Text Formatting (ms 984)

by

Daniel M. Berry

ABSTRACT

Text formatting is described in terms of algorithms that are used and their effects on the appearance of the printed document. Multilingual and multidirectional documents are considered. The interactions of the basic line-by-line formatting and more advanced paragraph formatting algorithms with text size; line length; justification; hyphenation; and figure, table, and footnote placement are considered. Both compiling and WYSIWYG formatters are described in terms of these algorithms and considerations.

Keywords: algorithms, compiling, direct-manipulation, document, figure, footnote, formatter, formatting, hyphenation, justification, markup, multidirectional, multilingual, point size, table, text, WYSIWYG

INTRODUCTION

The *text formatting* problem, to be solved by software residing on a computer, is to take a sequence of words stored in an input file and arrange them in the same order on as many pages as are needed, subject to the user's commands and his choices on a variety of constraints and options. The pages can be printed on paper at any time, and usually the user is able to see on the computer's screen approximately how the pages will appear when printed. The approximation is usually accurate to the resolution of the computer's screen.

The results the user gets depend on the commands he gives and on value of the choice he makes for each constraint or option. These commands are given and choices are made by directly or indirectly (See the later discussion on direct manipulation.) inserting additional characters, often called *markup*, at appropriate points in the input file. Later in the article, the commands are summarized and a full list of constraints and options is given. In the mean time, the constraints and options are collectively referred to as "choices". Telling a formatter to apply a particular option is called *turning the option on* and telling the formatter not to apply a particular option is called *turning the option off*.

For concreteness, the text that you are now reading was formatted in lines that are 3.154 inches (80.1 millimeters) wide, in the Sabon family of typefaces at the 10 point size on lines spaced at 12 points. The text is strictly left to right and is left and right justified with hyphenation turned on. The commands issued were mainly for introducing the title, introducing the author's name, introducing sections, introducing paragraphs, giving numbered and labelled lists, specifying footnotes, and including figures.

Each formatter worth its salt allows the user to give commands and to set the value of all choices. Some do not allow making some of these choices, e.g., one very popular formatter in its academic, not professional version does not allow installing automatic hyphenation.

The formatters that are considered in this article are MS Word [14], FrameMaker [3], TEX with LATEX,[1] ditroff [16, 8], and the Mozilla browser [15]. These are both representative and available, and are widespread in some sense of the word. The online version of this article shows the text of this article formatted by each of these formatters, both with and without hyphenation if possible, in a two-column format that resembles as much as conveniently possible the hard copy you are looking at right now.[2]

[1] LATEX is TEX extended by macro-defined commands that implement a collection of aesthetic designs for a variety of document types used by computer scientists.

In this article, each formatter is classified by the choices it makes for the user and what it allows the user to choose.

This article discusses each choice, i.e., constraint and option, by briefly describing both the algorithmic issues involved and the effect each choice has on the appearance of a formatted document. Many of these choices interact with each other.

The plan for this article is:

1. to define some terms;
2. to describe properties of the text to be formatted;
3. to describe the basic formatting algorithms in terms of their effects; and
4. to present each constraint and option, in terms of its effect on the formatting algorithm and in terms of its affect on the final appearance of a document.

The focus of this article is strictly on the formatting issues. (Other articles in this encyclopedia deal with details of typefaces (also called "fonts"), markup language, and the linguistic aspects of hyphenation.)

TERMS

Terms are defined by their being used in descriptive sentences.

Document:

The text being formatted is called the *document*.

Input, Output, and Output Lines:

This article distinguishes between *input* and *output*. The input is a sequence of characters some of which are space, tab, newline, and endfile. The output is a collection of characters each printed at a particular position on a page. A collection of characters with the same horizontal position forms an *output line*. If we consider the characters of one output line listed in order of their vertical positions, then we can talk about the sequence of characters in the output line.

Character Bounding Box, Glyph, Side Bearings, Width, Boundary, and Adjacency:

See Figure 1 for an output line consisting of the two nonsense words "Hmg Hmg" printed at a very large size. In this figure, various axes, distances, and properties of the characters are identified, and these are the subject of the following discussion.

In a modern printer that accepts PostScript as its page description language, each character has a *bounding box*, an imaginary rectangle that circumscribes the shape of the character, known also as the character's *glyph*. The bounding box is shown explicitly for only the first "g". If the character is not to connect with its neighbors, as is the case with nonscript versions of Latin letters, then to the left and to the right of the bounding box are some regions in which nothing is printed. The region to the left of the bounding box is called the *left side bearing* and the region to the right of the bounding box is called the *right side bearing*. In this article, the left edge of the left side bearing is called the *left boundary* of the character, and

---

[2] The IEEE two-column conference proceedings format was readily available for all the listed formatters except Mozilla. The author is not enough of an expert in all of these formatters to be able to match the encyclopedia's format exactly; so this close match is used instead.

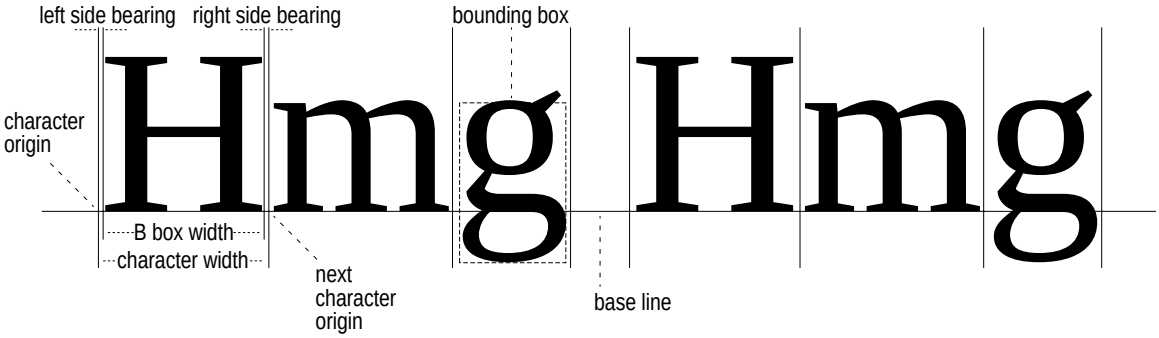

Figure 1: Nonconnected Latin Characters with Axes

the right edge of the right side bearing is called the *right boundary* of the character. The distance from the left boundary to the right boundary of a character is the character's *width*. Characters within one word, such as each "Hmg" in Figure 1, are normally printed with their side boundaries touching each other. Two characters with boundaries touching each other are called *adjacent*.

All the characters on one output line are printed on a single *base line*. The intersection of a character's left boundary and the base line is called the character's *origin*. Thus for a pair of adjacent characters, the origin of the right-hand character coincides with the left-hand character's right boundary.

