù	ù	ù

Not all tags are supported by all browsers, but unknown ones are simply ignored instead of raising an error. Commands are not case-sensitive (although, to improve human readability, a widespread and useful agreement is to write tags in upper-case and attributes in lower-case). Some tags denote the presence of a single element (e.g., a line break or an image); more often they apply to a piece of content, and hence require a corresponding directive that indicates the end of their scope. In the following, we will call the former *single* tags, and the latter *paired* tags. The closing directive of a paired tag (that can be sometimes omitted, being implicitly assumed by the interpreter when certain other directives are found) is denoted by a slash / before the tag:

```
... <TAG attribute="value" ... > content affected by the tag </TAG> ...
```

Attributes, if any, must be specified in the opening directive only. Comments, ignored by the interpreter, can be included as follows:

```
<!-- text of the comment -->
```

The high-level structure of an HTML document is as follows:

<HTML>	beginning of the document
<HEAD>	beginning of the heading
...	meta-information
</HEAD>	end of the heading
<BODY>	beginning of the actual content
...	actual content
</BODY>	end of the actual content
</HTML>	end of the document

where the tag HTML delimits the document specification, the tag HEAD delimits a heading (that expresses general information on the document that is not to be displayed), and the tag BODY delimits the actual hypertextual content of the document (that is to be displayed). Some possible attributes for the BODY tag are: background (to specify a background image for the page), bgcolor (to specify a background color for the page), etc.

The heading typically contains a document title (paired tag TITLE, in which spacings are significant), useful for identifying it in other contexts, that is usually

displayed in the window title bar. Additional meta-information on the document can be specified using the single tag `META`, in the form:

```
<META name="..." content="...">
```

where the `name` attribute denotes the kind of meta-information (typical values are: **generator**, the software used to produce the HTML code; **author**, the author of the document; **keywords**, a list of keywords related to the document content; **description**, a textual description of the content) and `content` specifies the actual meta-information.

The main elements that can be included in the body are: titles, lists (possibly nested at will), images, sounds, videos and Java programs (*applets*), hyperlinks, character-formatting styles. It should be noted that the interpreter generally ignores carriage returns, and considers tabbings or multiple spaces as a single blank. Thus, new lines and spacings in the source file are exploited just for improving human readability of the code. To prevent the document from appearing as a single paragraph, suitable tags that cause layout formatting must be used, such as text splitting ones:

- Line breaks (*Break Rule*), `BR`;
- Horizontal line separators (*Hard Rule*), `HR`;
- Paragraph boundaries (using the paired tag `P` or also the single directive `<P>` that displays a vertical space between the content before and after it).

The only exception is the *preformatted* text, delimited by the paired tag `PRE`, that is displayed with fixed size characters, in which spaces, new lines and tabbings are significant (it is useful for displaying program listings). However, HTML tags, hypertextual references and links to other documents can still be used in its scope.

Images are enclosed using the directive

```
<IMG src="filename">
```

where the attribute `src` specifies the name and path of the image file. Supported formats are *XBM* (*X BitMap*), GIF, JPEG. Vertical alignment of the image with respect to the surrounding text can be specified using the `valign` attribute (allowed values are **top** = upper border, **middle** = centered, **bottom** = lower border, that is the default). Using the optional `height` and `width` attributes, one or both display dimensions of the image can be independently resized (if just one is specified, the other is consequently set so to preserve the original ratio), in pixels or as a percentage of the browser window (by specifying the % symbol after the value). The (optional) attribute `alt` allows specifying an *alternate* textual description of the image, to be displayed when the mouse passes over it or by textual browsers that cannot display images.

HTML allows applying special character-formatting styles to pieces of text. The ‘physical style’ is defined by the user and shown by the browser, while ‘logical styles’ are configured by the interpreter. Both exploit paired tags. A scheme of the available tags for physical styles, along with the typical corresponding logical styles (and tags), available in HTML is provided in Table 2.22.

