Line breaking and page breaking

Jonathan Fine
203 Coldhams Lane
Cambridge, CB1 3HY
United Kingdom
jfine@activetex.org
http://www.activetex.org

Abstract

In their seminal paper of 1981, Knuth and Plass described how to apply the method of discrete dynamic programming to the problem of breaking a paragraph into lines. This paper outlines how the same method can be applied to the problem of page make up, or in other words breaking paragraphs into pages. One of the key ideas is that there must be interaction between the line breaking and page breaking routines. It is shown that TEX can, with one important limitation, fully support such interaction.

This article also shows how TEX can, by using a custom paragraph shape and a special horizontal list, suppress hyphenation of the last word on a page.

Introduction

For many years the conventional wisdom has been that TEX is good at breaking lines into paragraphs (and setting mathematics and tables), but that it is not at all good at page make-up. There is some measure of truth in this statement. However, it is the author's view that almost all of the deficiencies arise not from TEX the program, but from the macro packages and other tools used with it.

The main problem considered in this article is that of suppressing hyphenation on the last word of a page. Traditionally, this has been avoided wherever possible, for it breaks the reader's concentration, to have to go to the next page (rather than the next line) to complete a hyphenated word. The solution proposed involves constructing a special horizontal list, and an unusual paragraph shape.

Don Knuth's view of TEX's line-breaking algorithm is well expressed by this passage from *The TEXbook* (page 94):

> The remainder of this chapter explains the details precisely, for people who want to apply TEX in nonstandard ways. TEX's line-breaking algorithm has proved to be general enough to handle a surprising variety of different applications; this, in fact, is probably the most interesting aspect of the whole TEX system. However, every paragraph from now on until the end of the chapter is prefaced by at least one dangerous bend sign, so you may want to learn the following material in easy stages instead of all at once.

and twelve pages of technical details follow. Not all of it used here. The article by Knuth and Plass [4] and Plass' thesis [10] are also well worth consulting.

From ASCII to `dvi`

It will help to have an overview of the process by which TEX converts its input file into typeset pages. In general terms the process is the same for all macro packages, but at each stage each package can use in different ways the capabilities offered by TEX the program.

Here we divide the process into seven stages, namely ASCII, tokens, macros, horizontal list, lines, vertical list and pages.

This section concludes with a discussion of the look-ahead problem, whose solution is an important part of the line-breaking algorithm. The same problem arises in page make-up, and it is the present lack of a solution that has given rise to the view that TEX is not good at page make-up.

ASCII This is the input stream, a text file marked up in some syntax, formal or informal. The input file might contain macro definitions and parameter settings, as well as the text to be typeset. For example, with LATEX the body size is a parameter to the `\documentclass` command.

Strictly speaking, the input stream need not be ASCII. TEX is capable of reading 8-bit input files.

We use ASCII as a convenient shorthand for the input text file.

Tokens Internally, TEX deals with tokens. Category codes control the translation of input characters into tokens.

Traditionally, the ASCII character 'a' is given category code letter, which means that when read it become the token 'the letter a'. The same goes for the other letters. Digits and punctuation are given category other, and so the digit 7 become 'the character 7' when read.

Certain other symbols, such as `{`, `}` and `\`, have special category codes. It is this that gives TEX its familiar 'backslash and braces' input syntax. However, this syntax is not built into TEX the program.

Internally the only tokens that TEX has are character tokens (of various categories), and control sequences. (A control symbol is a control sequence whose name is a single character, usually a non-letter.) The traditional category codes cause the 'eyes of TEX' to convert the sequence of characters `\wibble` to the control sequence whose name is, well, `wibble`. This is done by TEX the program, as part of the process that turns ASCII characters into tokens.

In Active TEX, every input ASCII character is an active character. An active character is rather like a control sequence, in that it has a meaning, and this meaning can be changed at any time. However, its 'name' is the active character 'x', or whatever it is. In plain TEX, the '`~`' character is active.

Active TEX does not use the 'eyes' of TEX the program to form control sequences. Instead, it uses macros and the `\csname` primitive to form control sequences out of the active characters that it receives from the eyes of TEX. This means that it never has to change category codes, in order to achieve special effects, such as verbatim typesetting.

Macros The internal tokens of TEX (or more exactly their meanings) can be divided into two classes, namely the expandable and the unexpandable. Most expandable tokens are macros, and most of the primitive commands of TEX are unexpandable. However, some primitive commands, such as `\ifx`, the other conditional commands and `\csname`, are expandable.

Unexpandable commands do something (in the stomach of TEX the program), while expandable commands and macros control what it is that is done. In plain TEX the '`~`' character, which is active, is defined to be a macro that places a penalty and some glue on the horizontal list. This produces an unbreakable interword space.

Horizontal list TEX would not be able to typeset without commands that placed items on the horizontal list. The internal token 'the letter a' (obtained say by reading an 'a' from the input ASCII file) will place a 'character box' that is the character 'a' in the current font onto the current horizontal list. (This is only in horizontal mode. In math mode it does something else.) The internal token 'the character 9' behaves in the same way.

There are other items that can go on the horizontal list. For this article, we need to know only about glue, penalties and discretionary penalties. Glue is potentially stretchable and shrinkable interword space, while penalties record the undesirability of making a line break at this point.

