Julius-Maximilians-Universität Würzburg
Institut für Informatik
Lehrstuhl für Informatik I

Master's Thesis

Boundary Labeling for Annotations in Texts

Fabian Lipp

August 19, 2014

Supervisors:
Prof. Dr. Alexander Wolff
Dipl.-Inform. Philipp Kindermann

Abstract

We present a tool for annotating Latex documents with comments. Our annotations are placed in the left, right, or both margins, and connected to the corresponding positions in the text with arrows (so-called *leaders*). Problems of this type have been studied under the name *boundary labeling*. We consider various leader types (straight-line, rectilinear, and Bézier) and modify existing algorithms to allow for annotations of varying height. Our algorithms draw the leaders without crossings, some of them try to minimize the total leader length. We have implemented our algorithms in Lua; they are available for download as an easy-to-use Lualatex package. This package is designed modular so that it can easily be extended by new algorithms.

Zusammenfassung

Wir stellen eine Möglichkeit für Latex vor, mit der Dokumente mit Kommentaren versehen werden können. Die Anmerkungen werden im linken, rechten oder in beiden Seitenrändern platziert und durch eine Linie mit der entsprechenden Position im Text verbunden. Dieses Problem ist unter dem Namen *boundary labeling* bekannt. Wir untersuchen verschiedene Typen von Verbindungslinien (geradlinig, rechtwinklig und Bézier-Kurven) und verändern bestehende Algorithmen, um Anmerkungen mit unterschiedlichen Höhen zu unterstützen. Unsere Algorithmen zeichnen die Linien ohne Kreuzungen, einige davon versuchen deren Gesamtlänge zu minimieren. Die Algorithmen sind in Lua implementiert. Sie sind als Lualatex-Paket verfügbar, das einfach eingesetzt werden kann. Dieses Paket ist modular gestaltet, so dass es um neue Algorithmen erweitert werden kann.

Contents

1. Introduction **4**

2. Implementation **7**
2.1. Actions when a `\todo` command occurs 7
2.2. Actions on page shipout 8
2.3. Package options 9

3. Algorithms for Label Placement **10**
3.1. *s*-leaders 10
3.2. Bézier curves as leaders 12
3.3. *opo*-leaders and *os*-leaders 12
3.4. *po*-leaders 13

4. Improvements **16**
4.1. Label clustering 16
4.2. Two-sided label placement 17
4.3. Highlighting portions of text 17

5. Experimental Results **19**

6. Conclusion and Open Problems **21**

A. Example Documents **23**

1. Introduction

Many word processing systems support annotations for the text. The most common case for this annotations are comments, which can be inserted in arbitrary positions inside the text. The comments themselves are placed as *labels* in the margin next to the text and connected to the corresponding position, called *site*, by a line called *leader*. The endpoint of a leader at a label is called a *port*. Such comments are available, for example, in LibreOffice (see Figure 1.1) and Microsoft Word. This task can be expressed in the boundary labeling notion introduced by Bekos et al. [BKSW07]: the sites to be annotated lie inside the text area and the labels are to be placed outside the text area. They describe several types of leaders, such as straight-line leaders (*s*-leaders), rectilinear leaders with one bend (*po*-leaders) and rectilinear leaders with two bends (*opo*-leaders).

Previous work. Boundary labeling has been extensively investigated in the last few years, see a survey on the interaction between cartography and graph drawing [Wol13]. For labels of uniform size, the problem is well-studied. Most algorithms try to minimize the total leader length. For *s*-leaders, it suffices to compute a minimum-weight perfect matching, which can be done in $O(n^{2+\varepsilon})$ time [AES99]. For *opo*-leaders, Bekos et al. [BKSW07] gave three different algorithms for the number of sides used by the labels, with running times $O(n \log n)$ (one-sided), $O(n^2)$ (two-sided), and $O(n^2 \log^3 n)$ (four-sided). Further, they presented an $O(n^2)$-time algorithm for *po*-leaders that lie on one side or on two opposite sides of the text. The result for *po*-leaders was improved by Benkert et al. [BHKN09] for the one-sided case. They gave an $O(n \log n)$-time algorithm for length minimization and an $O(n^3)$-time algorithm for a very general class of objective functions, including, for example, bend minimization. They also studied leaders that contain a diagonal part and gave an $O(n^2)$-time algorithm for the one-sided case. This result was extended by Bekos et al. [BKNS10] to more than one side. Recently, Kindermann et al. [KNR+13] gave the first efficient algorithms for *po*-leaders that decide whether an instance with labels on two adjacent, three, or four sides has a crossing-free solution (and, if yes, compute one).

Boundary labeling for non-uniform labels is still largely unexplored. Bekos et al. [BKPS06] showed that it is NP-hard to find a crossing-free labeling if the labels have to be placed on two sides (or two stacks on the same side). Huang et al. [HPL14] considered a version of the problem that is always feasible: labels are placed into the right margin or into both margins, which are not bounded from below or above. For this model, *opo*-leaders, and labels of non-uniform size, they gave an $O(n^3)$-time algorithm that minimizes the total leader length in the one-sided case. For the two-sided case, they showed NP-hardness.

Available Latex packages. In this thesis, we focus on comments for Latex documents. There are some packages that support the placement of textual comments in the margin,

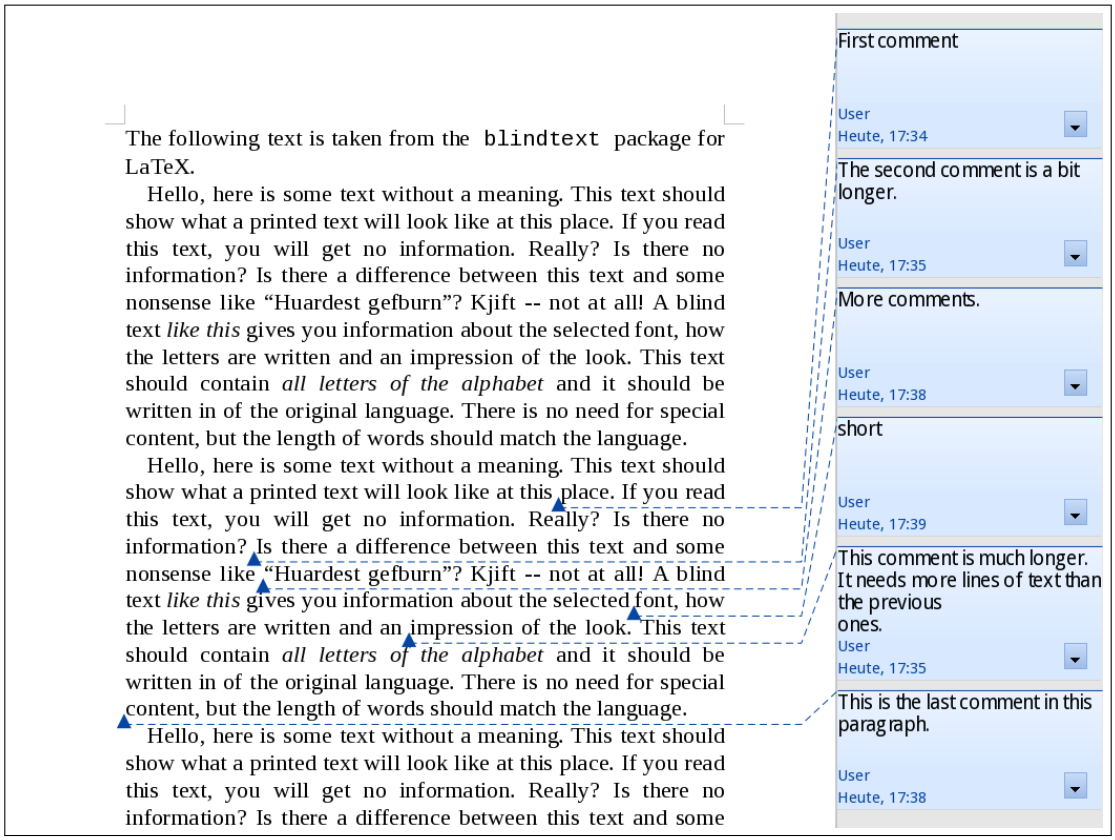


Figure 1.1.: Screenshot of comments in a document in LibreOffice 4.1.5.

namely `todonotes` [Mid12], `fixme` [Ver13] and `fixmetodonotes` [Bar13]. They have in common that they use Latex's `\marginpar` command to print the note as soon as the corresponding command is encountered in the source of the document. The drawback of this approach is that the positions of the following comments are not known and cannot be considered when placing a note. The first label is placed beside the first site, and the following ones are placed below. Often it happens that a lot of free space is wasted above the topmost label, while the bottommost label is only partially visible (if at all), see Figure A.1 in the appendix. Another disadvantage is that the `\marginpar` method cannot be used inside floating environments such as tables or algorithms. While the packages `fixme` and `fixmetodonotes` do not draw any leaders, `todonotes` uses *opo*-leaders. With this leader style it is hard to match a note to its corresponding site in the text when there are many comments in a short piece of text. A similar problem occurs with the leader style used by LibreOffice; see Figure 1.1.