Table 2.22 Logical and physical character-formatting style tags in HTML

Physical style	Italics	Bold	Underlined	TeleType-like
Tag	I	B	U	TT (fixed-size font characters)
Corresponding Logical styles	EM <i>emphasize</i>	STRONG <i>note</i>		CODE for program <i>code</i>
	CITE <i>citation</i>			SAMP <i>sample</i> (for examples)
	VAR <i>variable</i>			KBD <i>keyboard</i> (for names of keys)
	DFN <i>definition</i>			

Section headings are identified by the family of tags `Hn`, where *n* specifies the depth level (and hence the character size) in a 1 to 6 scale. Level 1 headings are the most important, usually displayed with larger characters in bold face. In complete documents, they usually express the document title, while in documents that represent sections of wider works (e.g., chapters of a book), they should be referred to the content of that particular section, using the `TITLE` tag in the heading to provide the structural reference (e.g., the title of the whole book plus that of the chapter).

As to lists, the following kinds are supported (each followed by the associated paired tag):

- Unordered List** `UL`
- Ordered List** `OL`
- Description List** `DL`

Elements of ordered or unordered lists are identified by the paired `LI` tag (*list item*), while in description lists the title of the item is identified by the paired tag `DT` (*Description Title*) and its description by the paired tag `DD` (*Description Description*). All item tags are allowed also as single tags. A paragraph separator between list elements is not needed, but each item may contain multiple paragraphs, lists or other descriptive information. Hence, an arbitrary nesting can be specified. Unordered lists are displayed differently by different browsers, as regards indentation and symbols for each level (typically a dot for the first level and a square for the second).

Tables are specified row-wise, according to the following structure:

```
<TABLE>
  <TR> ... </TR>
  ...
  <TR> ... </TR>
</TABLE>
```

where possible attributes of the `TABLE` tag are `border` (thickness of the line to draw the table grid), `cellspacing` (internal margin for the cell content), `cellpadding` (spacing among cells), `width`. Each *table row* tag `TR` delimits the single cells in a row, specified using the paired tags `TH` (for *heading* ones, to be highlighted) or `TD` (for content, or *description*, ones). `TR`, `TH` and `TD` can specify the horizontal or vertical alignment of the cell content using the attributes `align` (values **left**, **right**, **center**) and `valign` (**top**, **middle**, **bottom**), respectively. `TH` and `TD` additionally provide attributes `nowrap` (automatic wrapping of the contained

text), `rowspan` and `colspan` (to span a cell over several rows or columns, respectively). The paired tag `CAPTION` allows specifying an explanation, with attribute `align` (possible values are **top**, **bottom**, **left**, **right**).

Hypertextual links are regions of a document that, when selected, take the user to another document that can be on the same computer or on a different one. They typically appear highlighted (usually colored and underlined, but this setting can be changed), and are specified by delimiting the region with an *anchor* paired tag

```
<A href="..."> ... </A>
```

where the `href` attribute represents the (relative or absolute) URL of the linked document. It is also possible to specify anchors to specific document sections, useful for moving directly to intermediate passages. The target place (*named anchor*) must be labeled using the (paired or single) directive

```
<A name="reference">
```

that defines an identifier for it, to be exploited as a suffix for the URL in the links, as follows:

```
<A href="filename.html#reference"> ... </A>
```

(the file name being optional if the target document is the same as the starting one).

The paired tag `ADDRESS` specifies the author of a document and a way to contact him (usually an e-mail address). It is typically the last directive in the file, whose content is placed on a new line, left-justified.

Colors are expressed in RGB format, as a sequence `#RRGGBB` of pairs of hexadecimal digits for each basic color (RR = red, GG = green, BB = blue), or using symbolic names for some predefined colors: **black** (`#000000`), **silver** (`#C0C0C0`), **gray** (`#808080`), **white** (`#FFFFFF`), **maroon** (`#800000`), **green** (`#008000`), **lime** (`#00FF00`), **olive** (`#808000`), **yellow** (`#FFFF00`), **navy** (`#000080`).