Discretionary hyphens are hyphens that are optional. The line breaking algorithm can break lines at discretionary hyphens. If the break is taken at a discretionary hyphen, the hyphen appears, and otherwise nothing appears. Discretionary hyphens can be placed onto the horizontal list either explicitly, via the execution of a primitive command, or implicitly, as a result of the hyphenation algorithm.

Lines TEX's line breaking algorithm turns a horizontal list into a sequence of lines. It does this by choosing a sequence of break points in the horizontal list. Most of the time, any glue and penalty items after a chosen break point are discarded. This allows the interword glue to disappear at line breaks.

Normally, TEX breaks the paragraph into lines using the current value of the `\hsize`. However, the `\parshape` parameter allows the width (and offset) of each line to be specified individually.

Vertical list After the paragraph has been broken into lines, TEX places the lines onto the current vertical list. Often, this vertical list is the main vertical list, also known as 'the current page'. Each line of the paragraph is a box (in fact a horizontal box). As well as boxes, a vertical list can contain (vertical) glue and (vertical) penalties. A vertical list can also contain other items, such as insertions, that do not concern us here.

The line breaking algorithm places (vertical) glue between the lines, so that the baseline to baseline distance between the lines is uniform (unless the lines contain exceptionally tall or deep set matter). It also inserts (vertical) penalties between the lines, to aid in the page breaking process. The `\clubpenalty` is the extra penalty for a page break immediately after the first line of a paragraph. The

`\widowpenalty` is the extra penalty for a page break immediately before the last line of a paragraph.

Pages Whenever something is placed on the main vertical list, TEX the program checks to see if it has accumulated enough to break off from it the current page. If it has, then TEX chooses the best of the available break points on the main vertical list. It then calls a special token list, the `\output` routine, to add page numbers and the like to the broken off portion, and to ship it out to the `dvi` file. The `\output` routine is part of the macro package.

The `dvi` file This is the output file produced by TEX the program. As well as recording the placement of every character and every rule on the page, it can contain what are known as `\special` commands. Programs that process `dvi` files can read the specials, and use them as parameters to their actions. For example, a special might request the placement of a graphic.

The look-ahead problem TEX's line breaking algorithm 'looks-ahead' to the end of the paragraph before it makes any decisions as to where the first (or any other) line break occurs. Each line break is, so to speak, considered not by itself but in the context of the other line breaks.

The page breaking algorithm does not perform such a look-ahead. Each page break is considered in isolation, without regard for its consequences later in the document.

At the end of a paragraph, the line-breaking algorithm is called, and it produces lines of text. These lines are then placed on, say, the main vertical list. If enough material has accumulated, the page-breaking algorithm cuts off enough material for one page, and the output routine is called.

Thus, from the end of the paragraph to the calling of the output routine, everything is under the control of TEX the program. During this time neither the user nor the macro programmer has any opportunity to influence TEX's behaviour, other than through the values of parameters and the contents of the horizontal and vertical lists.

TEX's page-breaking algorithm clearly is deficient for complex work. One needs to be able look ahead, when there is floating matter to be placed. Multiple column layout is particularly complicated. There are two aspects to the problem. The first is that an improved algorithm requires more than the information local to the current page. The second is what it does with this information.

This article concentrates on making information available to an improved page-breaking algorithm, but has little to say on the internals of such an algorithm. As in the line-breaking algorithm, the page-breaking algorithm selects one sequence of possibilities from the many presented to it

The line-breaking algorithm has look-ahead. Its context is the current paragraph. To avoid hyphenating the last word on the last line of a page, the algorithm needs to know where that last line will fall (unless it suppresses all hyphenation, and so is done with the problem). Therefore, the page-breaking algorithm will have to feed information back to the line-breaking algorithm.

Once the location of the page break is known, this information can be fed to the line-breaking algorithm (in the form of a custom paragraph shape). Provided a suitable horizontal list is constructed, the algorithm will suppress hyphenation at the required point. How this is done will be shown later in this article.

Discrete dynamic programming

The purpose of this section is to describe those parts of TEX's line-breaking algorithm that are specially relevant to this article. This has two aspects. The first is those features that are relevant to suppression of hyphenation of the last word on some specified line of a paragraph. The second is those features that help us to understand what can be done for global optimisation of page breaks, and for establishing communication between the line-breaking and page-breaking algorithms.

In our simplified model, a horizontal list contains character boxes, glue, penalties and discretionary hyphens. Glue and penalties are what are known as discardable items. They can disappear at a line break. The other items are non-discardable. They will never disappear.

A legal breakpoint is any (finite) penalty, any discretionary hyphen, and any glue item, provided the glue is immediately preceded by something that is non-discardable. For any sequence of breakpoints, there is quantity called the total demerits, that depends on both the chosen breakpoints and on parameters that can be set by the macro programmer.

For example, when breaking at a penalty, the amount of the penalty is part of the sum that is the total demerits. Similarly, the `\hyphenpenalty` and `\exhyphenpenalty` parameters are the contributions made by discretionary and explicit hyphens respectively. If the line had been set loose or tight (shorter or longer than its optimum width) then a badness for the line contributes to the total demerits.

Of all the possible sequences of breakpoints for a given paragraph, TEX chooses one that has the smallest possible value for the total demerits. It does not choose the breakpoints line by line, or in other words locally. The breakpoints are chosen with a view to the whole paragraph, or in other words globally.

