Working Paper Series
ISSN 1170-487X

Extracting Text from PostScript

by Craig Nevill-Manning, Todd Reed and Ian H. Witten

Working Paper 97/10
April 1997

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Extracting Text from PostScript

Craig G. Nevill-Manning, Todd Reed and Ian H. Witten

Computer Science Department, University of Waikato, New Zealand
{ cgn, treed, ihw} @cs.waikato.ac.nz

Correspondence should be addressed to:

Craig Nevill-Manning,

Biochemistry Department,

Stanford University,

Stanford, CA 94305-5307

Phone:	(415) 723-5976
Fax:	(415) 725-6044
Email:	cnevill@stanford.edu

Summary

We show how to extract plain text from PostScript files. A textual scan is inadequate because PostScript interpreters can generate characters on the page that do not appear in the source file. Furthermore, word and line breaks are implicit in the graphical rendition, and must be inferred from the positioning of word fragments. We present a robust technique for extracting text and recognizing words and paragraphs. The method uses a standard PostScript interpreter but redefines several PostScript operators, and simple heuristics are employed to locate word and line breaks. The scheme has been used to create a full-text index, and plain-text versions, of 40,000 technical reports (34 Gbyte of PostScript). Other text-extraction systems are reviewed: none offer the same combination of robustness and simplicity.

Keywords: PostScript interpreter, text extraction, digital libraries, full-text indexing, page layout

1 Introduction

It is often useful to be able to extract plain text from PostScript files. For example, the New Zealand Digital Library[*] contains a collection of computer science research reports that is full-text indexed; to create this index it is necessary to extract all the text from the document files that constitute the originals of the reports. Similar requirements for recovering plain text from the page description arise in many applications.

Converting PostScript to plain text is like an optical character recognition process for which the input is not "optical" but rather an electronic description of a page. That makes the problem sound trivial, for of course the characters need not be "recognized" because the document file represents them symbolically, as ASCII codes, rather than pictorially as ink on paper. However, it is complicated by the fact that PostScript is more than a page description language; it is a programming language.

[*] `http://www.cs.waikato.ac.nz/~nzdl`

Identifying the textual content involves more than conversion from one format to another—it involves predicting the output of a program.

Moreover, although sometimes it is just the stream of words that needs to be extracted (for example, for indexing purposes), other applications will need an approximation to the formatting information so that the text can be reproduced in different ways. A common requirement is to produce a (necessarily) approximate representation of a PostScript document in HTML, the hypertext mark-up language used in the World-Wide Web, and for this, structural features such as paragraph boundaries and font characteristics must be identified.

We describe a simple and robust scheme for extracting text from PostScript. Essentially, it involves modifying the behavior of a PostScript document so that, when executed, it produces ASCII text rather than pixels on a page. We also describe heuristics that improve the detection of word and line divisions by employing global information. Because of differences in character spacing, text with hard-coded line breaks that correspond to the layout of lines on the page exhibits grotesquely ragged right margins when displayed on a screen with a different font. Consequently we have developed simple heuristics to recognize paragraph boundaries, and to solve other practical problems with the output of the extraction process. However, we make no real attempt to reconstitute a good approximation to the original page layout and formatting; that lies beyond the scope of this article.

These techniques have been tested and proven to be robust on a collection of over 32,000 documents from a wide variety of sources (300 FTP sites) that constitute the computer science research collection in the above-mentioned digital library. Other public-domain PostScript extraction programs proved to be either very unreliable, or far too computation-intensive for our purposes. It is worth noting that our scheme is robust enough to cope with Adobe Portable Document Format (PDF) files* without any modification at all.

---

* see `www.adobe.com` for details

Extracting text from PostScript is challenging both from a fundamental computational perspective and from a pragmatic programming stance; Section 2 explains why. Section 3 describes how a small PostScript program can perform the basic task elegantly and reliably. However, it makes simplistic assumptions about line and character spacing, and Section 4 describes improvements to the basic algorithm which identify text boundaries more robustly and also solve some other practical problems that arise. Section 5 compares this technique to others and situates it in the spectrum of what is available in the public domain.