An interesting feature is the possibility of splitting a single window into *frames*, a sort of tables each of whose cells can contain a separate document. This is obtained by the paired tag `FRAMESET`, with attributes `rows` and `cols` to specify the window splitting as a comma-separated list of values (in pixels or, if followed by %, in percentage of the window size; a * denotes all the available space). Other allowed attributes are: `src` (the document to be displayed in the cell), `name` (an identifier for referencing the frame), `scrolling` (with values **yes**, **no** and **auto**, to allow scrolling of the content of a cell), `noresize` (to prevent window resizing from the user), `marginwidth` and `marginheight` (to define the spacing among cells). A `NOFRAME` tag can be used to specify what to display in case the browser does not support frames.

Example 2.12 A sample specification of frames in an HTML document.

```
<FRAMESET rows="25%, 75%">
  <FRAME noresize name="top" src="t.html" scrolling="no">
  <FRAMESET cols="150,*,150">
    <FRAME name="left" src="l.html">
```

```

<FRAME name="center" src="c.html" scrolling="yes">
<FRAME name="right" src="r.html" scrolling="auto">
</FRAMESET>
</FRAMESET>

```

The window is vertically divided in two rows, taking 1/4 (25%) and 3/4 (75%) of the whole allowed height, respectively. The first row contains a frame named ‘top’, in which the source HTML is loaded from file ‘t.html’, and displayed so that scrolling is not permitted (scrolling widgets are not shown), even in the case the source is too large to fit the available frame space. The second row, in turn, is horizontally split into three frames (columns), named ‘left’, ‘center’ and ‘right’, respectively, of which the first and the last one take exactly 150 pixels, while the middle one takes all the remaining space (depending on the overall window width). These columns contain the hypertexts whose sources are to be loaded from files ‘l.html’, ‘c.html’ and ‘r.html’, respectively, of which the second one always shows scrolling widgets, while the last one shows them only if the source content exceeds the available frame space.

XML (eXtensible Markup Language) Developed by W3C for use in the Internet, in order to make more flexible and personalizable HTML, *XML* subsequently established also as the main language for information representation. It was designed for ease of implementation and for interoperability with both SGML and HTML [9, 13]. Indeed, it provides a shared syntax that can be understood by machines, allows homogeneous processing and enforces information storing, handling and interchange. It is acceptable by human users as well, since it provides the generic blocks for building a language, according to a comfortable tag-based syntax (just like HTML). It is a subset of SGML,¹³ focused on documents and data, in which no keyword is pre-defined (as in HTML), but a set of rules is provided according to which other languages (e.g., HTML) can be written. Of course, this leaves the user with the problem of choosing the tags and, if needed, with defining their intended meaning and behavior.

The structure of an XML document requires, in the first row of the file, a declaration concerning the language version (mandatory) and the character encoding used (UTF-8 by default):

```
<?xml version="..." encoding="..."?>
```

followed by the actual content.

The content is made up of *elements*, each delimited by directives in the same formalism as in HTML, but denoted using user-defined tags. Elements (and hence directives) can be nested, but a *root* element that contains all the others and defines the type of XML document is mandatory. The nesting order is relevant, and represents a good indicator for the choice of elements or attributes. Directives can even be empty (<tag></tag>, often abbreviated in <tag />).

¹³Note that HTML is an *application*, not a *subset*, of SGML.

As in HTML, tag effects can be modified by using *attributes* whose values are delimited by single or double quotes. Attribute values cannot be empty, and only text is allowed therein. They are often exploited for specifying metadata that express properties of the elements, and their ordering is not significant to XML (although it can be relevant to the applications that will process the file).