Other Latex packages support annotations as metadata for PDF documents, for example, `pdfcomment` [Kle12]. The drawback of this package is that the user needs a compatible PDF viewer and that the annotations cannot be printed with the text. Packages such as `easy-todo` [RV14] don't place annotations in the margins, but insert a

marker into the text and list all comments at the end of the document. With all of these packages the notes refer to a single point in the text only, not to a portion of the text, which should be annotated.

Our contribution. Our approach is different from all those listed above in that we collect the comments for a whole page and then compute a good placement for the labels. Of course, this computation needs more resources than the ad-hoc placement of the existing packages. Additionally, our Latex package supports different leader types, which the user can select when loading the package. We explain some of the technical details of the package in Section 2.

As the heights of our labels vary with the length of the texts placed inside them we cannot use the algorithms for uniform labels found in the literature. We give several algorithms for non-uniform labels, most of which are extensions of existing algorithms for the uniform case; see Section 3. We improve upon these basic algorithms by considering label clustering, the two-sided case, and annotating portions of text instead of single points inside the text; see Section 4. We have implemented all of our algorithms and have evaluated them experimentally; see Section 5. We conclude with some open problems; see Section 6.

The Latex package is available for download on CTAN [1]. Included in the package is an extensive documentation describing the available commands and options.

[1] `http://ctan.org/pkg/luatodonotes`

2. Implementation

We have implemented the algorithms in Lua and have bundled them into a package for Lualatex, which we call `luatodonotes`. The package requires the modern Tex-processor Luatex [HHH], which allows us to embed Lua code inside our Tex sources. This gives us access to a high-level programming language for implementing our label-placement algorithms. From the user's point of view, this does not change much. Luatex is part of every modern Tex installation, for example, Tex Live. Assuming such an installation, the difference in usage is simply that instead of calling (`pdf`)`latex`, the user calls `lualatex`.

The Lua engine in Luatex provides special objects to access the Tex engine. At any time, we can write some text to Tex, which is then interpreted as if it occurred in the input file. Additionally, we have methods to access the data structures used by Tex. For example, it is easy to read and modify the values of Tex counters and dimensions. With more involved functions we can read the contents of boxes (the basic structure out of which pages are built in Tex) and manipulate them. We are even able to install our own functions in the Tex workflow: for example, we can write a custom line breaking algorithm in Lua that is used instead of the original one.

Moreover, Luatex implements some modern features missing in classical Tex interpreters: For example, it supports Unicode without requiring additional packages and can use advanced features of modern OpenType fonts.

Our package is based on the `todonotes` package (see Section 1). It is downward compatible as it provides the same commands and options to the user as the original package. Usage is quite simple: the user loads the package with the command `\usepackage{luatodonotes}` and inserts a comment into the text with the command `\todo{comment text}`. Additionally we provide the macro `\todoarea` to annotate a portion of text with a comment, which is described in Section 4.3. In the following sections we explain how our package works.

2.1. Actions when a `\todo` command occurs

Wherever the user inserts a `\todo` command in the text, we store its position and its argument (that is, the comment) in a Lua list, but we do not print anything in this moment. Additionally, we store some of the options given to the command (for example, the colors for the label and the leader) and the font size in the current paragraph. We need this information later when drawing the labels and leaders.

There is no obvious way to store the comment in our Lua data structures for later use. When the argument contains Latex macros, the user expects our package to see their current value in the note. If the macro is redefined later the original value is lost and

we would print the new value. So we need to expand the macros at the time the `\todo` command appears in the text. To achieve this we write the text into a temporary box, which is later accessed by Lua. Unfortunately, there are some macros that are not fully expandable and that will not work as expected in `\todo` commands. This can apply to some commands changing the formatting of the notes (for example, the font size or line spacing).

These difficulties do not appear with the classical `todonotes` package as the comment is written instantly when it occurs and has not to be saved for later.

We expect that the values of options can be fully expanded as they only contain a string (like the name of a color). Thus we can write them into a temporary token register. We can read its contents from Lua and store them in a variable.

2.2. Actions on page shipout

When a page is finished ("shipped out" in Tex terminology), we compute the position of the labels and draw them. Before calling our label-placement algorithm, we have Tex determine the label heights. For that, we take the contents of the box that was saved for the note and put them into a box with the required width. With this the text is broken into lines and we can write the height of the box into a Tex dimension, which we can read in Lua later.

To determine the absolute positions of the sites, we use PGF/Ti*k*Z [Tan], a widely used Tex package for producing vector graphics. This package can locate the position of a site on the page where the `\todo` command was inserted using the `remember picture` option, even when the command occurs inside a floating environment (such as a figure or a table).

For each label, the placement algorithm computes the absolute coordinates on the page on which the label is to be placed. Then, we use Ti*k*Z to draw the labels and the leaders that connect the labels with their corresponding sites in text. Finally, a mark is placed at each site. This modular design simplifies the implementation of new algorithms and makes the package extensible. Placement algorithms and leader drawing algorithms are implemented as simple Lua functions. The user can specify an optional parameter for the Latex package to select which functions should be used.

The size and position of the rectangles that contain the label texts depend on the current page layout. Our Lua code reads the relevant Tex dimensions to compute the position of these areas. We provide options to control the distances between the labels and the text (`distanceNotesText`) and between the labels and the border of the page (`distanceNotesPageBorder`). The algorithms can place labels in the left and in the right margin (see Section 4.2), but a margin is used only if it is wide enough to accomodate a label, that is, if the label can be at least of width `minNoteWidth`. Our algorithms assume that the margin is free of other elements when placing the labels. When marginal notes (placed with `\marginpar`) are used in the document they can conflict with our notes. The classical `todonotes` package does not have this issue as it uses `\marginpar` itself to place its labels.

2.3. Package options

When loading the package with `\usepackage{luatodonotes}`, optional arguments can be specified in square brackets. The most relevant options are (a) the algorithm for label placement (`positioning`) and (b) the leader type (`leadertype`). Other options control the layout: the minimum vertical distance of the labels (`interNoteSpace`), the distance from the contents of the label to its border (`noteInnerSep`) and the color of the leaders (`linecolor`). Moreover, there are options that only apply for certain algorithms. They are explained at the appropriate place in Section 3.

Additionally, we accept all options supported by `todonotes`. For example, `bordercolor` and `backgroundcolor` can be used to control the colors of the notes. We can even set options for individual notes, for example, to change their textsize or color.

3. Algorithms for Label Placement

In the following, the algorithms are categorized by the leader type that they support. In principle, our package allows the user to combine any label-placement algorithm with any leader type. Still, some algorithms have been designed with certain leader types in mind. Other combinations will probably yield unwanted results, such as label overlap or crossing leaders.

In the descriptions of our algorithms below, we assume that labels are placed on the left side of the text, but this is not a restriction of our actual implementations. Additionally, we try to place the labels without gaps between them, while in reality we want to preserve a certain minimum distance between them. Clearly, this is easy to achieve.

3.1. s-leaders

Our algorithms designed for s-leaders have a common property: they draw the leaders without crossing each other. Their common objective is to place the labels one below the other on the boundary while avoiding gaps between them. They differ in the position of the ports, that is, the position on the label boundary to which the leader is attached. A pleasant position for the port would be the center of the right side of the label. Unfortunately, we don't have an algorithm that can place the labels without gaps using this port position. We don't even know whether every instance of site positions and label heights is feasible with respect to these criteria; see Section 6.

We don't give algorithms that minimize the total leader length here, but concentrate on drawings without crossings. The clustering approach described in Section 4.1 can decrease the leader length as labels are placed closer to their corresponding sites.

NorthEast. We use an algorithm of Bekos et al. [BKSW07] for fixed labels, which can easily be adopted to our problem with labels of non-uniform heights: The upper right corner of each label is used as its port. The labels are placed consecutively from the top of the page to the bottom. In each step, we emit a ray from the port of the next label vertically to the top and rotate it clockwise until the first unlabeled site is hit. Obviously, by connecting this site to a label at the current position, we don't hide any other sites and can label the remaining sites without crossings.

NorthEastBelow. This algorithm is based on the preceding one. The difference is that we lower the port from the corner by a constant offset. In our opinion the result looks better if the leader is not attached directly at the corner. A good value for this offset is half of the height of the smallest label. As we know the position

for each port while placing the label, we can still use the ray construction of the preceding algorithm to place the labels without spaces between them.

East. In this algorithm the port of every label is located at the center of its right side. When we try to find the next unlabeled site to be labeled, we do not know the port position as it depends on the height of the label. Therefore, we cannot use the ray construction from the previous algorithms. Algorithm 3.1 is a heuristic that guarantees crossing-free leaders while trying to avoid gaps between the labels. It can usually handle real-world inputs without additional gaps.

An instance that is not handled optimally by the heuristic is depicted in Figure 3.1. The sites can be labeled without gaps when placing the labels in the order 2, 1, 3. As mentioned above it is an open question if this is possible for all instances.