2 The problem with PostScript …

The programming language PostScript is designed to convey a description of a desired page to a printer[1]. Extracting text from PostScript files poses problems at two quite different levels. The first is at an abstract level. Because PostScript is a programming language, its output cannot, in principle, be determined by a textual scan. The second problem is a more pragmatic one. Although almost all actual PostScript documents have a fairly straightforward computational structure that makes the output text immediately available, it is fragmented and inextricably mixed with other text that does not form part of the desired output.

A problem in principle

In principle, the process of extracting text from a PostScript document is not one of format conversion, but rather a matter of predicting the output of a computer program. Consider the PostScript sample and output in Figure 1. This admittedly rather contrived example is a program that displays the sixth number of the Fibonacci sequence 1, 1, 2, 3, 5, 8, namely “8”. However, the character “8” appears nowhere in the input document: it is calculated recursively by the *fib* function. Any syntactic approach to conversion will be unable to accommodate such input and, to the extent that it occurs in practice, will be unreliable.

The only way to know the output of a program is to execute it. Thus any robust scheme for extracting text from PostScript must execute the document using an appropriate interpreter. Of course, the execution of the document must be altered to allow the text that is produced to be captured instead of rendered by placing pixels on the page. This paper describes just such a scheme.

A problem in practice

Most PostScript files do not contain code as subtle as the recursive function of Figure 1, and this may be regarded as a pathological case. Figure 2 shows excerpts from a more representative example of a document file, along with the text extracted from it. Characters to be placed on the page invariably appear as parenthesized strings in the source.

An obvious approach to text extraction, used in some conversion programs, is to extract and concatenate all such strings. There are two problems with this simple strategy. First, font names, file names, and any other internal string information is represented in the same way in the file; examples can be seen in the first few lines of Figure 2. Second, the division of text into words is not immediately apparent: spaces are usually implicit in the positioning of characters on the page. Text is often written out in fragments, so that each parenthetical string may only represent part of a word. Deciding which fragments should be concatenated into words is a difficult problem. Although simple heuristics might be devised to cover most common cases, this is unlikely to lead to a robust solution that can deal satisfactorily with the wide variety of files found in practice.

Another drawback to the general approach of analyzing parenthesized strings in PostScript files is that it requires modification to handle PDF files, which are compressed, and contain no readable text.

3 A simple text extractor

Our approach is to modify a document by prepending a small prefix which, when executed by a standard PostScript interpreter, has the effect of producing ASCII text in a file rather than pixels on a page. To execute the code, the modified document can be processed either by a previewer such as *ghostscript*, or by sending it to a printer. Details of the procedure are given in the Appendix.

The trick is to redefine the PostScript *show* operator, which is responsible for placing text on the page. Regardless of how a program is constructed, any text that appears on a page passes through this operator (or one of the five variants discussed below). We redefine *show* to write its argument, a text string, to a file instead of rendering it on the screen. When the document is executed a text file is produced instead of the usual set of physical pages.

We explain the operation by incrementally improving a simple program. Prepending the line */show { print } def*, shown in Figure 3a, to the document of Figure 2 redefines the *show* operator to print to standard output. The result is shown at the right of Figure 3a. One problem has been solved: separating what text is destined for a page from the remainder of the parenthesized text in the input file.

The question of identifying whole words from fragments must still be addressed. The text in Figure 3a contains no spaces, and needs to be segmented into words. Printing a space between each fragment results in the text in Figure 3b. Spaces do appear between each word, but they also appear within words, such as *m ultiple* and *imp ortan t*.

To detect where spaces should be inserted, it is necessary to consider how fragments are placed on the page. Figure 4 plots the distribution of distance between fragments, measured in printers' points,* in one technical report. The peak around zero corresponds to the fragments being part of the same word, while the broader hump around 16 corresponds to the inter-word spacing. The latter is spread out by the variation that occurs in word spacing. This is particularly striking when the margins are "justified" by adding extra space to pad out short lines. It seems reasonable from the graph to pick a threshold of about five points, below which fragments will be concatenated and above which spaces will be inserted.