An *entity* is any unit of content, started by `&` and ended by `;` (like escape sequences in HTML). Some entities are pre-defined (*escape sequences*):

- `lt` (`<`),
- `gt` (`>`),
- `apos` (`'`),
- `quot` (`"`),
- `amp` (`&`);

others are used as shortcuts for human convenience. References to characters are represented by UNICODE code points *n*, expressed in decimal (`&#n;`) or hexadecimal (`&#xn;`) notation.

Comments are as in HTML:

```
<!-- text of the comment -->
```

and cannot contain the `--` sequence of characters. They are exclusively for human use (differently from HTML, where instructions for applications can be enclosed), because other specific tools are provided to contain information intended for machines. For this reason, the use of comments must be avoided in machine-intended documents.

As to syntax, tag and attribute names are case sensitive, must start with a letter or an underscore and may continue with any mix of letters, digits, underscores, dots (not colons, that are reserved for namespaces, to be discussed below). Some names (those starting with `xml`) are reserved. Well-formed documents require that elements are correctly nested and that all open tags are closed. Checking the validity of an XML document consists in a check for admissibility of well-formed expressions, and is useful for machine-readability purposes.

Namespaces are used for disambiguation of identical names used in different contexts, according to the concept of *URI* (*Uniform Resource Identifier*). A namespace is denoted by a name, and defines a *vocabulary* whose elements are referred to a specific context and are independent from those in other namespaces. An XML document can specify which namespaces are going to be exploited: then, elements and attributes having the same name but coming from different contexts can be used together in that document by prefixing each of them with the intended namespace, separated by a colon: `namespace:element` and `namespace:attribute`, respectively. Namespaces are themselves URIs of the vocabulary `xmlns:namespace`.

Example 2.13 A short XML document:

```
<?xml version="1.0" encoding="us-ascii"?>
<countries>
```

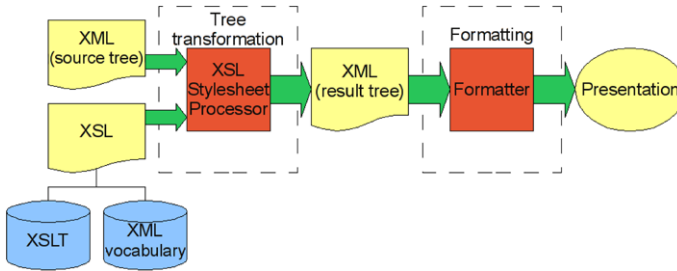


Fig. 2.6 XSL transformation of an XML document

```

<country id="c01">
  <name>Italy</name>
  <abbr>ITA</abbr>
</country>
<country id="c02">
  <name>United States of America</name>
  <abbr>USA</abbr>
</country>
<country id="boh"/>
</countries>

```

In addition to the abstract syntactic correctness, some kind of semantic correctness may be required to the used tags and attributes and to their nesting. The types of elements and attributes allowed in an XML document can be expressed by means of *DTD (Document Type Definition)* files. An XML document that must fulfill a DTD can reference it by means of a `<!DOCTYPE . . .>` declaration placed in the second line of the file. It indicates which is the root element of the XML document (that can even be different from the root of the DTD). If a DTD is specified, the elements in the XML document must be instances of the types expressed in the DTD. Specific procedures are in charge of processing the document with respect to its declared DTD, and of rejecting invalid documents.

XSL (eXtensible Stylesheet Language) is a language that allows specifying styles for displaying XML documents [10]. In this way, the same XML file can be presented in different ways: as a Web page, a printable page, speech, etc. The transformation takes place in two steps, as depicted in Fig. 2.6:

1. **Tree transformation**, carried out on an XML document (that represents the *source tree*) and an XSL document by an *XSL Stylesheet processor* that produces a *result tree*;
2. **Formatting**, carried out on the result tree by a *formatter* that produces the final presentation.