Algorithm 3.1: Placing labels using east anchors

Input: $p_1, \ldots, p_n$ are the sites in the text
Output: y-coordinate $y_1, \ldots, y_n$ of the top edge of each label

$P \leftarrow \{p_1, \ldots, p_n\}$
$L \leftarrow [\,]$ `// list contains labels in the order in which they have been placed`
$lastY \leftarrow 0$
while $P \neq \emptyset$ **do**
  $H \leftarrow \{height(p_i) \mid i = 1, \ldots n\}$
  foreach $h \in H$ **do** `//` H `sorted ascending`
    emit a ray from the port of a label of height h placed directly below the last label
    $i \leftarrow$ index of first point in P that is hit by the ray when rotated clockwise
    if $height(p_i) \leq h$ **then**
      break
  $y_i \leftarrow lastY - (h - height(p_i))/2$
  L.add(p_i)
  $P \leftarrow P - \{p_i\}$
  $lastY \leftarrow y_i - height(p_i)$
`// Postprocessing: try to shrink gaps`
foreach $l \in L$ **do**
  if there is a gap above l **then**
    move l up as far as possible without creating any new intersection between leaders

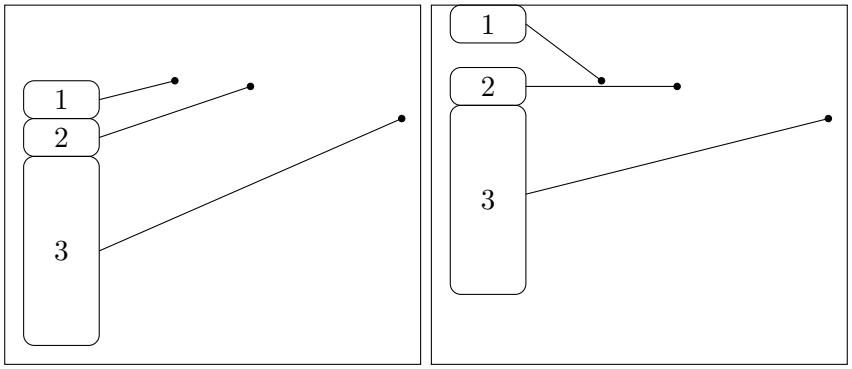


Figure 3.1.: An instance where the East algorithm does not yield a drawing without gaps. Left: label positions before postprocessing; Right: after postprocessing.

3.2. Bézier curves as leaders

We base our Bézier curves on *s*-leaders using a force-directed algorithm described by Fink et al. [FHS+12]. We use cubic Bézier curves that are required to enter the port at the label horizontally. This means that the first control point has to stay on the same horizontal line as the port and can only be moved to the left or the right. The second control point is always placed in the center between the first control point and the site.

In the first iteration of the algorithm, the control points are placed on the endpoints of the leader, that is, it starts as a straight line. Later, the first control point of each curve is moved by applying forces to it. We use a force that pulls the control point to its optimal point, which is computed beforehand and usually yields a good-looking curve. Other forces try to increase the distance between curves. In every iteration the forces on every point are limited by the distance to the nearest curve to inhibit new intersections between leaders. Therefore, the algorithm guarantees crossing-free Bézier curves when starting with straight-line leaders without intersections.

The runtime of this algorithm is dominated by the calculation of the distances between each pair of curves. This calculation is done by an approximation of the curves. We need the distances to update the forces in every iteration.

3.3. *opo*-leaders and *os*-leaders

Positioning the labels for crossing-free *opo*-leaders is simple as Bekos et al. [BKSW07] show: we place the labels in the order given by the y-coordinates of their sites. Sites with identical y-coordinates are processed from left to right. The vertical parts of the leaders are drawn in the *track routing area*, that is, the vertical strip between text and labels. The width of this track routing area is specified using the option `routingAreaWidth` of the package. We use Algorithm 3.2 to split the labels into groups, with labels sharing a common vertical segment being put in the same group. This can be done by a simple linear-time algorithm. Thus the vertical segments of the leaders in each group must be

Algorithm 3.2: Drawing *opo*-leaders

```
1  G ← ∅ // set of all groups
2  G ← ∅ // current group
3  lastDir ← nil
4  foreach note n do // from top of page to bottom
5      if port for label of n lies above the corresponding site in text then
6          dir ← down
7      else if port for label of n lies below the corresponding site in text then
8          dir ← up
9      else
10         dir ← nil
11     if dir = lastDir and
       ((dir = down and port of label n lies above the previous label's site in text) or
       (dir = up and site of label n lies above of he previous label's port)) then
12         G ← G ∪ {n}
13     else
14         G ← G ∪ {G}
15         G ← {n}
16         lastDir ← dir
```

placed side by side. We draw the vertical segments in one group with equal distances between them, using the whole width of the track routing area.

The algorithm is even easier for *os*-leaders, a leader style that was not discussed until now. We list it here because this is the style that, for example, LibreOffice uses (see Figure 1.1). Labels are placed in the same order as for *opo*-leaders. For the leaders, we connect the site with a horizontal line segment that extends to a fixed x-coordinate inside the margin. Then we connect the end of the horizontal segment to the label's port with a straight-line segment.

3.4. *po*-leaders

Benkert et al. [BHKN09] developed an algorithm to compute an optimal crossing-free labeling using *po*-leaders with respect to an arbitrary badness function. This algorithm, which uses a dynamic programming approach, is designed for uniform labels only. It needs $O(n^3)$ running time and $O(n^2)$ space.

For our application, we extend the algorithm of Benkert et al. to non-uniform labels. To be able to work with the arbitrary heights of the labels, we need to raster the page, that is, we define the y-coordinates on which labels may be placed. Our algorithm yields a labeling respecting this raster with minimum total leader length. The height of the raster can be chosen using the parameter `rasterHeight` of the Latex package. The port for each label can be chosen arbitrarily. In the following, the ports are fixed to the center

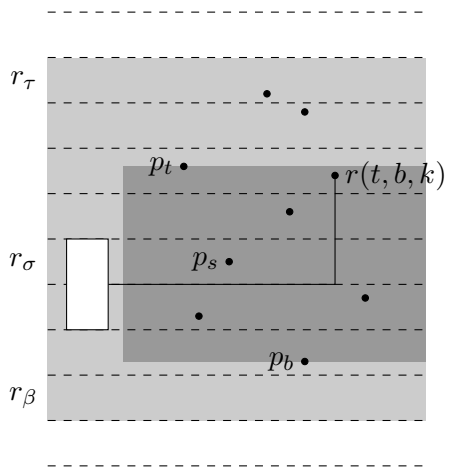


Figure 3.2.: The labeling problem for $T[t, b, \tau, \beta, k]$ is split into two independent subproblems by fixing the label position of $r(t, b, k)$. The dashed lines show the raster slots. The light gray area indicates the slots from r_τ to r_β. The dark gray area shows the sites between p_t and p_b.

of the right side of the labels.

Let $p_1, \ldots, p_n$ denote the sites from top to bottom and let $r_1, \ldots, r_m$ be the slots obtained by rasterizing the page from top to bottom. We use a 5-dimensional table in our dynamic program. The entry $T[t, b, \tau, \beta, k]$ represents the minimum length of a labeling of the k leftmost sites in $\{p_t, \ldots, p_b\}$ using only the raster slots $r_\tau, \ldots, r_\beta$. The labels must lie completely inside the given slots.

Let $r(t, b, k)$ the k-th point from the left in the set $\{p_t, \ldots, p_b\}$. The length of the shortest *po*-leader from the site p to its corresponding label beginning in slot r_σ is denoted by $l^*(p, \sigma)$. The entries of the table are computed using the following decomposition (illustrated in Figure 3.2):

$$T[t, b, \tau, \beta, k] = \min_{\text{feasible } \sigma \in \{\tau, \ldots, \beta\}} l^*(r(t, b, k), \sigma) + T[t, s, \tau, \sigma - 1, k_1] + T[s + 1, b, \sigma + h, \beta, k_2]$$

In this formula p_s is the lowest point that lies above the leader arm (the horizontal part of the leader), when the label for $r(t, b, k)$ is placed at slot r_σ. Let h the height of this label. The number of sites from $\{p_t, \ldots, p_b\}$ lying left of $r(t, b, k)$ and above resp. below the leader arm is denoted by k_1 resp. k_2.

A position for the label is feasible, if both partial solutions (above and below the leader arm) are feasible, that is, there are enough slots to label the contained sites.

Clearly, $T[1, n, 1, m, n]$ is the optimal labeling of the whole instance. With this algorithm we can compute an optimal solution in $O(n^4 m^3)$ time with $O(n^3 m^2)$ space, where n is the number of sites to be labeled and m is the number of slots in the raster on the page.

The quality of the result depends on the chosen raster height. For example, when the height of the labels is slightly greater than the raster, there will be large gaps between

them. We cannot reduce the raster height arbitrarily because this will strongly increase the running time and memory consumption.

Avoid overlappings with text lines. The algorithm described above does not take the position of the text lines of the document into account. Thus it can happen that a line gets striked out by the horizontal segment of a leader. We modified the algorithm to move the port up or down by a small offset to avoid such overlappings and place the leader into the gap between the lines.