The prologue in Figure 3c implements this modification. The variable *X* records the horizontal co-ordinate of the right-hand side of the previous fragment. The new *show* procedure obtains the current *x* coordinate using the *currentpoint* operator (pop

* There are 72 points to the inch.

discards the *y* coordinate), and subtracts the previous coordinate held in *X*. If the difference exceeds a preset threshold, in this case five points, a space is printed. Then the fragment itself is output.

In order to record the new *x* coordinate, the fragment must actually be rendered. The line *systemdict /show get exec* retrieves the original definition of *show* from the system dictionary and executes it with the original string as argument. This renders the text and updates the current point, which is recorded in *X* on the next line. The execution of the original *show* operator is one of the innovations of this technique over previous ones: it provides a simple and foolproof way of updating coordinates. This new procedure produces the text at the right of Figure 3c, where all words are segmented correctly. Line breaks can be detected by analyzing vertical co-ordinates in the same way and comparing the difference with a fixed threshold. There is one remaining problem in the text of Figure 3c—a *fl* ligature is represented by the § symbol—and we return to this in the next section.

The four variants of the PostScript *show* command, namely *ashow*, *widthshow*, *awidthshow*, and *kshow*, are treated similarly. A procedure is defined to do the work. It is called with two arguments, the text string and the name of the appropriate *show* variant. Just before it returns, the code for the appropriate command is located in the system dictionary and executed. Figure 3d shows the complete prologue.

Notwithstanding the use of fixed thresholds for word and line breaks, this scheme is surprisingly effective on the 40,000 technical reports in the technical report repository. In the next section, we describe several enhancements that further improve performance.

4 An improved text extractor

Although the solution developed in the previous section is elegant and robust, it has several minor problems that can be solved by introducing some heuristics. These problems include incorrect word-boundary identification when using very large or very small fonts, the need to distinguish paragraph and line breaks, dealing with non-ASCII characters, dehyphenation, and page reversal. We address each in turn.

They are implemented in the Python language[3] as a post-processor to the PostScript scheme just described. Because it is necessary to have access to detailed information on word and line spacing, a simple modification is made to the PostScript prologue in Figure 3d to provide this information.

Spacing in large and small fonts

The use of a fixed threshold to distinguish inter-word gaps from inter-fragment ones fails when the text is being printed in a very large or very small font. When using large fonts, as in titles, inter-fragment gaps are mistakenly identified as inter-word gaps, and words are broken up into individual letters. When using small fonts, as in footnotes, inter-word gaps are mistaken for inter-fragment gaps, and words are run together.

To solve this problem, we express the word break threshold as a fraction of the average character width. This width is calculated for the fragments on each side of the break by dividing the rendered width of the fragment by the number of characters in it. Examination of histograms of breaks as a fraction of character width leads to a threshold of 30%, which is effective for a wide range of type sizes. This modification also eliminates all dependence on the units in which the co-ordinate system is expressed.

Paragraph vs. line breaks

The line breaks in a PostScript document are designed for typeset text. Plain text does not always share the same line width, and it is often desirable to wrap lines differently depending on the situation—for example, the width of the window containing the text. Paragraph breaks, on the other hand, have significance for the document's content and should be preserved.

Paragraph breaks can be distinguished in two ways. The first presupposes that more space is left between paragraphs than between lines, as in Figure 5a. In this case, any break exceeding the average line space can be treated as a paragraph break. For "average line break" we use the most common non-trivial change in y coordinate throughout the document.

The second way separates paragraphs marked by indentation rather than line spacing. In Figure 5b, the first line of the second paragraph is indented from the left margin. This is often sufficient for identifying new paragraphs, but additional heuristics can reduce the number of mis-identifications. Consider the three lines numbered to the right of Figure 5b: one at the end of a paragraph and the next two beginning the subsequent paragraph. For these to signal a paragraph break,

- the second line must be indented relative to the other two;
- the second line must start with a capital letter;
- the third line must be longer than the other two;
- the second line must be almost as long as the third.