Figure 2 shows two Arabic characters that connect with each other. For each, the side bearing on the connecting side is zero. As a result, if the connecting pieces of two adjacent connecting characters meet their boundaries over the same range of *y* values, then when the character's boundaries touch, the characters end up connecting.

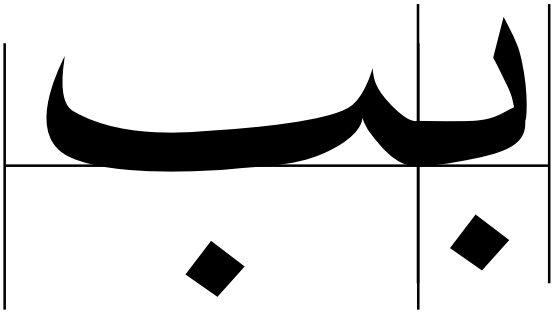

Figure 2: Connected Arabic Characters with Axes

Blank:

A *blank* character in an input line is a space, a tab, a newline, or an endfile character.

Word and Chunk:

A *word* in either kind of line is a sequence of adjacent non-blank characters. These characters are typically mostly letters and may occasionally include, especially at the end, a punctuation character. However, from the point of view of the formatting process, the nature of the characters is somewhat irrelevant. Of course, the notion of hyphenation is relevant only for words consisting only of letters and possibly one punctuation symbol at the end.

In input to a formatter, words are separated by and delimited by blank characters.

When hyphenation is turned on in a formatting, then, occasionally, a word will be broken into two pieces. Appended to the first piece will be a hyphen character.

The word *chunk* is used in this article to refer to a unit of formatting. Each non-hyphenated word, each first piece of a hyphenated word together with the appended hyphen, and each second piece of a hyphenated word is called a *chunk*.

Space Character:

Generally, the characters inside a chunk are printed adjacent to each other. When two consecutive chunks are printed on the same output line, the last character of the first chunk is not adjacent to the first character of the second. In Figure 1, the first "g" at the end of the first "Hmg" is not adjacent to the second "H" at the beginning of the second "Hmg". The region of the output line between two consecutive chunks, in which no character is printed, is called the *space* between the chunks. The minimum size of a space between chunks is usually about the same as that of a narrow character such as "i". This minimum spacing between chunks is often called *the space character* even though nothing is really printed.

More about Output Line:

An output line of a formatted document then consists of consecutive chunks in a sequence with some non-printed space in between them. At most the last chunk of a line is the first piece of a hyphenated word together with its appended hyphen. Generally, almost all lines of a document are the same length; the first and last lines of any paragraph may be shorter than this common length.

Output Page:

An *output page* of a formatted document consists of a sequence of lines. Generally, the number of lines in almost all pages in a document are the same; the first and last pages of a chapter may have fewer lines than others.

Justification and Raggedness:

When all the lines of a page have text right up to a margin, either left or right, then the lines are said to be *justified* to that margin. Conversely, if all the lines of a page do not meet a margin, either left or right, leaving some space between the chunk closest to the margin and the margin, then the lines are said to be *ragged* at that margin. In most documents, e.g., this article, lines are left and right justified. In some Latin-alphabet documents, lines are left justified and right ragged. Very rarely, mostly only in poetry, one finds portions of Latin-alphabet documents whose lines are left ragged and right justified. One does find portions of documents in which all lines are centered, effectively making the lines left and right ragged.

Formatting:

To use the vocabulary just established, *formatting* can be described as the process of arranging the words of an input document into chunks laid out in output lines on output pages according to commands, constraints, and options provided and set by the user.

TEXT CHARACTERISTICS

The text of a document to be formatted can be in a mixture of human languages, each with its own character set (See the article on Unicode). Some languages, e.g. English, French, German, and Russian, are written from left to right with lines flowing from top to bottom. These are called the *LR* languages. Other languages, e.g., Arabic, Hebrew, and Persian, are written from right to left with lines

flowing from top to bottom. These are called the *RL* languages. Still other languages, e.g., Chinese, Japanese, and Korean, are written from top to bottom with lines flowing from right to left. These are called the *TB* languages.

To a formatting algorithm, the specific languages of a document does not matter beyond the direction in which its text is written. Two different languages that are written in the same direction are equivalent from the formatter's point of view, even if they use different alphabets. The different alphabets can be regarded as no more than different typefaces. Therefore, this article talks only about the three directions of text, LR, RL, and TB, and not the actual languages in which the text is written. Of course, the rules for hyphenation do depend on the language of the text.

A document, or more properly, a section of a document, has a *document direction*. For example, a book in English about the Arabic Koran is a left-to-right (LR) document even though it contains quotations in Arabic. A book in Hebrew about Shakespeare is a right-to-left (RL) document even though it contains quotations in English. A book in Chinese about the Hebrew Bible is a top-to-bottom (TB) document even though it contains quotations in Hebrew. The document direction establishes the general flow of lines. It establishes also the side on which various formatting commands are obeyed. For example, if paragraphing means indenting the first line of the paragraph, then in a RL document, that indentation is from the right margin.

Document direction can be changed in the interior of a document. Therefore, there is the notion of the *current document direction at any point*. If a quotation in a language of a different direction is lengthy and is stretching over several paragraphs, it might be convenient to change the document direction for the duration of the quotation, so that the quotation's paragraphs are indented from its correct side.

To be direction independent, this article uses the terms *start* and *end* to designate the sides of a page in which a line starts and ends. Thus, in an LR document, the right margin is called "the end margin", while in an RL document, the right margin is called "the start margin".

Text in which each language is printed in its own direction is said to be in *visual order*. While different languages may be written in different directions, all languages are spoken in the same direction, namely forward. Therefore, for ease in input, the convention is that the characters of a mixed-language document are stored in an input file in the order that they are spoken when a knowledgeable reader is reading aloud the visually-ordered text of the mixed-language document; this reader knows the rules about reading visually-ordered multidirectional text. This stored order of characters is called *time order*. That text is stored in time order means that the characters may be keyed in in the order that they are spoken. It is the job of the formatter to arrange the output so that the text is written in visual order, so that the time ordering is recreated when a person who knows the rules about reading multidirectional text reads the visually-ordered output.

Here is an example of a line containing left-to-right English and right-to-left Arabic and Hebrew text displayed in the visual order that is used in the Middle east:

Dana said, "שלום דניאל ، دانيال سلام" to Daniel.