It is quite hard to determine the positions of the lines in Tex because they are not fixed until the document is written to the output file. But in Luatex we can modify the line breaking algorithm such that it inserts special nodes into the data structures of Tex that write the position of every line into a text file when typesetting the page. In a second Tex run we can read the line positions from this file and use them for our algorithm.

4. Improvements

In this section we discuss some general improvements that can be used by every algorithm described in the previous section. They are already implemented in our package.

4.1. Label clustering

Most of the algorithms described in the previous section place labels in a single stack (that is, without gaps between them) beginning at the upper margin of the page. This can produce unnecessarily long leaders, for example when the text contains a single site near the end of the page. We split the labels into separate clusters and place each of them near the corresponding sites in the text. An algorithm for clustered labeling is also described by Nöllenburg et al. [NPS10]. Our approach is simpler but slower.

To group the labels into clusters we use Algorithm 4.1. It repeatedly joins adjacent clusters as long as they intersect each other. To test if two clusters intersect we place the contained labels as a stack each beneath the arithmetic mean of the sites in the cluster. The clusters intersect if their corresponding stacks overlap.

The positioning algorithm is executed independently for each of the identified clusters. The intended position is passed to the algorithm as a parameter.

Algorithm 4.1: Clustering labels

Input: $p_1, \ldots, p_n$ are the sites in the text ordered by their y-coordinate from top to bottom
Output: list of clusters S

```
1  S ← [{p_1}, {p_2}, ..., {p_n}]
2  i ← 1
3  while i ≤ #S − 1 do
4      if clustersIntersect(S[i], S[i + 1]) then
5          S[i] ← S[i] ∪ S[i + 1]
6          S.delete(i + 1)
           // as the size of stack i has increased we check again for
              intersection with the previous stack in next iteration
7          i ← max{1, i − 1}
8      else
9          i ← i + 1
10 return S
```

4.2. Two-sided label placement

On some page layouts there is enough space to place labels in the margins on the left *and* the right side of the text. There are algorithms for uniform labels that minimize the total leader length for certain leader types placing the labels on both sides, some of them mentioned in Section 1.

We don't use such algorithms here, but give simple heuristics that yield a partition of the labels into two sets. We have to decide for each label on which side of the text it should be placed. Our approach is to split the sites by a vertical line through the text. The sites which are left of this split line are labeled on the left side, those right of the split line are labeled in the right margin.

There are several ways to determine the position of this split line. For most cases using the weighted median is the best option. It splits the sites such that the sum of the label heights on the left side is approximately equal to that of the right side. With this algorithm it is not an issue if the widths of the two margins are different (which means that the height of a label depends on the side on which it is placed). Another option is splitting the sites at the middle of the text area. This is especially useful for documents typeset in two columns: the notes for the left column are placed in the left margin, those of the right column in the right margin.

4.3. Highlighting portions of text

In the previous sections we connected every note to a specified point inside the text. However it is often useful to mark to which portion of text the note refers to. Our package provides the additional command `\todoarea`, which inserts a note connected to a highlighted text passage. This command can be used with every label placement algorithm and every leader type.

The specified text is highlighted by a colored background, which is quite hard to implement in Latex. We use the `soul` and `soulpos` package for this task. They are easy to use, but the negative side is that they require an additional Latex run. If the highlighted text contains line breaks, we indicate this by special marks at the beginning and end of the line (see Figure 4.1). In the current implementation we always connect the leader to the begin of the selected text. If the leader would intersect the highlighted text, it is cropped.

Of course, this is not the only reasonable anchor for the leader. When the label is below the selection, it makes more sense to connect the label to the end of the highlighted text. When the label is directly beside the text, a leader to the nearest point of the highlighted area would be better. These options are illustrated in Figure 4.2. Of course, other points on the boundary are conceivable as well. If the selected text contains a page break, we get even more options as the note could also be moved to the next page.

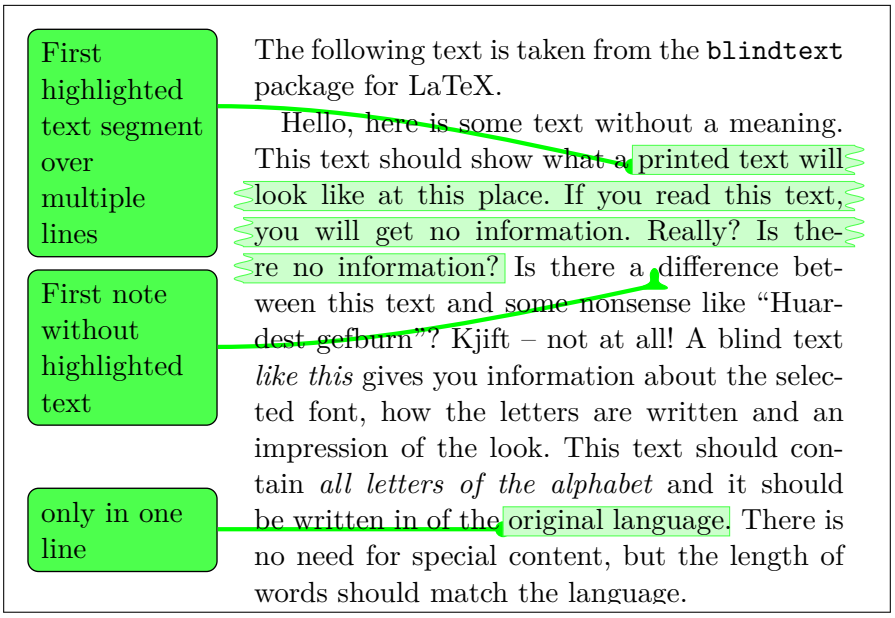


Figure 4.1.: Example for notes with highlighted text. The highlighted text for the first note spans several lines; this is indicated by special marks at the beginning and end of the lines.

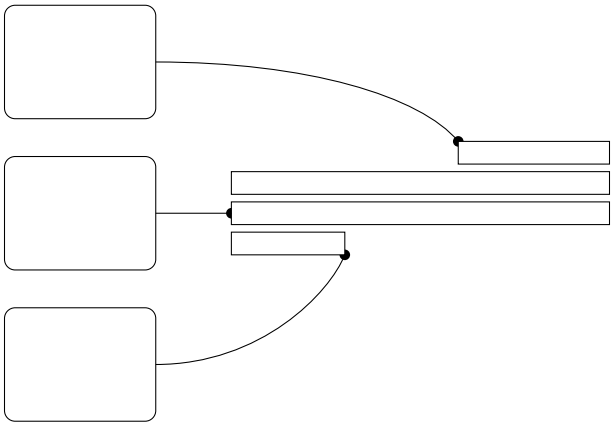


Figure 4.2.: Illustration of various anchor points on a highlighted portion of text. The rectangles on the right show one selection of text, which spans several lines. On the left there are different labels, which could be connected to the text. Depending on the label position different anchors at the text should be used to connect the leader.

5. Experimental Results

We first give a qualitative comparison of the presented algorithms based on an example document. We would have liked to introduce a numerical value that measures the drawing quality of the different algorithms, but it is not obvious how to find an appropriate indicator, which is suitable for all of the available leader types. So we stick to a subjective comparison of the results. After that, we compare the running times of the algorithms on some documents.

We compare the leader styles presented in the previous sections on an example document with nine comments in it. This document stays the same, only the options of our package are modified to switch between the available algorithms. The results of these algorithms are depicted in the appendix. We used the label clustering approach described in the previous section for all examples except for that of the *po*-leader algorithm. For comparison, we also processed the document with the `todonotes` package (see Figure A.1).

The NorthEastBelow algorithm for *s*-leaders (Figure A.2) is straight-forward and fast. It is easy for the reader to match the sites to their corresponding label. Using Bézier curves (Figure A.3) instead of the straight-line leaders yields a more aesthetic result with the disadvantage of a significantly higher runtime caused by the iterations of the force-directed algorithm. Using two-sided label placement with the same leader type (Figure A.4) produces shorter leaders because the labels can be placed closer to their site. Especially in text segments with a lot of comments this makes the relationship between sites and their labels clearer.

Our algorithm for *po*-leaders (Figure A.5) has a high asymptotic runtime and space consumption. But in practice when there are only few comments per page this is not an issue. Among the algorithms we implemented, this is the only algorithm minimizing the total leader length.

The *opo*-leaders (Figure A.6) and *os*-leaders (Figure A.7) are available mainly for comparison. Clearly, it gets hard for the reader to match sites to their labels on pages with many comments. In particular, if several sites are in the same line it is hard to tell the matching between sites and labels. On the other hand the leaders only run between the lines and in the track routing area and thus don't disturb the text.

The running times of Luatex with the different leader types for some example documents are shown in Table 5.1. Note that Documents 2 and 3 with 15 resp. 25 comments on one page are quite unrealistic. When using two-sided label placement both sides are processed independently and thus the algorithm for *po*-leaders becomes feasible again. The measured times are for a single run of Tex only. When the absolute position of a site of a label changes, a second run is needed. When we deactivate our package, processing still needs 1.4 seconds. This means that *s*- and *opo*-leaders cause only small extra cost compared to a standard Latex run. With the classical `todonotes` package processing needs about 1.8 seconds, too.