In the last two constraints it is the text length that is measured, not the distance between margins. The first line terminates a paragraph, and will not usually extend to the right margin, while the second one is indented. While certainly not infallible, these rules work well in practice.

Non-ASCII characters

When producing ASCII text from PostScript some character translation is inevitable because the character sets used in documents are much richer than the ASCII set. In technical documents, the most common use of unusual glyphs is in mathematical formulas. These include some ASCII characters—for example, variable names and digits—and some other symbols—Greek characters, integral signs and the like. The latter must either be approximated in ASCII, or deleted. We decided to flag unknown characters with a question mark because there can be no truly satisfactory character representation of mathematical formulae.

Other commonly-used glyphs include ligatures, bullets, and printer’s quotes. Ligatures are specially joined and kerned combinations of characters such as *fi* and *fl* that are rendered as a single glyph. This occurs very frequently in TEX output—indeed, an example appears in Figure 3—and since this is common in our environment we recognize such symbols and map them to their two-character equivalent. Bullets and printers’ quotes (“ ‘ ’ ” rather than ' ") are recognized in the same manner.

Dehyphenation

When documents are justified to a fixed right margin, words are often hyphenated. We reverse this process by detecting hyphens at the end of a line and concatenating the fragments on either side of the break. Admittedly this can remove the hyphen from a compound word that straddles a line break, but such situations are rare. Dehyphenation does not work across page breaks because headers and footers separate the last line of one page from the first line of the next.

Page reversal

The pages of PostScript documents often appear in reverse order. This is for mechanical convenience: when printers put pages face up on the output tray, the first page produced is the last page in the document. The Adobe document structure[2] defines a convention for specifying page ordering, but it is rarely followed in actual document files.

In order to detect page order, several heuristics were considered but rejected on grounds of unreliability. Our final solution is to extract numbers from the text adjacent to page breaks. These are usually page numbers, and one can tell if a document is reversed by checking whether the sequence is increasing or decreasing. This is fairly reliable because even though some numbers in the text are erroneously identified as page numbers, the decision is made on a global, majority, basis.

5 Other schemes

This system was developed as a solution to a practical problem that other systems failed to solve. Here we briefly describe similar schemes. The first two attempt extraction by a textual scan, while the remainder make use of a PostScript interpreter.

`ps2ascii.pl`	This is a widely-used Perl script[4] that extracts parenthesized text from a PostScript file.

`ps2txt`	This is a stand-alone C program that extracts parenthesized text from a PostScript file. It also includes some special code to deal with files generated by the *dvips* program.
`ps2a`	Written entirely in PostScript, this system requires *psh*, a component of the NeWS windowing system—although it could run using *ghostscript* or a printer. It is a rather complex program, and is optimized to perform with output from TeX. Like the system described in the present paper, it does execute the original *show* operators in the PostScript input.
`pstotext`	This system, from Digital Equipment Corporation, includes components in both PostScript and C. Although very comprehensive, is it correspondingly complex, and extremely slow to execute. It succeeds in tracking font changes and does an excellent job of conversion.
`ps2ascii`	This program, which is included with the standard *ghostscript* distribution, is complex but, in our experience, not at all robust.
`ps2ascii` [ucf]	This variant of the *ghostscript* `ps2ascii` was developed at the University of Central Florida. It includes components in both PostScript and Perl.
`ps2ascii` [jhu]	This variant of `ps2ascii` was developed at Johns Hopkins University to convert journal articles to HTML. It strives to preserve the formatting of the original PostScript document. However, it is designed exclusively for documents with a format specific to certain journals of the Johns Hopkins University Press, and recognizes—and gives special attention to—PostScript files that were generated with a specific software package (QuarkXPress). A table containing numerous parameters is used to aid conversion, and can, in principle, be modified for new formats.