The way in which it does this is interesting, because in general there are so many possible sequences of breakpoints, that it is impossible for them to be considered individually. The method used is known as discrete dynamic programming. This method allows the last line of a paragraph to 'communicate' with the first. (It is communication not in the sense of sending a message, but in the sense of being part a common larger whole.)

To save time, TEX tries first to make the paragraph without using any hyphenation. The parameters `\pretolerance` and `\tolerance` are limits on how bad a line can be respectively before and after hyphenation. For simplicity, we will assume that hyphenation is alway tried, say because the pretolerance is zero.

A sequence of breakpoints is said to be feasible if no line has badness exceeding the tolerance. The line-breaking algorithm considers only feasible sequences of break points. For formal reasons, the end of the paragraph is considered to be a breakpoint. It is, after all, the end of a line.

The formula the algorithm uses to compute the total demerits has the following useful property. Suppose an optimal sequence of breakpoints is selected, and say lines 5 to 9 are of the horizontal list are considered in isolation from the remainder of the horizontal list. The optimal sequence of breakpoints for the whole paragraph, when restricted to the isolated lines, is also optimal for the line-breaking problem represented by the isolated problem. This is called the property of locality. It is a property of the formula for total demerits.

Discrete dynamic programming, as applied to line-breaking, consists of the following. Start at the beginning of the paragraph. Calculate the feasible breakpoints for the end of the first line. From these breakpoints calculate the feasible breakpoints for the end of the second line. We now prune the list of feasible sequences of breakpoints. If two or more sequences end the second line at the same point, keep only the best one. (If several are joint first, keep only one.) For each of the remaining two-line breakpoint sequences, compute all the feasible extensions to three-line sequences, and prune as before.

As this process continues, so the number of both feasible and locally optimal sequences will in general grow. However, the growth will not be too rapid. Consider the spread in the location of the breakpoint that is the end of, say, the nth line. If the first n lines contain as little set matter as is possible, then we get one location in the horizontal list. If they contain as much as is possible, we get another. This is the spread. It is roughly linear in n. The number of breakpoints in this spread is the number of locally optimal breakpoints that the algorithm must carry along to the $n + 1$ stage.

This analysis limits the running time to of the order of n^2. However, we can do better. When the spread gets large, it will cover the the length of a whole line, and so some of the calculations for $n + 2$ will have been done as part of $n + 1$. This also shows why using a custom paragraph shape is computationally expensive. There is no longer such a sharing of computations between lines.

The line-breaking and the page-breaking algorithms have a certain amount in common. This is how Don Knuth puts it in *The TEXbook* (page 100):

> TEX breaks lists of lines into pages by computing badness ratings and penalties, more or less as it does when breaking paragraphs into lines. But pages are made up one at a time and removed from TEX's memory; there is no looking ahead to see how one page break will affect the next one. In other words, TEX uses a special method to find the optimum breakpoints for the lines in an entire paragraph, but it doesn't attempt to find the optimum breakpoints for the pages in an entire document. The computer doesn't have enough high-speed memory capacity to remember the contents of several pages, so TEX simply chooses each page break as best it can, by a process of "local" rather than "global" optimisation.

The situation is not impossible though. In Appendix D (page 400) Don Knuth writes:

> An output routine can also write notes on a file, based on what occurs in a manuscript. A two-pass system can be devised where TEX simply gathers information during the first pass; the actual typesetting can be done during the second pass, using `\read` to recover information that was written during the first.

Provided sufficient information can be gathered in the first pass, it can then be presented to TEX's line-breaking algorithm, or some other program, so that an optimal choice can be made from amongst

The sixth line has zero width. This prevents hyphenation of the word at the end of the fifth line. This is because when a word is hyphenated, part of the word is placed on the next line. (This is the

very purpose of hyphenation.) A special sequence of items of glue and penalties is placed between words. This allows the interword glue to span the zero-width line.

Figure 1: Example of suppressed hyphenation

those which are feasible. The second pass can then do the actual typesetting.

Avoiding 'last-word' hyphenation

In this section we explain how a suitable horizontal list and paragraph shape specification taken together will cause the line-breaking algorithm to suppress hyphenation of the word at the end of some specified line.

The basic idea is quite simple. Hyphenation places matter on the next line. Indeed, this is the very purpose of hyphenation. However, if the next line is not long enough to hold even the smallest fragment of a word, then the word at the end of the previous line will not be hyphenated. (It is possible for a very long word to be hyphenated two or more times. Each hyphenation point is a legitimate breakpoint.)

We can achieve this effect on say the fifth line by making the width of the sixth line equal to zero. This however creates a problem. If we use ordinary glue between words, then between any two words there will be only one breakpoint, namely the glue that was between the words. For some word to be allowed to occur at the end of the fifth line, it must be followed by a special piece of 'glue', that is capable of spanning the zero width sixth line.

Recall that in our simplified model (which is all we need), breaks can occur at penalties, at discretionary hyphens, and at glue that is preceded by something that is not discardable. To go further, we need to understand exactly what happens at a line break.

According to *The TEXbook* (page 97):

> When a line break actually does occur, TEX removes all discardable items that follow the break, until coming to something non-discardable, or until coming to another chosen breakpoint. For example, a sequence of glue and penalty items will vanish as a unit, if no boxes intervene, unless the optimum breakpoint sequence includes one or more of the penalties.