Leader type	D1		D2		D3	
number of margins	1	2	1	2	1	2
s	1.8	1.7	1.9	1.9	2.2	2.2
Bézier	5.7	5.4	33.2	11.1	322.9	116.3
po	4.8	3.0	17.7	6.2	—	27.6
po avoiding text lines	7.0	4.0	26.8	9.5	—	42.4
opo	1.8	1.7	1.9	1.9	2.2	2.2
classical `todonotes`	1.9		2.2		2.6	
without `luatodonotes`	1.4		1.4		1.3	

Table 5.1.: Running times of the different label styles on three one-page documents D1, D2, and D3 (in CPU seconds). The times were measured using a Intel Core 2 Duo E8400 with 3.0 GHz. D1 is the instance with 9 comments shown in the figures in the appendix. D2 has 15 comments, D3 has 25. For each document, we report two running times; for label placement into one margin vs. both margins. We use a raster height of 1 cm for *po*-leaders, resulting in 28 horizontal strips. We couldn't use *po*-leaders for D3 with one margin because the algorithm needed too much memory. For comparison we also give the running times for the classical `todonotes` package (which does not support placing labels in both margins) and the running times for the document without loading the luatodonotes package.

6. Conclusion and Open Problems

All our algorithms turned out to work well in practice—some of them cannot process too many labels on a single page. Using both margins helps in terms of speed. By visual inspection we reached the conclusion that *s*-leaders or Bézier leaders work better than the *os*-leaders used by other type-setting programs. The reason may be that the reader's eye can follow leaders without bends more easily. It would be interesting to verify this in a user study. With the modular design of our Latex package it is easy to improve the label-placement algorithms or add additional ones.

Comparison to `todonotes`. Our package has some benefits compared to the classical `todonotes` package: With the various leader types and algorithms the user has many configuration options that control the layout of the labels on the page. The drawback of some of these algorithms is that they strongly increase the running time needed to process the document. With our method to locate the sites on the page we are able to place the notes at almost arbitrary positions in the document, for example, inside floating environments or footnotes. But on the other hand, this method requires a second Tex run when the absolute position of one of the sites changes. When the page margins are wide enough, we can place our notes on both sides. So pages with many labels on them can be arranged more neatly opposed to the one-sided layout of `todonotes`.

Working with Luatex. Luatex is a major step forward compared to classical Tex interpreters. Even if it is still in its beta phase, it is quite usable today. Besides from the possibility to embed high-level programming code into Tex documents—which is much more comfortable compared to traditional Latex package writing—there are powerful methods to access internals like counters, dimensions and boxes.

But sometimes the collaboration between Lua and Tex gets really tricky, especially when exchanging data between Lua variables and Tex registers. For some tasks you have to pay attention to Tex basics like the category codes of characters and subtleties of the macro expansion mechanism.

Open problems. An interesting theoretical problem remains open: Given an instance with non-uniform label heights, is it always possible to place the labels without gaps so that *s*-leaders do not cross each other even if we insist that the ports are centered vertically at each label?

Admittedly, our dynamic program for *po*-leaders is quite slow. Can we save time by computing labelings that are just feasible rather than length-minimal? For the other leader types, on the contrary, it would be interesting to minimize the total leader length.

Such algorithms are known only for the case of uniform labels. Perhaps we can transfer these results using a raster for the labels as done in the *po*-leader algorithm. Especially our *s*-leader algorithms sometimes generate unnecessarily long leaders in the current implementation, which could be shortened by this approach.

When minimizing the leader length in the two-sided case, one could also try to improve the approach for partitioning the labels. Our current algorithms split the sites into two sets by a vertical line. Depending on their positions there could be partitions that yield better results as illustrated by Benkert et al. [BHKN09]. One could investigate if some of the known algorithms for two-sided boundary labeling can be adapted to work with non-uniform labels.

Improvements of the package. We have some ideas for further improvements of our package. The force-directed Bézier curve algorithm is quite slow at the moment. We think that we could speed up the computation of the distances between curves by doing a rough estimate first and computing the fine approximation only when needed. It would be interesting to transform the *po*-leaders into Bézier curves. As our algorithm yields a length-minimal *po*-labeling this could produce a shorter leader length than our approach with *s*-leaders. But it is not clear how to inhibit intersections between the curves.

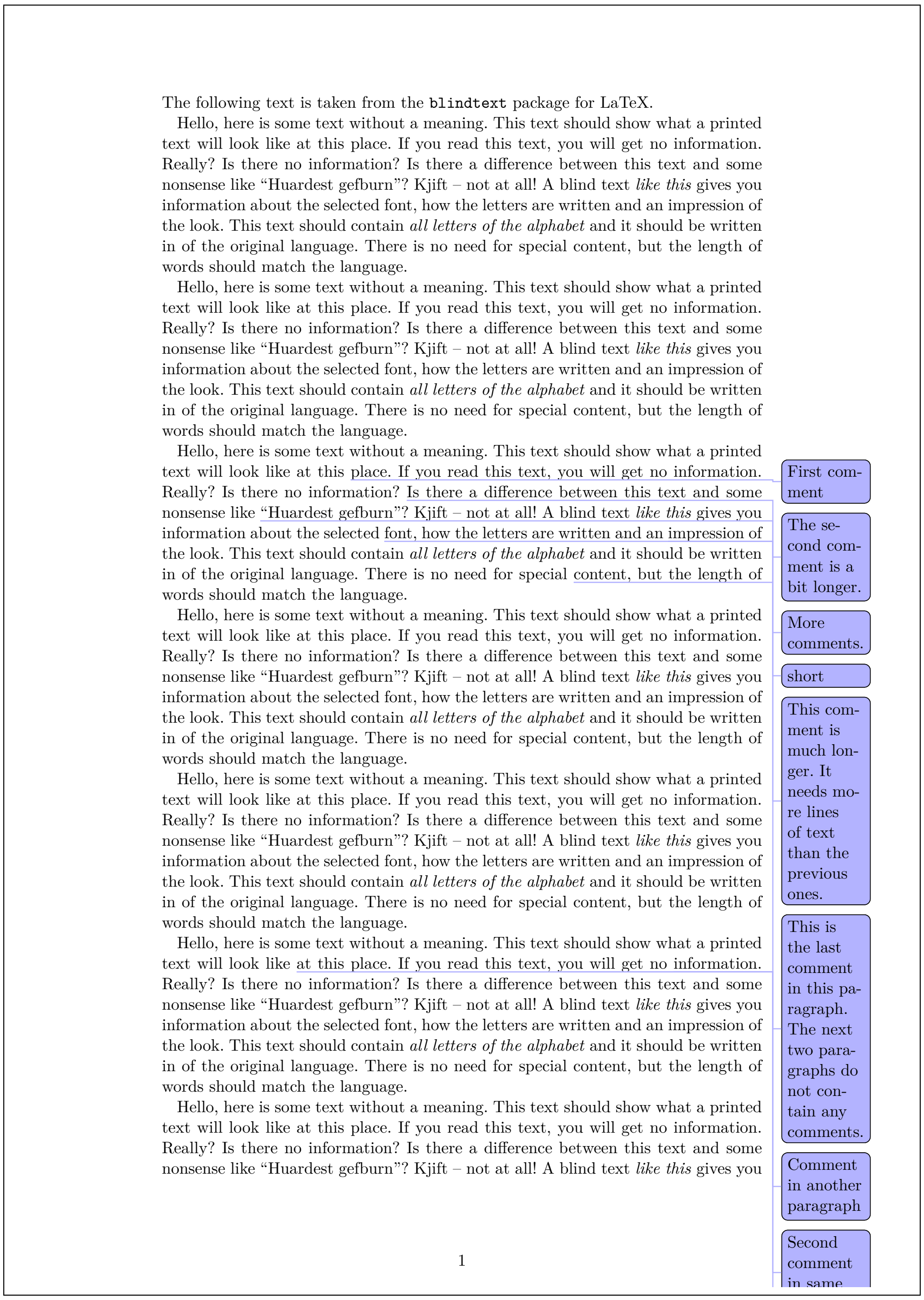
In the current version the raster height needed for the *po*-leader algorithm is given as a package option by the user. A heuristic that computes the raster height depending on the height of the present labels would be useful.

Highlighting text portions has still room for improvements. The current implementation using the `soulpos` package is quite slow as it requires several runs of Tex. One could speed this up by finding another way to highlight the text. The other problem is to choose the best position for the anchor, that is, the point the leader is connected to. One solution could be to compute several candidates for the anchor point and choose the best one when placing the labels or drawing the leaders.

Appendix A.

Example Documents

On the following pages we show one example document. It is processed with various options for the `luatodonotes` package.



The following text is taken from the `blindtext` package for LaTeX.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text *like this* gives you

1

Figure A.1.: Example document produced by the `todonotes` package. Note that the labels are clipped at the bottom because they don't fit on the page. The last comment is not shown at all.