Since this line is in a left-to-right document (this article), it is read in an overall left-to-right direction. Any embedded right-to-left text is read in its left-to-right turn, but from right to left. Thus, this line is read:

1. Dana said, " [*from left to right*]
2. שלום דניאל ، دانيال سلام [*from right to left*] (*salaam danyal, shalom danyel*)
3. " to Daniel. [*from left to right*]

Here is the same line in time order, giving each character in the order it is pronounced when a

knowledgeable reader reads aloud the visually-ordered line above:

Dana said, “سلام دانا ل، שלום דניאל” to Daniel.

Even if you cannot read the Arabic and Hebrew phrases in between the pairs of English quotation marks, you should be able to see that in the two lines, the directions of their characters are reversed. Note that in the visually-ordered line, some of the Arabic characters connect to each other, but none of the Hebrew and Latin characters connect to any other.

BASIC FORMATTING ALGORITHMS

A formatting algorithm is described as occurring over time. Thus, there is always the notion of *current*, *past*, and *future*. There is the *currently read and processed character or chunk of the input*. There is a *current output line being constructed*; there are the *characters or chunks already inserted into the line*. There is a *current page being constructed*; there are the *lines already inserted into the page*. The article talks about *output lines being built character-by-character or chunk-by-chunk from left to right* and about *output pages being built line-by-line from top to bottom*. If part of the text flows in a different direction, then the formatter is regarded as having to reorder the characters of this part before inserting them into the lines and the pages, so that when a user reads the constructed lines and pages, the text appears to flow in the proper direction.

The reality is that a page to be printed is described in a page description language such as PostScript [1] or PDF [2]. A page to be printed is not necessarily built in this strict left-to-right, top-to-bottom order. However, the results are always equivalent to having been built in this strict order.

A formatting algorithm is classified by the number of *passes* it makes over the input. If an algorithm reads the input only once, it is called a *one-pass* algorithm, etc. An algorithm must have at least two passes if information it needs to format the beginning of of the input cannot be known until it has seen the end of the input. Sometimes, a formatter, e.g., TEX, can simulate an additional pass for a specific purpose by saving in an external file the information that is needed earlier than it is found. This external file is read on the next run of the formatter on the same input.

Describing a formatter requires describing both formatting algorithms and options. Describing each requires understanding the other. To break this vicious cycle, this article describes first vanilla formatting that depends on standard settings of options. Then, an option can be described in detail by describing the effects the option has on vanilla formatting. In the description of vanilla formatting, there are references to some ideas that are described more completely later.

Vanilla formatting deals with ordinary word-by-word unidirectional text that occurs in the interiors of paragraphs. Most any formatter provides ways to achieve special effects, e.g., chapter and section headers; paragraph breaks; centering; indentation; bulleted, enumerated, or labelled lists; footnotes; formulae; tables; etc. Each such feature involves local changes that are quite straightforward to achieve. However, even in these features there are snippets of ordinary text that must be handled by the general formatting algorithms described below.

The simplest formatting algorithm operates on a line-by-line basis. That is, each output line is filled with as many words from the input as is possible given the current typeface size and an adequate minimum spacing between each pair of words. There is a last full word that fits in the output line, and there is its successor word in the input that does not fit in its entirety in the output line. If hyphenation is turned on, this successor word is examined to see if it can be broken into two pieces according to the currently invoked hyphenation rules, such that the first piece together with an appended hyphen fit into the output line being built. If so, the chunk formed by that first piece plus an appended hyphen is added to the output line being

built. If there is no suitable way to divide that next word, the output line is ended with the last inserted full word.

Now, if the formatter does not end justify output lines, the output line is finished, leaving the output with a ragged end margin. If the formatter does end justify output lines, then the chunks in the output line being built are spread apart with additional space as much as is necessary so that the distance between the start of the first chunk and the end of the last chunk is the line length. Generally, the spreading is done so that the sizes of the spaces between all pairs of chunks are the same.

Line-by-line formatting is used by ditroff, and it appears[3] to be that used by MS Word, FrameMaker, and the Mozilla[4] browser.

An enhancement to this simple and any formatting algorithm is to make the space following punctuation a bit longer than that after no punctuation. An enhancement of this enhancement is to make the space following a sentence-ending symbol (e.g., a period) a bit longer than that following a clause-ending symbol (e.g., a semicolon), which in turn, is a bit longer than that following a phrase-ending symbol (e.g., a comma). TEX uses these enhancements and FrameMaker appears to use these enhancements.

The main problem with this simple formatting algorithm is the large variation in the interword spacing from line to line or in the distance from the end chunk to the end margin. When there is end justification, the interword spacing on each line is computed independently of the interword spacing on all other lines. Thus, there can be large variations between the interword spacing on adjacent lines. Some believe that this variation is ugly. Some find it annoying, and some find it tiring to read, as the mind cannot get used to any one spacing for very long. When there is no end justification, the space between chunks is uniformly equal to that of the space character. However, there can be large variations in the space after the end word on each line, giving rise to a very ragged end edge of the text.

Turning hyphenation off exacerbates these variations because on average, each line has fewer words and more space to distribute between the words or after the end word. This exacerbation is a problem with formatters with no automatic hyphenation, such as Mozilla and some versions of MS Word. Automatic hyphenation is not even available in some versions of MS Word 2000.[5] Moreover, based on the appearance of many Word-typeset documents, it appears that many users do not bother to turn automatic hyphenation on.

---

[3] To the author's expert eyes, the output of each of these formatters appears to be formatted line by line, because of the variability of spacing from one line to the next. The manuals or help systems do not discuss this this issue, and the author does not have access to any of the source code. In the cases of ditroff and TEX, the author has access to the proprietary and open source code, respectively. Thus, he is able to be certain about the behavior of the algorithms of ditroff and TEX. This footnote applies to all claims of appearance of the behavior of an algorithm.

[4] You may be surprised to learn that HTML [18], the language of the input to Web browsers, such as Mozilla, allows specification of end justification on a paragraph basis. In the "<p>" markup defining the beginning of a paragraph, if one says "<p ALIGN="JUSTIFY">", the paragraph at hand is end justified when displayed by a browser. Mozilla does a reasonable job of justification on a line-by-line basis. However, Netscape forces end justification for some, but not all, last lines of paragraphs, creating very large gaps between words in last lines that are less than about 7/8 full.

[5] An attempt to turn on automatic hyphenation in the educational version of Word 2000 that this author has fails. Word asks that the user install hyphenation. When the Educational Office 2000 CD is inserted, the user is told that the Professional version must be inserted.

When the line length is short compared to the typeface size, such as in newspaper columns, the variation in interchunk spacing is even more pronounced. When there is end justification, the result is what is called *rivers of white space*, with large gaps of space between very few words on a line. Sometimes, only one word can be fit on a line, forcing either a ragged edge in the middle of start- and end-justified text or space between the characters of the one word.