6 Conclusions

We have presented a simple, elegant and robust scheme for extracting text from PostScript. The heuristics it uses to reconstitute word spacing and paragraphing, and to restore other small aspects of the text, work well although they are obviously not infallible. Quantitative evaluation is not really meaningful because everything depends on how the PostScript is generated in the first place, but a large-scale sample from a wide variety of sources can be viewed in the New Zealand Digital Library collection of computer science research reports.[*]

In future we will address the extraction of further structural information from layout, ranging from section headings to bibliography entries. We anticipate that deficiencies will inevitably continue to emerge as new documents, with new layout styles, are encountered, and we will have to address them by evolving the system constantly. In order to prevent it developing into a large, unstructured tangle of heuristic hacks, which will rapidly become unmaintainable, we propose to investigate the application of machine learning techniques to infer a rule set from example reports. Whenever a new format is encountered, we would augment the set of training examples to include it and relearn the set of rules from scratch. Meanwhile, our heuristic system already does a good job of extracting text and rudimentary structure from PostScript documents.

References

1. Adobe Systems Incorporated (1985) *PostScript language reference manual*. Addison Wesley, Reading, Massachusetts.

2. Adobe Systems Incorporated (1989) *Document structuring conventions*. Addison Wesley, Reading, Massachusetts.

3. Lutz, M. (1996) *Programming Python*. O'Reilly and Associates, Sebastopol, California.

4. Wall, L., Christiansen, T. and Schwartz, R.L. (1996) *Programming Perl*. O'Reilly and Associates, Sebastopol, California.

[*] `http://www.cs.waikato.ac.nz/~nzdl`

Appendix: Obtaining the extraction system

The simplest way to employ this scheme is to copy the prologue from Figure 3d, prepend it to a PostScript file by hand, and run it through a PostScript interpreter. *Ghostscript* is an excellent public-domain interpreter, and is available from `www.cs.wisc.edu/~ghost`. Once it is installed, save the prologue above in a file called *prescript.ps*, and type

```
% gs -q -dNODISPLAY -soutfile=outfile prescript.ps infile.ps -q
```

where *infile.ps* is the PostScript file, and the text is to go into *outfile*.

An intriguing option exists on a Macintosh, or any other system that provides a mechanism for receiving data from a direct connection to a PostScript printer: the interpreter in the printer itself can be used. On the Macintosh, use BBEdit (`www.barebones.com/bbedit.html`) to open the PostScript file, paste in the prologue, send the file to the printer, and capture the returned text. Because the *showpage* operator is redefined, the printer will not output any pages.

To take advantage of the improvements described in Section 4, download the *prescript* system from the 'technology' link on the New Zealand Digital Library page (`www.cs.waikato.ac.nz/~nzdl`). This consists of an augmented prologue, *prescript.ps*, and a Python script, *prescript.pl*, that implements the heuristics described in Section 4.

Figure Captions

Figure 1 A PostScript program that prints a character that does not occur in the text of the program
(a) program
(b) rendered output

Figure 2 A PostScript document and the text extracted from it

Figure 3 Simple and complex ways of producing text from PostScript
(a) printing all fragments rendered by *show*
(b) putting spaces between fragments
(c) putting spaces between fragments more than 5 points apart
(d) extension to other PostScript *show* operators