The XSL exploits an *XSLT (XSL Transformations)*, defined by the W3C [5]) language for transforming XML documents and an XML vocabulary for specifying formatting semantics. The former describes how the document is transformed into another XML document that uses the latter. The result tree is a new representation

of the XML source that can be used by the formatter for producing the final presentation.

XML processing tools are parsers that take an XML file as input and return information related to them. They fall in two categories: *event-based* and the *DOM* (*Document Object Model*—see Sect. 5.1.2). The former act by means of calls to APIs (program functions) any time a parsing is involved, or by means of a standard API library called SAX (available in Java, Python, Perl). The latter, whose standard is defined by the W3C, load the XML document content into main memory in the form of a tree. Three levels of DOM exist, each including several specifications for different aspects thereof. XSLT processors define transformations of an XML document into another XML or HTML document. For instance, *Xpath* [4] is a very flexible query language for addressing parts of an XML document, designed to be used by XSLT but used also to run processing functions.

2.4.2 Office Formats

The most straightforward and classical way for producing a natively digital document is using a Word Processor, usually included in a suite of office-related programs. Several software packages of this kind have been developed over the years by various software houses. Up to recently, each of them used to save the documents in a different proprietary format, often binary. This might not be a problem while the very same version of the application is used, but becomes a strong limitation as soon as one needs to manipulate the document content, directly or using another program, in order to carry out operations that are not provided by the original application used for producing them. The only way out in these cases is to pass through intermediate, neutral or ‘universal’ formats, such as RTF or CSV (*Comma-Separated Values*). However, just because of them being neutral, they are often not able to preserve all the original settings of the document. For this reason, many Public Administrations have recently required their applications to adopt open formats and save the documents as pure text (typically XML), so that the content can be accessed independently of the visualization application.

Today only two office application suites are widely exploited and compete on the market: the Microsoft Office and OpenOffice.org suites. While the former has a long tradition, the latter is much more recent, but has gained large consensus in the community thanks to its being free, open-source, portable, rapidly improving its features and performance, and based on an open, free and ISO-standardized format. Conversely, the former is commercial and based on proprietary formats that only recently have been accepted as an ISO standard. For these reasons, here we will focus on the latter.

ODF (OpenDocument Format) The *OpenDocument* format (*ODF*) [12, 18] was developed for the office applications suite OpenOffice.org whose aim is to provide users with the same applications on any hardware and software platform, and to

make them able to access any data and functionality by means of open formats and components that are based on the XML language and on API technology, respectively. The OASIS (Organization for the Advancement of Structured Information Standards) decided to adopt this format and to further develop and standardize it, resulting in the definition as an ISO standard (ISO 26300) since 2006.

The OpenDocument format is an idealized representation of a document structure, which makes it easily exploitable, adaptable and extensible. An OpenDocument consists of a set of files and directories that contain information on the settings and content of the document itself, saved altogether in a compressed file in ZIP format.¹⁴ A document generated by the OpenOffice.org word processor (*Writer*), identified by the *ODT* acronym (for OpenDocument Text) is made up of the following elements:

- `mimetype` a file containing a single row that reports the MIME content type of the document;
- `content.xml` the actual document content;
- `styles.xml` information on the styles used by the document content (that have been separated for enhanced flexibility);
- `meta.xml` meta-information on the content (such as author, modification date, etc.);
- `settings.html` setting information that is specific to the application (window size, zoom rate, etc.);
- `META-INF/manifest.xml` list of all other files contained in the ZIP archive, needed by the application to be able to identify (and read) the document;
- `Configurations2` a folder;
- `Pictures` a folder that contains all the images that are present in the document (it can be empty or even missing).

It should be noted that the XML code is often placed on a single row to save space by avoiding line break characters. The only mandatory files are ‘content.xml’ and ‘manifest.xml’. In principle, it is possible to create them manually and to properly insert them in a ZIP file, and the resulting document could be read as an OpenDocument (although, of course, it would be seen as pure text and would not report any further style, setting or other kind of information).