Perhaps the best variation of the basic formatting algorithm is to format a whole collection of output lines, e.g., those of one paragraph, together. This variation is the approach of TEX and thus of LATEX. TEX's algorithm uses dynamic programming to find the choice of line breaks that yields the least variation in spaces between the chunks in the presence of double justification and the least raggedness of the end margin when end justification is turned off [9]. Thus, a line may not be as full as it can be. If under the simple line-by-line algorithm, one line has significantly larger spacing between the words than the previous line, it is possible that moving the last word of the first of these lines to the second line will result in interchunk spacing that is more uniform over the two lines. For example, compare the spacing in the two lines:

```
11    22    33    44    55
66666                 77777
```

with that in the two lines of the same length:

```
11      22      33      44
55      66666       77777
```

Even though the "55" fits into the first line, spacing is more uniform in the two lines when the "55" is put into the second line. In this manner, TEX's algorithm tries to find the best assignment of chunks to lines of a paragraph, which results in the most uniform sized spaces in the lines of the paragraph. The result is that TEX's interchunk spacing is the most uniform and pleasing of all formatters.

FORMATTER OPERATION

There are two basic flavors of formatters:

1. *WYSIWYG (what you see is what you get), direct manipulation* and
2. *compiled from mark up.*

MS Word and FrameMaker are examples of the former. TEX, LATEX, ditroff, and the Mozilla browser are examples of the latter.

A WYSIWYG, direct-manipulation formatter presents to the user a window containing an approximation of part of one page of the output of the document. The user operates a pointer (e.g., a mouse) to indicate directly where in the page a command is to be obeyed. When a user types characters to be inserted at the pointed-to spot, the characters seem to be inserted one by one directly into a growing line. The result is that when characters are inserted into a new document, the document appears to grow on the screen the same way a typewritten document would grow on paper if the same characters were typed. Of course, the analogy with the typewriter falters when the user backspaces to remove characters or when characters are inserted in the midst of an already full page.

At any time, what the user sees on the screen is a close approximation of what would show up on paper if the document were printed at that time. The approximation is supposed to be as good as the screen resolution allows. The user makes changes to the document by what appears to be directly manipulating the document at the places of the changes.

ditroff input:

```
.SU "FORMATTER OPERATION"
.pp
There are two basic flavors of formatters:
.l1 1
.le
\f2WYSIWYG (what you see is what you get), direct manipulation\fP and
.le
\f2compiled from mark up\fP.
.e1
MS Word and FrameMaker are examples of the former. \*(TX, \*(LT,
ditroff, and the Mozilla browser are examples of the latter.
.pp
```

LaTeX input:

```
\section{FORMATTER OPERATION}

There are two basic flavors of formatters:
\begin{enumerate}
\item
{\em WYSIWYG (what you see is what you get), direct manipulation} and
\item
{\em compiled from mark up}.
\end{enumerate}
MS Word and FrameMaker are examples of the former. \TeX{}, \LaTeX{},
ditroff, and the Mozilla browser are examples of the latter.

%next paragraph begins here, because a blank input line means "new paragraph"
```

HTML input:

```
<h2>FORMATTER OPERATION</h2>
<p>
There are two basic flavors of formatters:
<ol>
<li>
<i>WYSIWYG (what you see is what you get), direct manipulation</i> and
<li>
<i>compiled from mark up</i>.
</ol>
MS Word and FrameMaker are examples of the former. TeX, LaTeX,
ditroff, and the Mozilla browser are examples of the latter.
<p>
```

Figure 3: Inputs with Markup

A formatter that compiles from markup translates an input file into page description language commands that print the formatted pages. The input file contains the time-ordered text of the document with commands inserted at the proper places. Each command is applied at the spot it appears to the text that

immediately precedes and follows it. These commands are often called *markup*. Figure 3 shows parts of the inputs of this article to the ditroff, LaTeX, and Mozilla formatters. The part shown for each is of the same part of the document, the first paragraph of this section. You see the markup for beginning a section, for beginning a paragraph, for a numbered list, for italicizing text, and, in some cases, for referencing macros to typeset the words "TeX" and "LaTeX" in their fancy ways.

Some of the compiled-from-markup formatters have WYSIWYG, direct-manipulation front ends or WYSIWYG previewers that allow at least previewing what will be printed if not direct manipulation. However, the user must remember that what is manipulated is the marked up time-ordered input file.

The reality for a WYSIWYG formatter is that the internal representation has the text with markup, this markup being inserted into the internal representation as a result of the user's direct manipulation on the screen image of the document. In other words, if a user indicates that some selected sequence of characters should be italicized, then internally, there is some indication in the internal representation of the selected characters that says they are now italicized; that indication is effectively markup.

The algorithms given in this article are used in each kind of formatter, albeit in a different way. In a compiled-from-markup formatter, these algorithms are used as part of the translation process exactly as described. In a WYSIWYG, direct-manipulation formatter, these algorithms appear to be applied by the formatter on an incremental basis in order to update the appearance of what the user sees on the screen to what it should be after each change to the document. Optimizations are used to try to compute rapidly the change to what is visible to the user without having to format the whole document after each tiny change. The changes appear to be propagated to the rest of the document during idle periods, when the user's input has slowed down a bit.

CONSTRAINTS AND OPTIONS

A *constraint* is a physical attribute of formatting that has continuous scale of values. An *option* is feature which may or may not be turned on at any given point of any given formatting. Sometimes a collection of options may be mutually exclusive; only one of them can be turned on at any given time.

Constraints

The main constraints of formatting are the size of the typeface, the spacing of the lines, and the dimensions of the printing area.

The size of a typeface is measured in points, each of which is approximately 1/72 inch or .352778 millimeter. The concept comes from the days in which typesetting was done by assembling *metal pieces of type*, one for each character to be printed, into rows, one for each line to be printed. On the face of each such piece is embossed one reversed character. When ink was spread on the faces of the assembled metal pieces and paper was pressed to the faces, the characters appeared on the paper oriented correctly. (See articles on typefaces for more details on this issue.) The size of a typeface was the common height of these pieces of metal type on which characters were embossed. This height would be a bit bigger than the distance from the bottom of the lowest descender, e.g., the bottom of the "q", to the top of the highest ascender, e.g., the top of any upper case letter. This distance is called called *the point size of the typeface*. The most common point size used in books and newspapers is 10. Generally, one does not find smaller than 6 point type, because humans have problems reading text even as small as that. Display type and headlines may be up in the 36 or 72 point range or even higher.

Also the spacing of the lines of the text is measured in points. It is measured by the distance between the baselines of two consecutive lines of the text. Normally, the line spacing is 120% of the current point

size; this spacing yields a comfortable single spacing. Thus, for 10 point type, the usual spacing is 12 points. Smaller than that, e.g., at 110% of the point size, looks rather tight. If the spacing is equal to the point size, then a descender on one line may come very close to touching an ascender on the next line. Spacing that is 180% of the point size is called 1.5 spacing and spacing that is 240% of the point size is called double spacing.