In other words, most of the time discardable items are discarded, but any (finite) penalties are allowed to be part of the breakpoint sequence, if that is what the algorithm decided to do. In other words, when moving on to the next feasible breakpoint, it has something of a free choice in the discarding of discardables.

Therefore, each piece of 'glue' between words will have to contain two legitimate break points, as well as an ordinary piece of interword glue. The way to get this is to place two penalties of zero, followed by the ordinary interword glue. (The penalty for breaking at glue preceded by a non-discardable, such as a word, is zero. Thus, in ordinary cases we get the same behaviour as before.)

Something similar arises in ordinary practice. Sometimes a line is deliberately left short, say because the next word is too long to fit on the line, and it cannot be hyphenated. The standard way to achieve this is to insert `\hfil \break` in the line. The `\break` is just a shorthand for a penalty of zero, and the `\hfil` is glue that stretches to fill the line. When the line has zero width, no glue is required to fill it.

In August 1999, the author posted to the newsgroup `comp.text.tex` example code that suppressed hyphenation. A lively debate followed, but not until the author came to write this article did he discover, to his shock and horror, that the code he posted last summer did not work in many cases. In the first version of this paper, his solution had an unnecessary but harmless zero-width piece of glue between the two penalties. This was not noticed until after the paper had been refereed. Clearly, some of us have something to learn about penalties and glue.

The `echowords` environment

Figure 1 shows the result of applying the methods of the previous section. So that there are many hyphens, a discretionary hyphen has been placed between adjacent letters of a word. The spaces between words contribute, as described in the previous section, two penalties of zero and an ordinary interword space. Although it is clearly possible to construct such a horizontal list by hand, doing so is laborious and prone to error.

Instead, the author has used Active TEX to simplify the form of the example's input. In fact the author wrote

```
\begin{hyphdemo}{5}
The sixth line has zero width.
[...]
zero-width line.
\end{hyphdemo}
```

and it is the purpose of this section and the next to describe the macros that were used.

By way of an example, this section contains the complete source for a LaTeX environment that echoes its content to the console, word by word. Here a word is a maximal sequence of visible characters. White space separates words. The next section gives the listing for the `hyphdemo` environment.

In the header of the source for this article the author wrote:

```
\RequirePackage{atcode}[2000/07/22]
\RequirePackage{atlatex}[2000/07/22]
```

and this loads two Active TeX macro packages.

The first of these packages defines a programming environment in which it much easier to write macros that make extensive use of active characters. It has other advantages, which make it useful even when writing macros that have no special features. One uses the command `\@code` to enter the environment, and `]]` to exit it. Later in this section there will be examples of the input syntax and programming style for the package.

The first section also define macros for making all ASCII characters active, and giving them standard meanings. More exactly, the active character 'a' is a macro whose expansion is the control sequence `\active:lcletter` followed be the active character 'a'. This apparent recursion is very useful, for it allows each active character to know its own identity. By letting the prefix control sequence `\active:lcletter` be `\string`, for example, a character can be made to typeset itself.

The second of these packages defines a new command, `\active:latex`, that makes it possible for LaTeX macro programmers to access the facilities of Active TeX. The `hyphdemo` environment above is coded using this command, within an `\@code` programming environment.

First we enter the atcode environment. What follows is the most general way, for it does not assume that the ASCII character `@` has category code letter. There must be a space before the `@`, and no space after the `code`.

```
\csname @code\endcsname
```

Now all ASCII characters are active, and we are in the atcode environment. Here, control sequences do not need to be prefixed by a backslash. You can use a backslash as a prefix, but it is neater to omit the backslash whenever possible. Strings, however, have to be enclosed in double quote marks. Here is an example. (This semi-colon is not syntactic sugar. It tells atcode that it is safe to release the tokens it has been accumulating. Semi-colons within braces, however, are syntactic sugar. The same goes for commas.)

```
message { "Hello world" } ;
```

Next we set up a shorthand feature. This allows us to type `.digit` instead of `active:digit`, and so on. From now on we will drop the backslash before control sequence names, in both text and in atcode source. We will also drop the `active:` prefix. Thus, `active:latex` is `.latex`.

```
def active:prefix { "active" } ;
```

The macro `get.word` parses `the.word` from the input stream, and calls `do.word` to process it. It depends on `init.get.word`, whose value will be set by the calling context. It also relies on some system macros that have not yet been described. In the atcode environment white space is ignored, unless it is part of a string or the like.

```
def get.word
{
  begingroup ; aftergroup do.word ;
    init.get.word ;
    .suspend.white.space ;
    let .suspend .end.xdef ;
    xdef the.word { iffalse } fi ;
}
```

Here is the definition of the `echowords` environment. It provides an example of the `.latex` command. The `]]` closes the atcode environment. All is as it was before except that the macro `get.word` and the environment `echowords` have been defined.

```
newenvironment { "echowords" }
{
  begingroup ;
    let .lcletter get.word ;
    let .ucletter get.word ;
    let .digit get.word ;
    (default) ; let .symbol get.word ;
    let .rs relax ; let .re relax ;
    let .re-sp relax ;
    let !  relax ; let |D09 relax ;
    let init.get.word .string.visible ;
    def do.word
      { message { the.word } } ;
    .latex ;  //  must come last
}
{ endgroup } ;
```

```
]] %% now back to the usual catcodes
```

We will now explain what is going on. First, the `.latex` command. This command opens a group, in which all ASCII characters are active. It looks for a line that begins with the (active) characters `\end{`. When it finds this, it closes the all-active group, and pretends that it had read the `\end{` with LaTeX's normal category codes. Thus, the first and last of the input lines

```
\begin{echowords}
 These words are echoed,
 one by one.
\end{echowords}
```

are processed by the begin and end commands of the `echowords` environment. The two lines in the middle are read and processed with all characters active, and with the values set by the environment.