References

1. Graphics Interchange Format (sm) specification—version 89a. Tech. rep., CompuServe Inc. (1990)
2. TIFF specification—revision 6.0. Tech. rep., Adobe Systems Incorporated (1992)
3. HTML 4.01 specification—W3C recommendation. Tech. rep., W3C (1999)
4. XML Path Language (XPath) 1.0—W3C recommendation. Tech. rep., W3C (1999)

¹⁴This representation is inspired by JAR files (*Java ARchives*), used by the Java programming language to save applications.

5. Transformations, X.S.L.: (XSLT) 1.0—W3C recommendation. Tech. rep., W3C (1999)
6. International standard ISO/IEC 10646: Information technology—Universal Multiple-octet coded Character Set (UCS). Tech. rep., ISO/IEC (2003)
7. Portable Network Graphics (PNG) specification, 2nd edn.—W3C recommendation. Tech. rep., W3C (2003)
8. Lizardtech djvu reference—version 3. Tech. rep., Lizardtech, A Celartem Company (2005)
9. Extensible Markup Language (XML) 1.1, 2nd edn.—W3C recommendation. Tech. rep., W3C (2006)
10. Extensible Stylesheet Language (XSL) 1.1—W3C recommendation. Tech. rep., W3C (2006)
11. Microsoft Office Word 97–2007 binary file format specification [*.doc]. Tech. rep., Microsoft Corporation (2007)
12. Open Document Format for office applications (OpenDocument) v1.1—OASIS standard. Tech. rep., OASIS (2007)
13. Extensible Markup Language (XML) 1.0, 5th edn.—W3C recommendation. Tech. rep., W3C (2008)
14. Adobe Systems Incorporated: PDF Reference—Adobe Portable Document Format Version 1.3, 2nd edn. Addison-Wesley, Reading (2000)
15. International Telegraph and Telephone Consultative Committee (CCITT): Recommendation t.81. Tech. rep., International Telecommunication Union (ITU) (92)
16. Deutsch, P.: Deflate compressed data format specification 1.3. Tech. rep. RFC1951 (1996)
17. Deutsch, P., Gailly, J.L.: Zlib compressed data format specification 3.3. Tech. rep. RFC1950 (1996)
18. Eisenberg, J.: OASIS OpenDocument Essentials—Using OASIS OpenDocument XML. Friends of OpenDocument (2005)
19. Hamilton, E.: JPEG file interchange format—version 1.2. Tech. rep. (1992)
20. Huffman, D.: A method for the construction of minimum-redundancy codes. In: Proceedings of the I.R.E, pp. 1098–1102 (1952)
21. Lamport, L.: \LaTeX , A Document Preparation System—User’s Guide and Reference Manual, 2nd edn. Addison-Wesley, Reading (1994)
22. Reid, G.: Thinking in PostScript. Addison-Wesley, Reading (1990)
23. Shannon, C.E., Weaver, W.: The Mathematical Theory of Communication. University of Illinois Press, Champaign (1949)
24. The Unicode Consortium: The Unicode Standard, Version 5.0, 5th edn. Addison-Wesley, Reading (2006)
25. W3C SVG Working Group: Scalable Vector Graphics (SVG) 1.1 specification. Tech. rep., W3C (2003)
26. Welch, T.: A technique for high-performance data compression. IEEE Computer **17**(6), 8–19 (1984)
27. Wood, L.: Programming the Web: The W3C DOM specification. IEEE Internet Computing **3**(1), 48–54 (1999)
28. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory **23**(3), 337–343 (1977)
29. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24**(5), 530–536 (1978)



<http://www.springer.com/978-0-85729-197-4>

Automatic Digital Document Processing and
Management

Problems, Algorithms and Techniques

Ferilli, S.

2011, XXVI, 297 p., Hardcover

ISBN: 978-0-85729-197-4