The spacing between lines used to be implemented by strips of lead 1 or 2 points wide placed between the assembled rows of pieces of metal. Hence, the difference between the line spacing and the typeface point size is called *leading*. Thus, 10 point type with 12 point spacing has leading of 2 points.

The dimensions of the printing area is usually measured in picas, each of which is 12 points, about 1/6 inch or 4.2333 millimeters. When one is printing single-spaced 10 point text on letter-sized paper, a typical line length is 39 picas or 6.5 inches. At 12 point spacing, a typical number of lines is 51, covering 51 picas or 8.5 inches. On A4 paper, a typical line length might be 38 picas or about 16 centimeters, and a typical number of lines might be 52, covering 52 picas or about 22 centimeters.

Clearly, the larger the printing area, the smaller the line spacing and the point size, the more text can be printed per page. Moreover, the longer the line width and the smaller the point size, the more uniform is the spacing between the chunks on a line, because more chunks can be put into a line. With very short line lengths or large point sizes, one may be put in a situation in which only one or two chunks can be put on a line, and the spacing between chunks in lines varies radically.

The longer the line width and the smaller the text size, the less necessary is hyphenation to reduce the variability of the spaces between chunks on a line.

(See articles on typefaces, etc. for more details on these and related issues, such as ligaturing).

Options

The main options concern directionality; hyphenation; justification; pagination; and figure, table, and footnote placement.

Directionality

The typical document is unidirectional. That is, all of its text is written in one direction. For example, if a document is an English document, then all of its text is written from left to right. That is, the start of an output line is on the left side of a column of lines and the end of a line is on the right side of the same column of lines. In the vanilla formatting algorithm, filling an output line starts at the left margin, and it ends when adding another word would leave its right end sticking out past the right margin. End justification causes the last chunk on the right end of an output line to end at precisely the right margin of the line.

Conversely, if a document is an Arabic or Hebrew document, then nearly all[6] of its text is written from right to left. That is, the start of an output line is on the right side of a column of lines and the end of a line is on the left side of the same column of lines. If the vanilla formatting algorithm is applied in the right-to-left direction, filling an output line begins at the right margin, and it ends when adding another word would leave its left end sticking out past the left margin. End justification causes the last chunk on the

[6] In Arabic and Hebrew, numerals are written from left to right, i.e., a phone number is written with the first dialed digit on the left side. Users of Arabic and Hebrew typewriters have learned to enter numerals backwards.

left end of an output line to end at precisely the left margin of the line.

There is also the possibility of top-to-bottom writing of text, e.g., for Chinese. In traditional Chinese printing, a line is written from the top of the page to the bottom, with lines laid out from right to left. The details of the vanilla formatting algorithm can easily be adjusted to deal with such top-to-bottom text. Indeed, as is shown in Figure 4 with two layouts of a famous two-line poem by Li Bai, a top-to-bottom layout is identical to a left-to-right layout with the characters rotated 90 degrees counter clockwise.

牀前明月光，疑是地上霜。
舉頭望明月，低頭思故鄉。

牀舉
前頭
明望
月明
光月
，，
疑低
是頭
地思
上故
霜鄉
。。

Figure 4: Left-to-right and Top-to-bottom Layouts of Same Text

Alternatively, a document may be multidirectional. The most common multidirectional document is bidirectional, e.g. for a mixture say of English with Arabic and Hebrew. Indeed, because of the presence of left-to-right numerals, a plain Arabic or Hebrew document can be considered bidirectional. When formatting a bidirectional document, the assumption is that the entire input is stored in *time order*.

The basic bidirectional formatting algorithm assumes this time ordered input and needs to know at any time, the current document direction and the current text direction. The current document direction is as described in the subsection titled “Text Characteristics”.

At any time, the current text direction is the direction of the language of the currently considered character. A Latin character is generally left-to-right (a user can declare otherwise for special effects, e.g., to mimic Da Vinci’s writing) and an Arabic or Hebrew character is generally right-to-left. A seemingly neutral character, e.g., a period, must be regarded as having the direction of its language; thus, there is a left-to-right period and a right-to-left period.

The basic bidirectional algorithm [6, 11, 17, 5, 7] has a standard unidirectional formatting algorithm first break the time-ordered input into lines, taking into account point size, line length, hyphenation, and justification, as if all the text in the document were left to right. Then on a line-by-line basis, the text of each output line is reordered:

for each output line in the document **do**
    if the current document direction is LR **then**
        reverse each contiguous sequence of RL characters in the line
    else (the current document direction is RL)

reverse the whole line;
reverse each contiguous sequence of LR characters in the line
end if
end do

You should verify that applying this algorithm to the time-ordered Arabic, English, and Hebrew line:

Dana said, “سلام دانيال ، שלום דניאל” to Daniel.

yields the visually-ordered line:

Dana said, “לאינד סולש ، لايناد مالاس” to Daniel.

Less common among the multidirectional documents is the tridirectional document. There is an algorithm assuming all input in time order that lets a unidirectional formatter format the time-ordered input file, and then, it reorganizes the characters of the output so that left-to-right text is written from left to right, right-to-left text is written from right to left, and top-to-bottom text is written from top to bottom [4]. It deals with the two horizontal directions as does the bidirectional formatting algorithm. There is an algorithm for doing the reorganization of the top-to-bottom text after bidirectional formatting is done. This algorithm takes advantage of the fact that all characters in Chinese, Japanese, and Korean have bounding boxes that are squares of the same size; thus, any character fits in the bounding box of another. Figure 5 shows a visually-ordered tridirectional English, Hebrew, Japanese, and Chinese document; The English text is printed from left to right from the left margin on one line; the Hebrew text is printed from right to left from the right margin on the next line; the Katakana Japanese text is printed from left to right from the left margin on the next line; and finally the Hiragana Japanese text and the Chinese text is printed from top to bottom at the right margin. Figure 6 shows a stylized time-ordered input for the output of Figure 5; each printed character is shown in its output form, and among this input, there is HTML-like markup defining the language of the text and the applicable document directions. Space constraints do not permit giving the details of the algorithm here.