We simplify slightly. To avoid needlessly filling TeX's macro processor (the mouth) with a long list of tokens, this looking for the `\end{` is done on a line by line basis. However, this makes no difference in practice.

The first three assignment commands tell Active TeX how to process letters (upper- and lowercase) and digits. Symbols are rather different. Most if not all of the time, all lowercase letters are dealt with by the same rules. The same goes for uppercase letters, and for digits. It often happens, however, that each symbol has a specific meaning. For example, in the atcode environment, each symbol has a distinct meaning of its own.

For this reason, Active TeX uses the concept of symbol sets. Within its realm, it 'owns' all the active symbols. (This is done in a way that does not interfere with their use outside of its realm.) Instead of directly assigning a value to a symbol, one selects a symbol set, perhaps of one's own creation. The owner of a symbol set is free to change the meaning of symbols in that set. For as long as that symbol set is selected, for almost all practical purposes changing the meaning of a symbol in a set is the same as changing the meaning of the active symbol itself. However, when a different symbol set is selected, the meaning of all the symbols changes to those of the newly selected set.

Parentheses, as above, are used to select a symbol set. The `(default)` symbol set is part of the atcode package, and in it every symbol expands to the control sequence `.symbol`, followed the active symbol itself. Thus the line of code:

```
(default) ; let .symbol get.word ;
```

causes all symbols to call `get.word`. In short, all visible characters are to call the `get.word` command we just defined.

The `.latex` command 'owns' the active end-of-line character. Only when one knows for sure that it is safe to do so, should one change its meaning. It is used by `.latex` to inspect the next line for the `\end{` characters.

At the start and end of each non-blank input line, `.latex` generates `.rs` and `.re` events. Blank input lines generate the `.rs-re` event. These events are control sequences, whose values can be set by the macro programmer. Here we are setting them to do nothing. (One can think of the visible characters as similarly being events, but this time with parameters.)

We have now initialised all the ASCII characters except for space and tab. The next line sets them both to `relax`. (The construction `|ABC` generates a character whose category code is hexadecimal `A`, and whose character code is hexadecimal `BC`. Thus, `|D09` is active tab.)

The low-level events (reading a character from the input stream) have now been dealt with. They create higher level events, namely the initialisation of the parsing of a word, and the processing of the word once parsed. The `.string.visible` macro is a low-level system macro that causes all visible characters to behave as if they were characters of category code other. This system macro by-passes the symbol set mechanism. It runs quicker, but must be used with care.

We are almost done. There are some commands in `get.word` that need explanation. The command `.suspend.white.space` cause the active form of the white space characters (space, tab and end-of-line) to expand to `.suspend` followed by the active white space character. This should only be done within a group, which is closed by white space. The parsing of a word is exactly such a context.

Finally, the `atlatex` package contains a helper macro that is very useful for closing a 'flying `xdef`'. Here is its definition.

```
def .end.xdef
  { iffalse { fi ; } ; endgroup } ;
```

To conclude, we reconsider the `get.word` macro.

```
def get.word
{
  begingroup ; aftergroup do.word ;
    init.get.word ;
    .suspend.white.space ;
    let .suspend .end.xdef ;
    xdef the.word { iffalse } fi ;
```

```
}
```

It opens a group. After the group, we call `do.word`. The variable part of the initialisation routine, namely `init.get.word` is defined to make all visible characters 'other'. Thus, when the `xdef` which closes the macro executes, it simply accumulates visible characters in `the.word`. (The `iffalse` is a hack that allows a macro to 'contain' unbalanced braces.) The two suspend commands cause white space to close the `xdef`, and thereby trigger the processing of the word.

The reader may find it instructive to run these macros with `tracingall` on, and examine the resulting log file.

The `hyphdemo` environment

Our next example is more substantial. The suppression of hyphenation on the last line of a page requires the construction of a fairly special horizontal list. Here is the sequence of penalties and glue that is to be placed between words. (We use `hd` as a two letter prefix for 'hyphenation demonstration'.) But first we enter the atcode environment.

```
\csname @code\endcsname
def hd:iwspace
{
  unskip ;
  penalty "0 " ; penalty "0 " ;
  ~ ;  // ordinary interword space
}
```

The `unskip` is in case we get two spaces in a row. This is not rigorous, but in the context it is good enough. Then we put down two penalties, which allows `hd:iwspace` to span a blank line. Finally, we put down an ordinary piece of interword glue. In Active TEX, `~` produces an ordinary space character.

So that we get lots of hyphens, we will place a discretionary hyphen between adjacent letters in a word. To do this, we use a variant of the `get.word` command. This macro applies `string` to the first character in the word. Each subsequent character is then responsible for putting down a discretionary hyphen before `string`ing itself. To avoid hyphenating just before punctuation at the end of a word, symbols do not insert a discretionary hyphen.

```
def hd:get.word { get.word ; string } ;
def hd:init.get.word
{
  def .lcletter { \- ; string } ;
  let .ucletter .lcletter ;
  let .digit .lcletter ;
  let .symbol string ;
}
```

The `hyphdemo` environment takes a single parameter, namely the number of the line, at the end of which hyphenation is to be suppressed. This parameter controls the construction of a custom `parshape`, which will be coded later. If the parameter is zero, no suppression is offered.