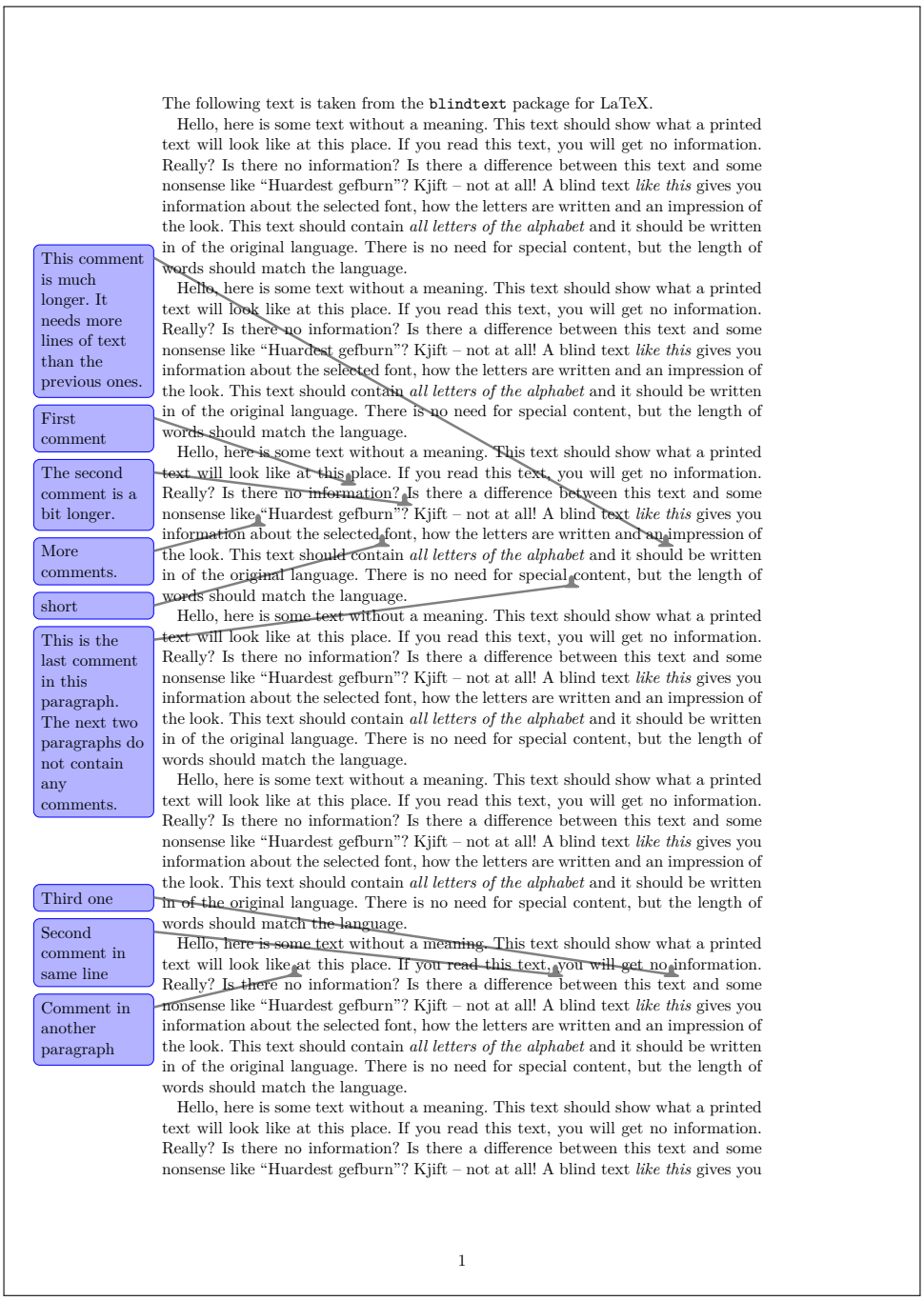
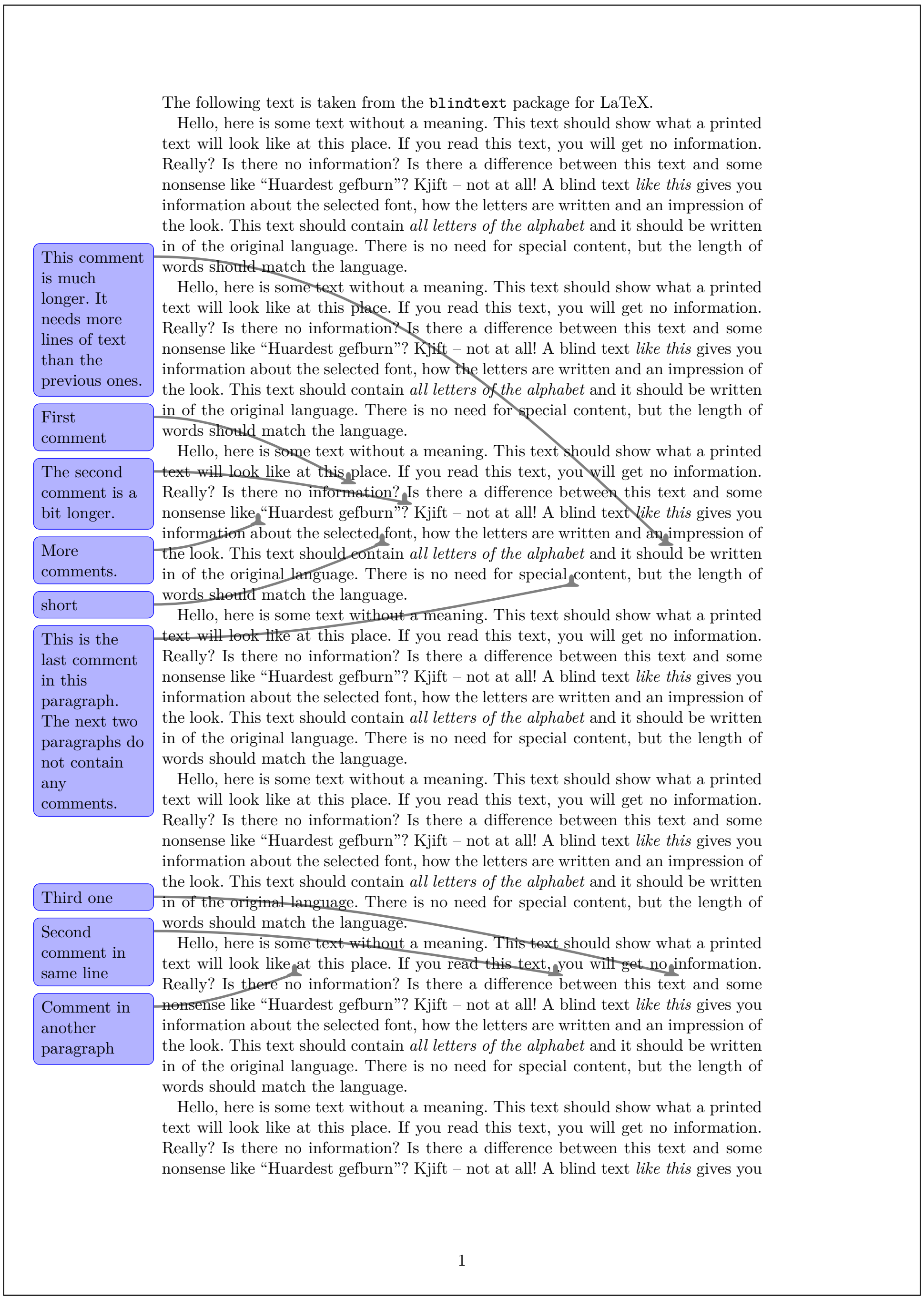


Figure A.2.: Example document produced using the NorthEastBelow positioning algorithm with *s*-leaders. The package was loaded with the parameters `positioning=sLeaderNorthEastBelowStacks` and `leadertype=s`.


The following text is taken from the `blindtext` package for LaTeX.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you

1

Figure A.3.: Example document produced using the NorthEastBelow positioning algorithm with Bézier leaders. The package was loaded with the parameters `positioning=sLeaderNorthEastBelowStacks` and `leadertype=sBezier`.