Figure 4 Distribution of inter-fragment distances

Figure 5 Two styles for paragraph breaks
(a) additional space
(b) indentation

a

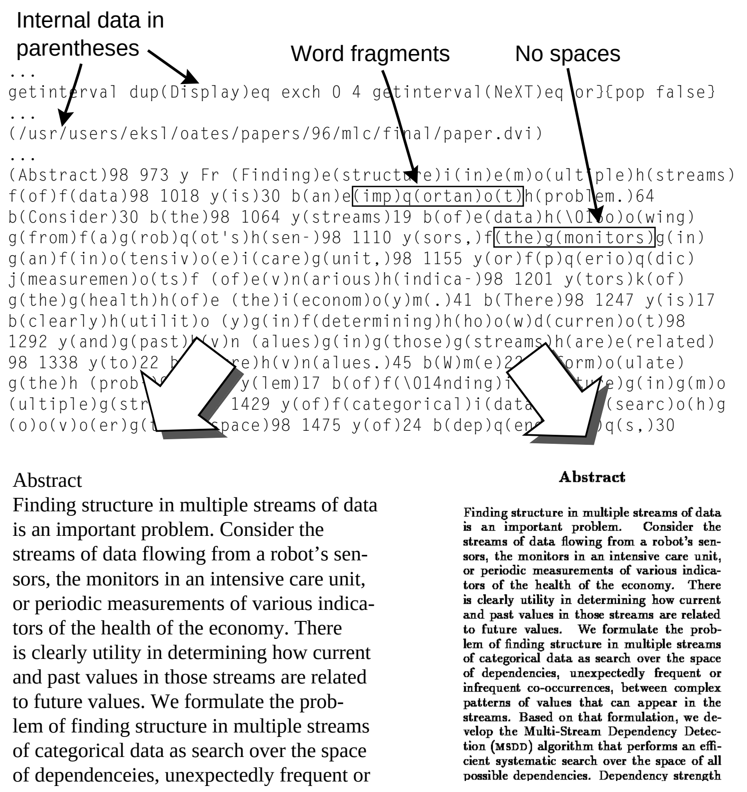
```
/fib {
  dup
  1 eq { }
      { dup 2 eq { pop 1 }
                 { dup 1 sub fib exch 2 sub fib add }
         ifelse }
  ifelse
} def

(Fib[6] = ) show 6 fib (        ) cvs show
```

b

Fib[6] = 8

Figure 1 A PostScript program that prints a character that does not occur in the text of the program
(a) program
(b) rendered output



```
...
getinterval dup(Display)eq exch 0 4 getinterval(NeXT)eq or}{pop false}
...
(/usr/users/eksl/oates/papers/96/mlc/final/paper.dvi)
...
(Abstract)98 973 y Fr (Finding)e(structure)i(in)e(m)o(ultiple)h(streams)
f(of)f(data)98 1018 y(is)30 b(an)e(imp)q(ortan)o(t)h(problem.)64
b(Consider)30 b(the)98 1064 y(streams)19 b(of)e(data)h(\01o)o(wing)
g(from)f(a)g(rob)q(ot's)h(sen-)98 1110 y(sors,)f(the)g(monitors)g(in)
g(an)f(in)o(tensiv)o(e)i(care)g(unit,)98 1155 y(or)f(p)q(erio)q(dic)
j(measuremen)o(ts)f (of)e(v)n(arious)h(indica-)98 1201 y(tors)k(of)
g(the)g(health)h(of)e (the)i(econom)o(y)m(.)41 b(There)98 1247 y(is)17
b(clearly)h(utilit)o (y)g(in)f(determining)h(ho)o(w)d(curren)o(t)98
1292 y(and)g(past)h(v)n (alues)g(in)g(those)g(streams)h(are)e(related)
98 1338 y(to)22 b      re)h(v)n(alues.)45 b(W)m(e)22     orm)o(ulate)
g(the)h (prob         y(lem)17 b(of)f(\014nding)i         ture)g(in)g(m)o
(ultiple)g(str        1429 y(of)f(categorical)i(data        (searc)o(h)g
(o)o(v)o(er)g(t       space)98 1475 y(of)24 b(dep)q(en        )q(s,)30
```

Abstract
Finding structure in multiple streams of data
is an important problem. Consider the
streams of data flowing from a robot's sen-
sors, the monitors in an intensive care unit,
or periodic measurements of various indica-
tors of the health of the economy. There
is clearly utility in determining how current
and past values in those streams are related
to future values. We formulate the prob-
lem of finding structure in multiple streams
of categorical data as search over the space
of dependenceies, unexpectedly frequent or

Abstract

Finding structure in multiple streams of data is an important problem. Consider the streams of data flowing from a robot's sensors, the monitors in an intensive care unit, or periodic measurements of various indicators of the health of the economy. There is clearly utility in determining how current and past values in those streams are related to future values. We formulate the problem of finding structure in multiple streams of categorical data as search over the space of dependencies, unexpectedly frequent or infrequent co-occurrences, between complex patterns of values that can appear in the streams. Based on that formulation, we develop the Multi-Stream Dependency Detection (MSDD) algorithm that performs an efficient systematic search over the space of all possible dependencies. Dependency strength