The parameters encourage hyphenation. The large value of the line penalty is to stop the linebreaking from making the lines very loose, just so it can get the reward (negative penalty) for the additional hyphen.

```
newenvironment { "hyphdemo" } [1]
{
  par ;
  begingroup ;
    hyphenpenalty "-100" ;
    doublehyphendemerits "0 " ;
    linepenalty "200 " ;
    leftskip "2pc " ;
    rightskip leftskip ;
    hd:set.parshape { #1 } ;

    let .lcletter hd:get.word ;
    let .ucletter .lcletter ;
    let .digit .lcletter ;
    (default) ; let .symbol .lcletter ;
    let !  hd:iwspace ;
    let |D09 !  ; let .re !  ;
    let init.get.word hd:init.get.word ;
    def .re-sp { par } ;
    def do.word { the.word } ;
    .latex ; //  don't forget this
}
{ par ; endgroup }
```

The remainder of the definition of this environment sets up the conditions for the parsing and processing of words, in much the same way as in the previous section. Note that `let do.word the.word` would be very wrong. This would cause the macro to continually process the value of `the.word` that was current at the start of the environment.

The difficult part of setting the parameters is to feed the parameters to TEX's `parshape` primitive. It takes $2n + 1$ parameters, where n is the number of lines, whose width we are specifying. These are TEX number and dimension parameters, and not macro or token parameters. We use aftergroup accumulation to build up this list. Scratch counters are used to hold the values of parameters whose values have to be calculated.

```
def hd:set.parshape #1
{
```

```
    count@ #1~ ;
    ifcase count@ ,
    else
      advance count@ tw@ ;
      dimen z@ leftskip ;
      advance dimen z@ rightskip ;
      def temp { z@ ; hsize } ;
      begingroup ;
        aftergroup parshape ;
        aftergroup count@ ;
        count@ #1~ ;
        loop ; ifnum count@ > z@ ,
         aftergroup temp ;
         advance count@ m@ne ;
        repeat ;
      endgroup ;
      //  accumulated tokens released here
      z@ ; dimen z@ ;  //  zero-width line
      temp ;  //  remaining lines normal
    fi
  }
```

Some words of explanation. If the parameter is zero, we do nothing, otherwise we store in scratch registers the number of lines in the parshape, and the sum of the `leftskip` and the `rightskip` (the actual width of a 'zero-width' line). We then store in `temp` the specification for a normal line. A process of 'aftergroup accumulation' is then used to build up the `parshape` along with its parameters. (The `@code` environment uses the same method to gain its power. The asterisks problem in Appendix D of *The TEXbook* (page 373) is a simpler example of this.)

Outside the group, `count@` is $n+2$, the number of lines in the paragraph shape. Inside the group it is set to n, which is the number of lines before the zero width line. (Grouping ensures that the two values do not interfere with each other.) The loop accumulates n `temp` tokens. At the end of the group, the accumulated aftergroup tokens re-appear. The zero-width line and the final `temp` complete the paragraph specification.

All that remains now is to close the atcode environment.

```
  ]]
```

Flexible paragraphs

Sometimes it is helpful, for purposes of page make-up, to set a paragraph slightly longer or shorter than is optimal. For this purpose TEX provides the `looseness` parameter. Negative values of looseness can be thought of as tightness. If the looseness is 1, then TEX will try to make the paragraph one line longer than it would otherwise. Traditionally, in the TEX world, looseness is applied by hand, when fine-tuning the document for publication.

Let us consider now how it might be done automatically. Ahead of time, we will not know how long we will want the paragraph to be. Nor will we know where the paragraph appears on the page, and thus which custom paragraph shape to use. Therefore, we shall consider all possibilities.

We might find, for example, that a given paragraph can be set using 9, 10 or 11 lines. We might also find that when 9 lines are used, we can suppress hyphenation at the line breaks 4, 6, 7 and 8. (We are lucky if we can suppress hyphenation early on in a paragraph.)

The following table represents this data about 9-line versions of the paragraph. Each line gives a way of breaking the paragraph, and the number pointed to by the arrow is the total demerits for the optimal way of so breaking the paragraph. Similar tables can be constructed for the 10 and 11 line versions of the paragraph. Such a report, on the flexibility of all paragraphs in the document, will be the input for the global optimisation algorithm considered in the next section.

```
4+5->3489
6+3->2748
7+2->2956
9->2413
```

The reader may object that to prepare such a report, even by computer, will take a long time. This may be true, but the situation is not hopeless. First, if the document is in its final form, this report need be prepared only once. The page-breaking algorithm, by design, requires no knowledge of the document, other than this report.

Second, even if the document is not in its final form, changes are likely to be confined to a small proportion of its paragraphs. Matters can be configured, provided macros have been written with this in mind, so that fresh report data need only be generated for the paragraphs that have changed. This is probably something that could be done in real time on the entry-level hardware available today.

What is true is that much more time will be spent on trial paragraph breaking, to generate the report, as is spent on breaking the paragraphs for the final triumphant globally optimised version. The same is true, however, of the line-breaking algorithm.