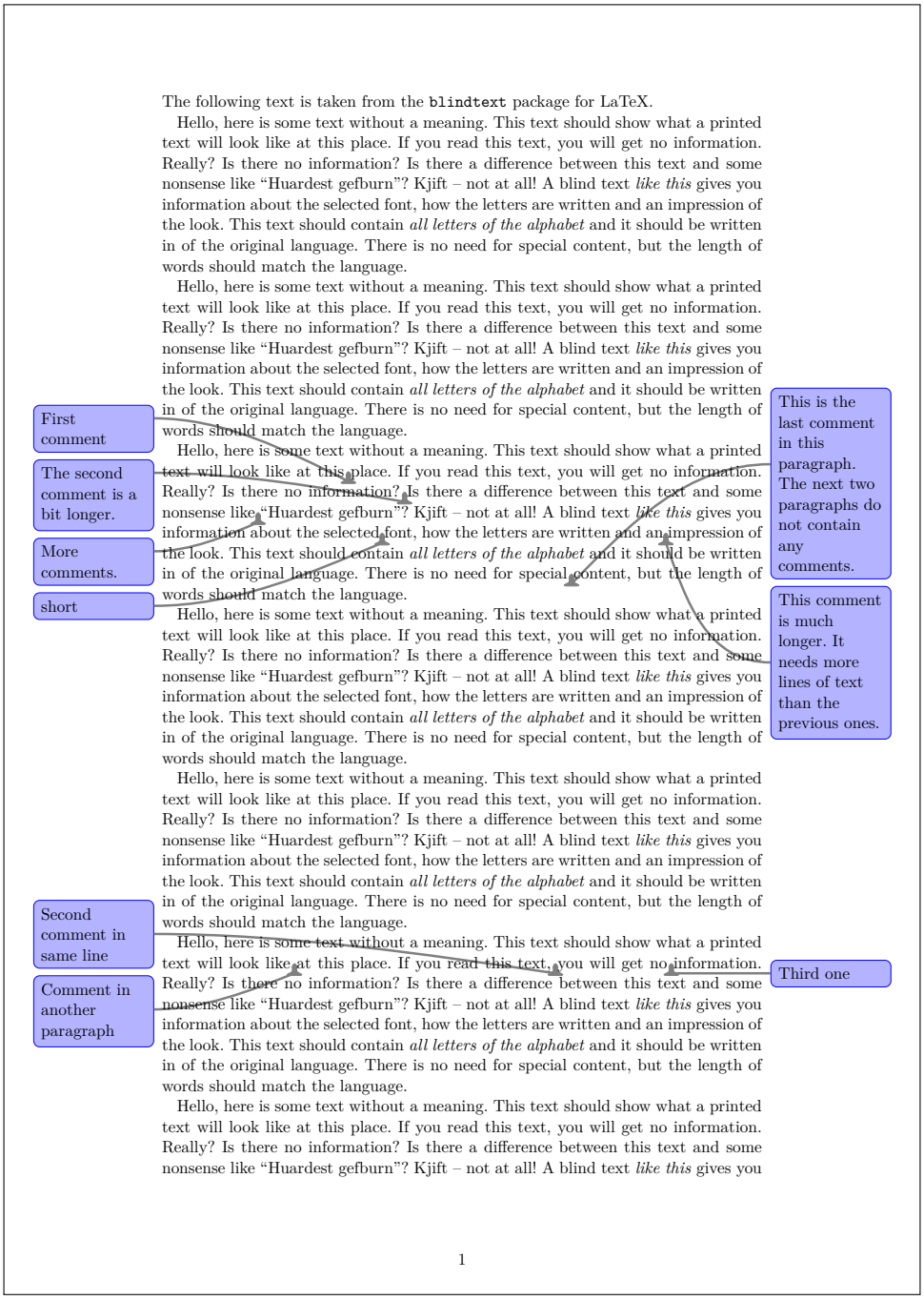
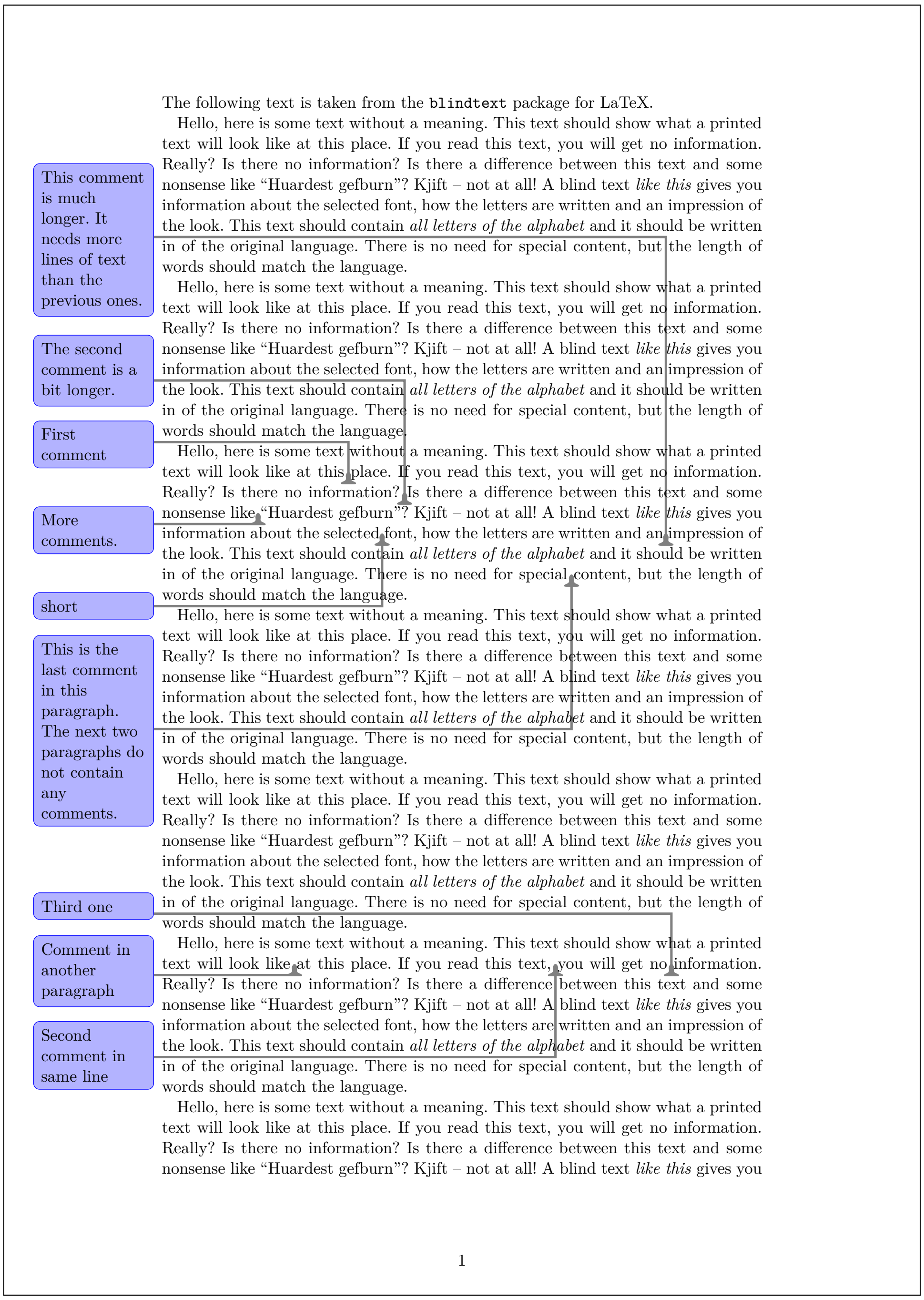


Figure A.4.: Label placement produced by the NorthEastBelow algorithm using Bézier leaders. The comments were partitioned left and right using the weighted median. The package was loaded with the parameters `positioning=sLeaderNorthEastBelowStacks`, `leadertype=sBezier` and `splitting=weightedMedian`.



The following text is taken from the `blindtext` package for LaTeX.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you

This comment is much longer. It needs more lines of text than the previous ones.

The second comment is a bit longer.

First comment

More comments.

short

This is the last comment in this paragraph. The next two paragraphs do not contain any comments.