English

עברית

カタカナ

ひらがな

漢字

Figure 5: Visually-ordered Tridirectional Output

Justification

The most common choice for justification is whether or not there is end justification. Some formatters offer also the option of end justification without start justification. For example, in a left-to-right document, this justification regime is right justification and ragged left. Each line is filled with as many chunks as possible and then all the text is shifted to the right so that the last chunk is flush with the right margin. In terms of the variability of the spacing between words and of the space from the start margin to the start chunk of a line, this justification regime, called *start ragged*, is equivalent to end ragged.

```
<LR>
English
</LR><RL><Hebrew>
תירבע
</Hebrew>
</RL><LR><Katakana>
カタカナ
</Katakana>
</LR><TB><Hiragana>
ひらがな
</Hiragana>
<p>
<Chinese>
漢字
</Chinese>
</TB>
```

Figure 6: Time-ordered Input for Tridirectional Output

Hyphenation

There are two basic choices concerning hyphenation, turning it on or off. If hyphenation is on, the formatter usually uses a hyphenation algorithm that implements basic English hyphenation rules combined with a dictionary, perhaps the same one that is used for spelling checking. Some formatters offer the possibility of overriding the automatic hyphenation decisions with explicit indications of possible hyphenation points in a word, including that there are none. This indication may be directly in the words in the input file or in a separate file providing the user's hyphenation dictionary. When a formatter offers the feature of overriding automatic hyphenation rules, the feature can be used to cause hyphenation according to the rules of a language other than English. Either a pre-pass goes through the document inserting explicit hyphenation points in every word according to the other language's hyphenation rules, or the pre-pass builds a user's dictionary for the words in the document, again according to the other language's hyphenation rules. TEX has a table-driven hyphenation algorithm that is provided with a default table for English. Tables describing the rules for another language can be installed in any formatting run.

Most formatters offer the choice of turning hyphenation on or off. As mentioned, some versions of MS Word 2000 have it, but some do not. In practice, MS Word documents are not hyphenated. TEX, ditroff, and FrameMaker have hyphenation, and it is easy to turn on in each. The Mozilla browser has no automatic hyphenation.

Pagination

The pagination problem is that of determining when the current page is filled with lines. In the absence of stylistic considerations, the problem is almost identical to that of determining when a line is filled with words. Basically, if the next line would end up below the bottom margin on a page, the current page is ended; an optional page footer, including an optional page number is issued; a new page is begun; an optional page header, including an optional page number is issued, and then the postponed line is inserted as the first line on the new page. Some complexity comes when for stylistic reasons, we do not want to create *orphan* or *widow* lines. A orphan line is a section header or first line of a paragraph as the last line on a page. An widow line is a last line of a paragraph as the first line of a page. Neither a orphan nor an widow looks good sitting there all by itself.

It is fairly easy to avoid orphan lines. The formatter can examine any section header or any first line of a paragraph and determine if it would be the last line on a page, that is, there is not enough room to print at least two[7] more lines after the section header or at least one more line after the first line of a paragraph. If there is not enough additional space on the page, the section header or first line of a paragraph goes on the next page.

Avoiding widows is more complex, as it requires having determined one line before the last line of a paragraph, that the next line, i.e., the last line of the paragraph, will end up on the next page. If this determination can be made, then a solution is to terminate the page one line sooner and to put both the next to the last line and the last line of the paragraph on the next page. This determination requires either look ahead or that page breaking be done in a two-pass algorithm. For many formatters, which operate strictly with one formatting pass, the only solution is for the user to notice the widow in one formatting of the document and to then to issue in the next formatting of the document an explicit page break instruction at the right point. The user thus implements a two-pass algorithm manually; that is, after the user's observing the widow and fixing the input, the next run of the formatter is effectively the second pass. When a formatter has a look ahead or a two-pass algorithm, there is a complexity that shows up only when footnotes, figures, and tables have to be placed automatically, as is discussed in the next subsection.

This article is printed in double-column pages. To a formatting algorithm, each column is a page, albeit a narrow one. The formatter does have to be a bit careful not to give each column its own page number and other header and footer items. Also, it is considered good style to make sure that the lengths of all columns sharing a physical page are the same length.

Placement of Figures, Tables, and Footnotes

The general problem is that each figure, table, or footnote may be printed in a place different from that in which it is referenced. A figure or table should appear in its entirety no earlier than the beginning of the page in which it is first referenced. The preference is that a footnote should appear, in its entirety, at the bottom of the page on which it is referenced. However, that may not be possible; there may not be enough room between the footnote reference and the bottom of the page for the entire footnote. In this case, the footnote has to be split over a page boundary. A figure may not be split at a page boundary. A footnote may be split at a page boundary. Tables are a special case; each may be split or not split, depending on the user's taste. Many prefer not to split a table that has vertical lines, and many consider a table with no vertical lines to be splittable. Thus, a figure or nonsplittable table must be printed in its entirety within the margins on one page. Each figure, table, or footnote is what is called a *floater*; that is, it is floated to where it can be printed according to the constraints described below. A figure or nonsplittable table is indivisible and must be printed in its entirety on one page. A footnote or a splittable table may be split at page boundaries.

This article has figures and footnotes, but no tables. The definitions of each of these figures and footnotes were supplied just after their first references. However, you see them elsewhere. Therefore, these items were floated.

It is assumed that the full figure, table, or footnote is input just after the first reference to it. This way, the formatter is able to read the figure, table, or footnote immediately after encountering the first reference to it. The formatter thus knows how big the footnote, table, or footnote is; and it knows whether the table is splittable. It is assumed also that any captions are part of the figure or table.

[7] At least two more lines are needed after a section header to prevent the first line of the section's first paragraph from being an orphan in its own right.

There are two basic algorithms for placement of figures, tables, and footnotes:

1. *one pass greedy* and
2. *multipass, optimizing in some sense*.

In a one-pass, greedy algorithm, after seeing a footnote reference and its footnote, the formatter first reads enough of the text after the footnote to fill out the output line containing the reference. Then, the formatter notes whether there is sufficient room on the current page for the entire footnote. If not, the formatter splits the footnote into two pieces, the part that fits on the current page and the rest. Then it prints the first piece, ends the current page, and arranges for the rest to be printed on the next page as the first footnote. Note that there may be recursion here as the rest may have to be split over the next page and the pages after that.

In a one-pass greedy algorithm, after seeing a figure reference and its figure, the formatter first reads enough of the text after the figure to fill out the output line containing the reference. Then, the formatter notes whether there is sufficient room on the current page for the entire figure. If so, the formatter prints the figure on the current page and continues normal formatting after the bottom of the figure. If not, the formatter arranges to print the figure on the top of the next page and to move, or to *bubble up*, the text that follows the figure to fill in the space at the bottom of the page that was not big enough for the figure. If the figure is simply too large for any page, then the user must arrange to shrink the picture to fit.

A nonsplittable table is treated as a figure.