Indeed, the two are yet more similar than this. Discrete dynamic programming depends on the principle of local optimality, which is a property of the

formula for total demerits. One consequence of this property is the following. If an optimal sequence of breakpoints takes in the feasible breakpoints A and B, then over the range A to B this sequence is also optimal for this local form of the problem.

Now suppose A is 'sufficiently distant' from the start of the paragraph, and that B is 'sufficiently distant' from A. Here, 'sufficiently distant' means that there is a feasible (but not necessarily optimal) sequence of breakpoints linking the two points. If enough set matter of random width intervenes between the two points, the concept has its intuitive meaning. In these circumstances the line-breaking algorithm will find an optimal sequence of breakpoints between A and B. It will do this whether or not this is part of the finally chosen optimal sequence.

Thus, the line-breaking algorithm finds not only the best breaking for the whole paragraph, but also for a great many portions of the paragraph. In the same way, any worthwhile discrete dynamic programming solution to the problem of global optimisation will consider most or all possible feasible ways of breaking the paragraphs that constitute the document. The strength of Knuth and Plass's algorithm is not that it runs quickly in abstract, but that the running time is roughly linear, rather than quadratic, in the size of the problem. Because of linearity, in time hardware will be able to catch up, even if the problems are large.

Global optimisation

This section describes briefly how a report on paragraph, as in the previous section, can be used as the input for a global optimisation process. For simplicity, we assume that we are setting straight text on a grid, and that hyphenation is to be suppressed on the last word of each page. We also assume that no paragraph is longer than a page, or in other words, that it cannot span two page breaks.

First, it is convenient to recast each paragraph's report into the following form. We give the possibilities in order first of the number of lines before the potential page boundary, and then in order of the number after. Thus, a fragment of paragraph's report might look like the following. (The values for total demerits are fictional, and are chosen to make the rest of the exposition clearer. The right hand column will be explained later.)

```
4+5->4050 ; wibble 4050, 0 ;
4+6->4060 ; wibble 4060, 1 ;
4+7->4070 ; wibble 4070, 2 ;
          ; wobble ;
5+4->5040 ; wibble 5040, 0 ;
5+5->5050 ; wibble 5050, 1 ;
```

Given such a sequence of paragraph reports, and the requirement that there be, say, exactly 12 lines on each page, there is an associated optimisation problem. First, for each paragraph report choose one of its entries. Call this a selection (of paragraphs). Write the selection in the form

$$(5) + (3) + (4 + 7) + (5 + 2) + 10 + \ldots$$

and say that the selection is feasible if, when summing from left to right, successive exact multiples of 12 are reached during the progress of the sum. The above selection is feasible (as far as it goes). For every feasible selection, define the grand total demerits to be the sum of demerits associated with the terms of the form $(a+b)$. Thus, the $(4+7)$ terms contributes 4070 to the grand total demerits.

The optimisation problem is to find a feasible selection that minimises the grand-total demerits. This is one way (there are many others) of defining a global optimisation for the line and page breaks of a document. If such a problem is to be solved using discrete dynamic programming, the global optimisation data might take a more elaborate form, but the general structure will be the same. (The interested reader might wish to look at how TEX's line-breaking algorithm supplies demerits for adjacent lines whose looseness is visually incompatible. It is done by providing each partial problem with a context.)

It is both interesting and fortunate that the global problem, as described above, can be solved using TEX's line-breaking algorithm. It is a matter of 'putting the book on its side', and thinking of each line as a 'word' in a paragraph. The problem is to construct a suitable list of boxes, glue and penalties. So that we can get nice diagrams, we will let one pica represent one line.

Discardable items can vanish at line breaks, and with trickery this allows the problem to be solved. Consider for example the sequence of horizontal list items,

```
penalty "4070 " ;
kern "-2pc " ;
noalign {} ;
kern "2pc " ;
```

Kerns are discardable items. If the line break is taken at the penalty, the first kern will be discarded. The `noalign` is non-discardable, and it prevents the second kern from being discarded. Thus, if the penalty is not a break-point, the kerns cancel, but if the penalty is a break point, it effectively inserts a kern of two pica.

Denote such a sequence of horizontal list items by `wibble 4070, 2;`, and let `wobble` represent a `kern` by one pica. This procedure translates the sequence of paragraph reports into a horizontal list. When the line-breaking algorithm is applied to this list, with a `hsize` of twelve pica, the result is a global optimisation of the line and page breaks. If we are not typesetting on a grid, then some 'interword glue' (representing interline glue) should be added to the above construction.

As mentioned earlier, the line-breaking algorithm introduce penalties between the lines, in order to help TEX's page-breaking algorithm. These prevent, or discourage, page breaks just after the first line of a paragraph, and just before the last line. In the algorithm described here, the potential page break is part of the paragraph's specification. The club penalty thus becomes an extra demerit charged for paragraph specifications of the form $(1+n)$, and similarly the widow penalty applies to $(n+1)$.

Although there are some difficulties of a technical nature in implementing such a solution, there is a more fundamental problem. In 1989 [6], when Don Knuth released version 3 of TEX, he introduced several new primitives. One of them, the `\badness`, records the badness of the box that was most recently constructed. Thus, this quantity is made available to the macro programmer. Sadly, he did not at the same time introduce `\totaldemerits`, and so there is no ready access to this quantity.