Figure 2 A PostScript document and the text extracted from it

a

```
/show { print } def
```

```
Findingstructureinmultiplestreamsofdataisanimportant
problem.Considerthestreamsofdata§owingfromarobot'
ssensors,themonitorsinanintensivecareunit,orperiodi
cmeasurementsofvariousindicatorsofthehealthofthee
conomy.Thereisclearlyutilityindetermininghowcurrenta
ndpastvaluesinthosestreamsarerelatedtofuturevalues
```

b

```
/show { print ( ) print } def
```

```
Finding structure in m ultiple streams of data is an imp
ortan t problem. Consider the streams of data §o wing
from a rob ot's sensors, the monitors in an in tensiv e
care unit, or p erio dic measuremen ts of v arious
indicators of the health of the econom y . There is
clearly utilit y in determining ho w curren t and past v
alues in those streams are related to future v alues
```

c

```
/X 0 def

/show {
  currentpoint pop
  X sub 5 gt { ( )  print } if
  dup print
  systemdict /show get exec
  currentpoint pop /X exch def
} def
```

```
Finding structure in multiple streams of data is an
important problem. Consider the streams of data
§owing from a robot's sensors, the monitors in an
intensive care unit, or periodic measurements of
various indicators of the health of the economy. There
is clearly utility in determining how current and past
values in those streams are related to future values.
```

d

```
/X 0 def

/protoshow {
  currentpoint pop
  X sub 5 gt { ( )  print } if
  dup print
  systemdict exch get exec
  currentpoint pop /X exch def
} def
```

```
/show       {/show       protoshow} def
/kshow      {/kshow      protoshow} def
/widthshow  {/widthshow  protoshow) def
/ashow      {/ashow      protoshow} def
/awidthshow {/awidthshow protoshow} def
```

Figure 3 Simple and complex ways of producing text from PostScript
(a) printing all fragments rendered by *show*
(b) putting spaces between fragments
(c) putting spaces between fragments more than 5 points apart
(d) extension to other PostScript *show* operators

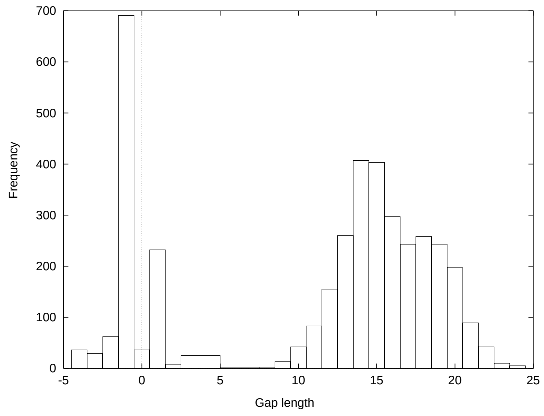
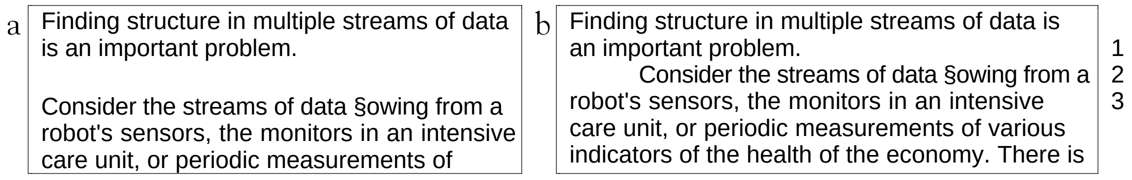


Figure 4 Distribution of inter-fragment distances



a Finding structure in multiple streams of data
is an important problem.

Consider the streams of data §owing from a
robot's sensors, the monitors in an intensive
care unit, or periodic measurements of

b Finding structure in multiple streams of data is
an important problem. 1
Consider the streams of data §owing from a 2
robot's sensors, the monitors in an intensive 3
care unit, or periodic measurements of various
indicators of the health of the economy. There is

Figure 5 Two styles for paragraph breaks
(a) additional space
(b) indentation