The steps of a one-pass greedy algorithm are illustrated by some of the parts of Figure 7 that is used to describe several different formatter's algorithms. Each part, (a) through (e), shows a page on top and the next page in the bottom. An upside down "T" represents a reference to the figure or table that follows it. Each of parts (a) and (b) represents a situation in which a figure or non-splittable table fits on the current page. Parts (c), (d), and (e) represent three steps in the treatment of a one-pass greedy algorithm on a figure that is too big to fit on the current page. Part (c) shows the algorithm discovering that the figure or table does not fit on the current page. Part (d) shows the algorithm floating the figure to the top of the next page. Finally, Part (e) shows the algorithm bubbling the text that logically follows the figure or table to the space on the first page vacated by the figure or table that was floated.

In a one-pass greedy algorithm, after seeing a reference to a splittable table and the table itself, the formatter first reads enough of the text after the table to fill out the output line containing the reference. Then, the formatter notes whether there is sufficient room on the current page for the entire table. If so, the formatter prints the table on the current page and continues normal formatting after the bottom of the table. If not, the formatter prints what it can of the table on the current page, ends the current page, and prints the rest of the table on the top of the next page.

Note that the formatter may have to deal with several floaters at once. Doing so is no real problem, as floaters or their pieces can be merged. There is a slight complexity introduced by having to deal with orphan lines in the presence of floats. It may be necessary to avoid a orphan before a figure, table, or footnote appearing at the bottom of a page.

In a multipass, optimized-in-some-sense algorithm, the algorithm uses the first pass to note all the floaters as well as potential orphans and widows and uses some dynamic programming technique to try to place a floater no earlier than the top of the of the output page in which it is referenced and to balance page lengths while avoiding orphans and widows. Sometimes there can be no optimal solution meeting all the constraints, and a figure or table may float to a page before it is referenced or blank lines not in the input are inserted into the output.

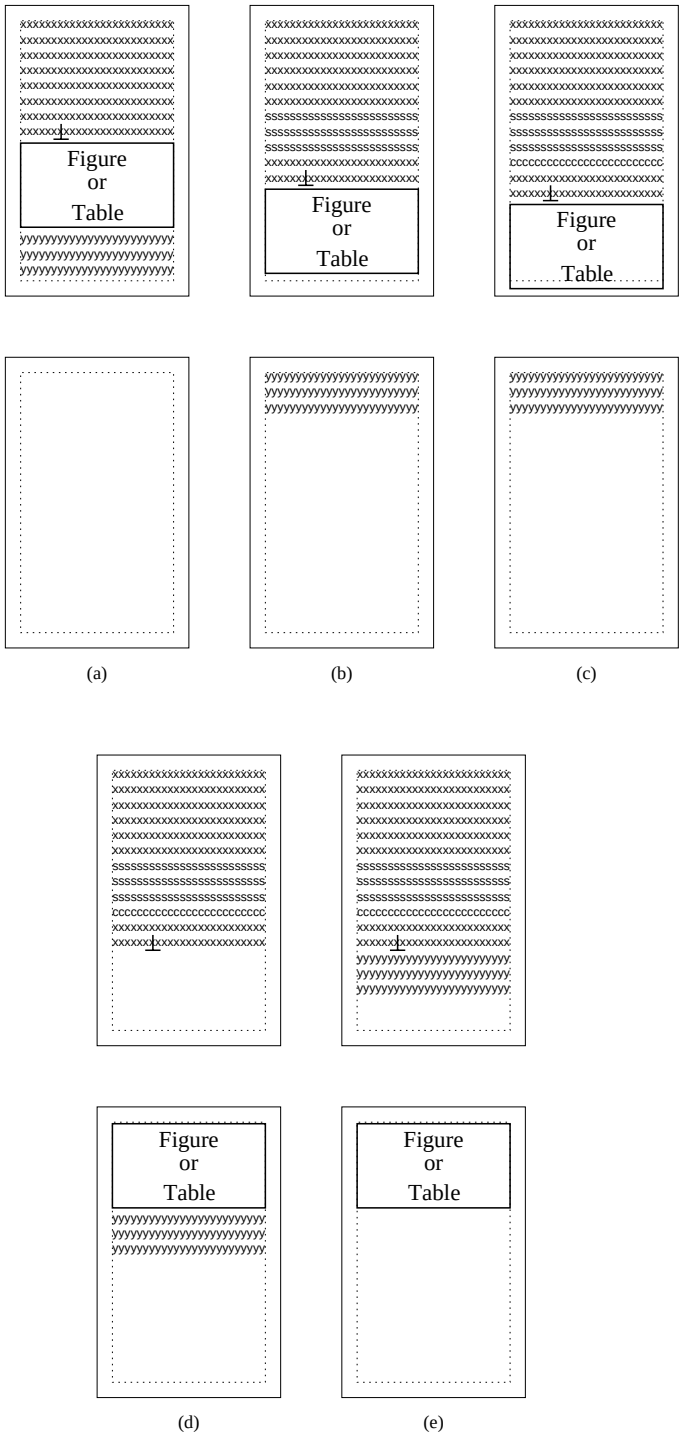

Figure 7: Treatment of Figures and Tables

Ditroff uses a one-pass greedy algorithm. TEX and LATEX use a multipass algorithm. Since Mozilla has an unbounded page length, it does not have to float any figure or table. A figure or table is placed where

it occurs. Footnotes have to be manually placed either where they are referenced as parenthetical remarks, or as end notes, deferred to the end of the page at hand. Word and FrameMaker appear to use a one-pass algorithm, but there is more discussed below, arising from their WYSIWYG, direct-manipulation nature.

The disadvantage of a greedy algorithm is that it sometimes ends up having to put a floater long after its reference because of an overbooking of floaters all referenced very near to each other. Also, sometimes manual intervention is required to deal with widows and to achieve better placement of floaters by putting some floaters before they are referenced. With a multipass algorithm, these manual interventions should not be necessary.

However, the disadvantage of the multipass algorithm is that a figure or table may end up on a page before it is referenced. Also the floater placement and page breaks are very unstable. It occasionally happens that inserting new text **after** a figure causes that figure to move up, changing floater placement completely. If one has been working hard to get floaters placed in a pleasing manner, and has done so in all previous pages, one can find a change on the current page causing a reorganization of previous pages.

A one-pass greedy algorithm is very stable in the sense that no change can affect the output of the text lying before the change. Thus, while manual intervention is required to achieve optimal placement and widow avoidance, one is assured that once he has achieved the desired formatting of pages up to the current one, no change to what follows can affect this formatting. The user can proceed safely through the document page-by-page, getting each page right before going on to the next page with assurances of the immutability of what has already been gotten right.

Particularly messy is the placement in multiple-column text of figures and tables that are wider than the column width. The placement of a figure or table in one column can have a true side effect on a neighboring column whose text must flow around the too-large figure or table that extends into its territory.