Summary and conclusions

When Don Knuth announced [8] in 1990 that his work on developing TEX had come to an end, he pointed out that improved macro packages could be added on the input side, and improved device drivers added on the output side. This article shows that ten years after the event, there is still plenty of room for improvement on the macro package side. (However, the lack of a `\totaldemerits` command is unfortunate.)

The problem of suppressing hyphenation at the end of a page is relatively simple, particularly if a macro package such as Active TEX is used to construct the horizontal list. What has not been discussed is how to rearrange the resulting sequence of lines, so that the blank amongst them can be discarded. In abstract this is not difficult, but in the context of an existing macro package one may find assumptions being made that are inconsistent with this goal.

The problem of page make-up is much harder, particularly where there are multiple columns and floating material. TEX was not designed to do such work, although it can readily typeset the paragraphs that will go into the pages. As in shown in the previous section, it is possible to use the line-breaking algorithm to solve simple page make-up problems. For more complicated problems, an external program might be more suitable.

TEX is not good at complicated page makeup, but that is no reason to 'improve' it. Complicated page make-up was never a design goal of TEX. Instead, TEX can be used to feed paragraphs and paragraph reports to an external make-up program. Such can be thought of as an improved device driver, in the same way as Active TEX is intended to be an improved macro package.

Postscript

Prior to the TUG 2000 meeting I sent an earlier version of this paper to Don Knuth, and invited his comments. He told me that I should cite and read Michael Plass's thesis [10]. The citing is done, and I hope soon to read this work. He also says that he cannot add `\totaldemerits`, as that would mean changing TEX and suggests instead that I approach the authors of extensions to TEX.

In his essay on the errors of TEX, [7] Don Knuth wrote:

> Of course I don't mean to imply that all problems of computational typography have been solved. Far from it! There are still countless important issues to be studied, relating especially to the many classes of documents that go far beyond what I ever intended TEX to handle.

I hope that this article shows that a few judicious extensions to TEX will produce a new system that can handle well many new classes of documents, and that even TEX can make a fair attempt at doing the job. What seems to be required, above all, is an understanding of the problem, and the development of suitable algorithms. From then on, the programming of the extensions should be straightforward.

The article by Frank Mittelbach in these proceedings addresses a different aspect of the page makeup problem. His concern is with placement of floats. Combining his work with mine, even at the level of algorithms, is already a challenge. When it comes to implementation, the widespread use of active space characters is likely to present LATEX with many problems. Assumptions about category codes are built into its input syntax.

So much for output. On the input side the papers by David Carlisle and by Pedro Palao Gostanza in these proceedings have significant overlap with

this paper. I am delighted that others are taking steps in the direction of making all characters active. However, we now have three incompatible systems of values for the meaning of active letters and digits.

Active TEX provides a powerful and effective programming environment, especially for defining active characters. Without such a device, the programmer has to resort to ad hoc tricks, time and time again. For example, of the 1,936 lines of xmltex (v0.07), exactly 194 contain the string `catcode`. By contrast, of the 6,361 lines of my `sgmlbase` package (v0.00), exactly 23 contain the string `catcode`. Perhaps if Carlisle had used Active TEX, his work would have been easier.

For this area to flourish, standards are required. Without standards, incompatible versions of the basic macros will be re-invented. Application programmers will then have to work harder, to cope with this unhelpful diversity. There is also be the danger of schisms within the community.

To understand this, imagine what life would be like if there we used incompatible mechanisms for register allocation (`\newcount` and the like). In *The TEXbook* (page 346), Don Knuth addressed precisely this problem:

> *Allocation of registers.* The second major part of the `plain.tex` file provides a foundation on which systems of independently developed macros can coexist peacefully without interfering in their usage of registers.

We need the same for active characters. The packages `atcode.sty` and `atlatex.sty` have been written to be a fixed point that opens this area to the plain and LATEX macro programmer. They differ only in a small but significant detail (colon instead of prefix is used to segment the name space) from the version announced at TUG 1999.

I offer these packages to the community, and hope for the rapid and widespread adoption of a standard for the use of active characters. I would of course prefer that my own macros were the standard, but more important both to me and to the community as a whole, I believe, is that a standard acceptable to all is adopted.

References

[1] David P. Carlisle, *xmltex: A non validating (and not 100% conforming) namespace aware XML parser implemented in TEX*, these proceedings

[2] Jonathan Fine, Active TEX and the DOT input syntax, *TUGboat*, **20**, (1999), 248–254

[3] Pedro Palao Gostanza, Fast scanners and self-parsing in TEX, these proceedings

[4] Donald E. Knuth, Michael F. Plass, Breaking paragraphs into lines, *Software — Practice and Experience*, **11** (1981), 1119–1184.

[5] Donald E. Knuth, *The TEXbook*, Addison-Wesley (1984).

[6] ———, The new versions of TEX and METAFONT, *TUGboat*, **10** (3) (1989), 325–328

[7] ———, The Errors of TEX, *Software — Practice and Experience*, **19** (1989), 605–685; reprinted with additions and corrections as Chapter 10 of *Literate Programming*.

[8] ———, The future of TEX and METAFONT, *TUGboat*, **11** (4) (1990), 489.

[9] Frank Mittelbach, *Formatting documents with floats*, these proceedings

[10] Michael F. Plass, *Optimal Pagination Techniques for Automatic Typesetting Systems*, Ph.D. thesis, Stanford University (1981). Published also as Xerox Palo Alto Research Center report ISL-81-1