Clearly these algorithms and their variants can be used by WYSIWYG, direct manipulation formatters.

FrameMaker seems to provide the full functionality afforded by a one-pass greedy algorithm. A figure or table that must not or should not be split is put into a *frame*. One may *anchor* a frame to a particular point in the text, usually chosen by the user to be the point in the text that references the contents of the frame. The three options for frame placement is that it appears

1. at the top of the page containing the anchor,
2. just below the full line containing the anchor, or
3. at the bottom of the page containing the anchor.

A frame positioned, according to choices 1 or 3, at the top or bottom of the page containing the anchor, moves only when the anchor moves to another page, taking the frame with it to the top or bottom of the new page. A frame positioned, according to choice 2, just below the full line containing the anchor is put there below the anchor if there is sufficient room on the current page for the frame. If there is not sufficient room on the current page for the frame, the frame is moved to the first subsequent page that has room at the top for the frame. The room at the top of a page may be reduced by another frame that is already there by virtue of its being positioned at the top of its page. If in addition, the frame has been designated as *floating*, the text that logically follows the frame is bubbled up to fill the space on the current page that was vacated by moving the frame to the next page.

The parts of Figure 7 show 5 steps in the life of a frame containing a figure or table as text is added above its anchor. In each part, the anchor for the frame is the upside down “T”. Note that in Step (a), in which the frame fits well within the first page, the text before the frame is all “x” and the text below the

frame is all “y”. In Step (b), three lines containing all “s” have been added before the frame, pushing the three all “y” lines to the next page, and the frame still fits within the dotted-line printing margin. In Step (c), one more line, containing all “c”, has been added before the frame, pushing the bottom of the frame to outside the printing margin. Step (d) shows FrameMaker having moved the frame to the top of the next page, and Step (d) shows FrameMaker having bubbled the all “y” lines to after the anchor in the space on the first page vacated by the moved frame.

In FrameMaker, independently of the above, one can specify that text wrap around a frame that is not as wide as the page in which it sits to make use of the space on either side of the frame.

Microsoft Word seems not to be as flexible as FrameMaker. Also in Word, a figure or table that must not or should not be split is put into a frame. A frame can be anchored to a point on a specific page or to a point in a specific paragraph. A frame anchored to a point on a specific page does not move as the text before it grows or shrinks. Rather the text flows around the frame. That is, lines are moved from before or after the frame to after or before the frame as the text before the frame grows or shrinks.

A frame anchored to a point in a paragraph will be put after that point if there is enough room for the frame on the current page. Unfortunately, the text after the frame is not automatically bubbled up to fill the space vacated by the moved frame. Moreover, there seems to be no way to force this bubbling to occur automatically. One way to achieve this bubbling is for the user to manually move the text after the frame to the space vacated by the moved frame. Another way to achieve the *effect* of bubbling is to anchor the frame to the page to which it will be moved. When the frame is so anchored, the text automatically flows around the frame. The drawback of this simulation of the desired behavior is that if the text that would have been the anchor is moved to another page, the figure is not moved. It’s anchored to an absolute position. Under this circumstance, the user must move the figure manually to be anchored in the page that contains the text that would have been the anchor.

Recall the steps that FrameMaker follows through the parts of Figure 7. Basically, for a frame anchored to a point in a paragraph, Word does only Steps (a) through (d), leaving the space vacated by the moved frame to be dealt with manually.

In both FrameMaker and Word, independently of the anchoring of frames and whether or not bubbling is automatic, the user can specify that the text wrap around a frame to fill in the empty space to the side of the frame. When in addition, automatic bubbling occurs, it occurs through the wrapping text.

FURTHER INFORMATION

More information about formatting algorithms can be found in Knuth’s books about TEX [10, 12].

REFERENCES

[1] *POSTSCRIPT Language Reference Manual, Second Edition*, Adobe Systems Incorporated, Addison Wesley, Reading, MA, 1992.

[2] *PDF Reference: Version 1.4, Third Edition*, Adobe Systems Incorporated, Addison Wesley, Reading, MA, 2001.

[3] Adobe, *FrameMaker 6.0 User Guide*, Adobe Systems, Inc., San Jose, CA, 2000.

[4] Becker, Z. and Berry, D.M., "triroff, an Adaptation of the Device-Independent troff for Formatting Tri-Directional Text", *Electronic Publishing* **2**(3), pp. 119–142, October 1990.

[5] Berry, D.M., "Stretching letter and slanted-baseline formatting for Arabic, Hebrew, and Persian with ditroff/ffortid and Dynamic POSTSCRIPT Fonts", *Software—Practice and Experience* **29**(15), pp. 1417–1457, 1999.

[6] Buchman, C., Berry, D.M., and Gonczarowski, J., "*DITROFF/FFORTID*, An Adaptation of the UNIX *DITROFF* for Formatting Bi-Directional Text", *ACM Transactions on Office Information Systems* **3**(4), pp. 380–397, October 1985.

[7] Habusha, U. and Berry, D.M., "vi.iv, a Bi-Directional Version of the vi Full-Screen Editor", *Electronic Publishing* **3**(2), pp. 3–29, 1990.

[8] Kernighan, B.W., "A Typesetter-independent TROFF", Computing Science Technical Report No. 97, Bell Laboratories, Murray Hill, NJ, March 1982.

[9] Knuth, D.E. and Plass, M.F., "Breaking Paragraphs into Lines", *Software—Practice and Experience* **11**, pp. 1119–1184, 1981.

[10] Knuth, D.E., *Computers & Typesetting, Volume B: TEX: The Program*, Addison Wesley, Reading, MA, 1986.

[11] Knuth, D.E. and MacKay, P., "Mixing Right-to-left Texts with Left-to-right Texts", *TUGboat* **8**(1), pp. 14–25, 1987.

[12] Knuth, D.E., *The TEXbook*, Addison Wesley, Reading, MA, 1988.

[13] Lamport, L., *LATEX: A Document Preparation System*, Second Edition, Addison Wesley, Reading, MA, 1994.

[14] Microsoft, *Word*, as documented by online Help, Microsoft, Inc., Redmond, WA, 2000.

[15] Mozilla, "Mozilla Firefox Web Browser", 2004, http://www.mozilla.org/.

[16] Ossana, J.F., "NROFF/TROFF User's Manual", Technical Report, Bell Laboratories, Murray Hill, NJ, October 11 1976.

[17] Srouji, J. and Berry, D.M., "Arabic Formatting with ditroff/ffortid", *Electronic Publishing* **5**(4), pp. 163–208, December 1992.

[18] W3C, "HTML 4.01 Specification", W3C Recommendation, W3C, 24 December 1999, http://www.w3.org/TR/REC-html40/.