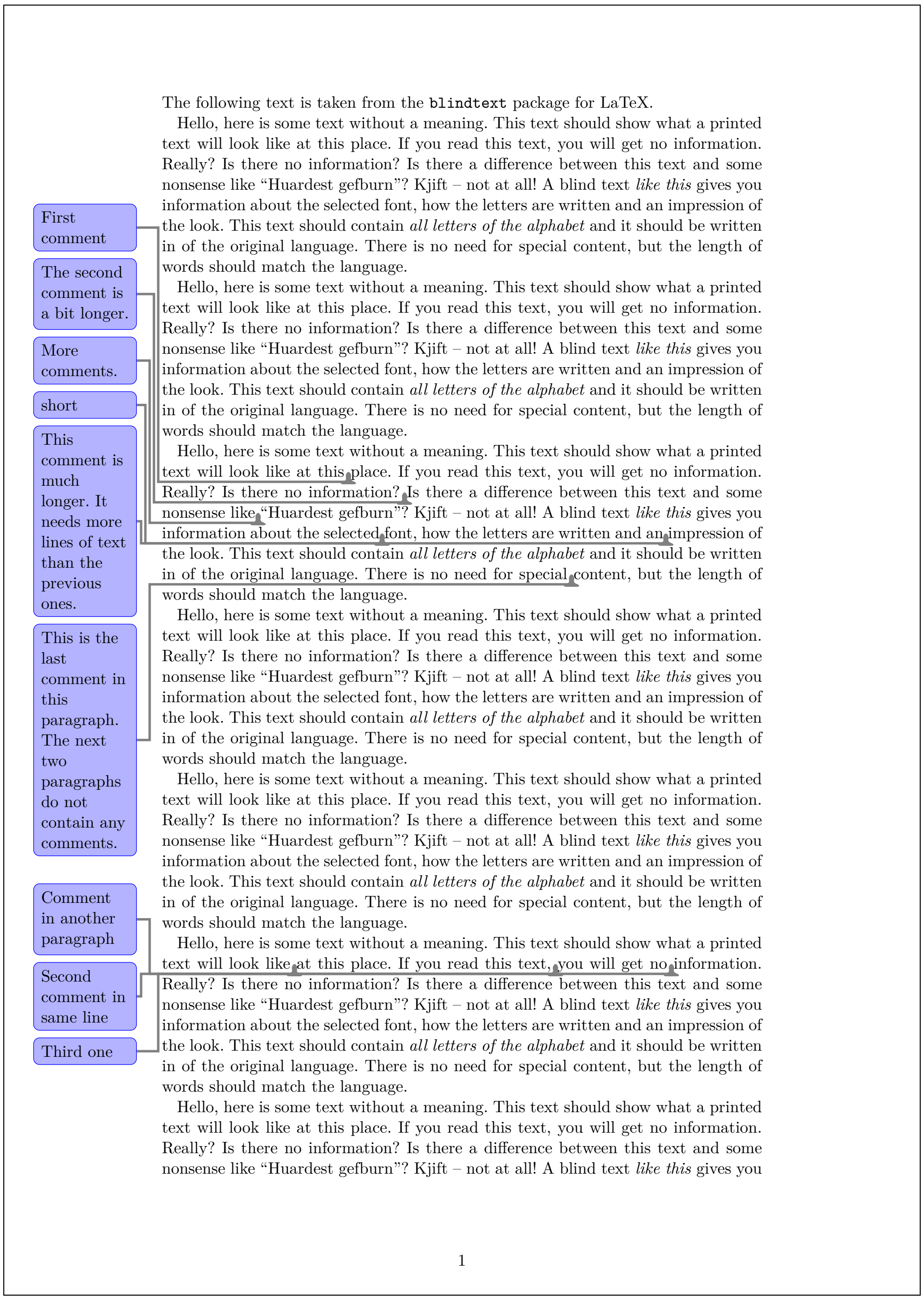
Third one

Comment in another paragraph

Second comment in same line

1

Figure A.5.: Example document produced using *po*-leaders. The package was loaded with the parameters `positioning=poLeadersAvoidLines` and `leadertype=po`.



The following text is taken from the `blindtext` package for LaTeX.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

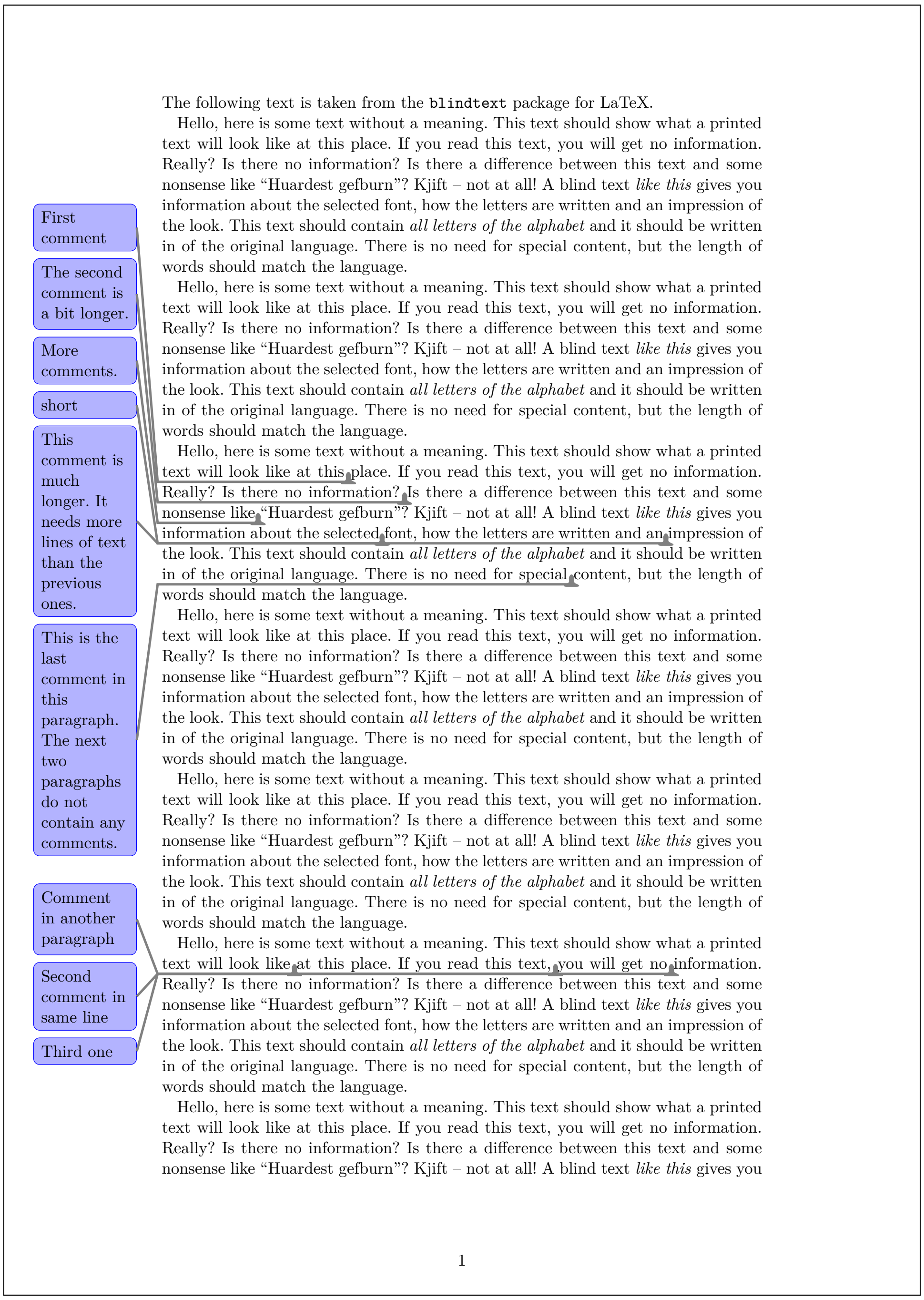
Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you

1

Figure A.6.: Example document produced using *opo*-leaders. The package was loaded with the parameters `positioning=inputOrderStacks` and `leadertype=opo`.


First comment

The second comment is a bit longer.

More comments.

short

This comment is much longer. It needs more lines of text than the previous ones.

This is the last comment in this paragraph. The next two paragraphs do not contain any comments.

Comment in another paragraph

Second comment in same line

Third one

The following text is taken from the `blindtext` package for LaTeX.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you information about the selected font, how the letters are written and an impression of the look. This text should contain *all letters of the alphabet* and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text *like this* gives you

1

Figure A.7.: Example document produced using *os*-leaders. The package was loaded with the parameters `positioning=inputOrderStacks` and `leadertype=os`.

Bibliography

[AES99] Pankaj K. Agarwal, Alon Efrat, and Micha Sharir: Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM J. Comput.*, 29(3):912–953, 1999.

[Bar13] Gioele Barabucci: fixmetodonotes. http://www.ctan.org/pkg/fixmetodonotes, 2013.

[BHKN09] Marc Benkert, Herman J. Haverkort, Moritz Kroll, and Martin Nöllenburg: Algorithms for multi-criteria boundary labeling. *J. Graph Algorithms Appl.*, 13(3):289–317, 2009.

[BKNS10] Michael A. Bekos, Michael Kaufmann, Martin Nöllenburg, and Antonios Symvonis: Boundary labeling with octilinear leaders. *Algorithmica*, 57(3):436–461, 2010.

[BKPS06] Michael A. Bekos, Michael Kaufmann, Katerina Potika, and Antonios Symvonis: Multi-stack boundary labeling problems. In S. Arun-Kumar and Naveen Garg (editors): *Proc. Int. Conf. Foundat. Software Tech. Theor. Comput. Sci. (FSTTCS'06)*, volume 4337 of *LNCS*, pages 81–92. Springer-Verlag, 2006.

[BKSW07] Michael A. Bekos, Michael Kaufmann, Antonios Symvonis, and Alexander Wolff: Boundary labeling: Models and efficient algorithms for rectangular maps. *Comput. Geom. Theory Appl.*, 36(3):215–236, 2007.

[FHS+12] Martin Fink, Jan Henrik Haunert, André Schulz, Joachim Spoerhase, and Alexander Wolff: Algorithms for labeling focus regions. *IEEE Trans. Visual. Comput. Graphics*, 18(12):2583–2592, 2012.

[HHH] Hans Hagen, Hartmut Henkel, and Taco Hoekwater: Luatex. http://www.luatex.org/.

[HPL14] Zhi Dong Huang, Sheung Hung Poon, and Chun Cheng Lin: Boundary labeling with flexible label positions. In *Algorithms and Computation*, pages 44–55. Springer, 2014.

[Kle12] Josef Kleber: pdfcomment. http://www.ctan.org/pkg/pdfcomment, 2012.

[KNR+13] Philipp Kindermann, Benjamin Niedermann, Ignaz Rutter, Marcus Schaefer, André Schulz, and Alexander Wolff: Two-sided boundary labeling with adjacent sides. In F. Dehne, R. Solis-Oba, and J. R. Sack (editors): *Proc.*

13th Int. Algorithms Data Struct. Symp. (WADS'13), number 8037 in *LNCS*, pages 463–474. Springer-Verlag, 2013.

[Mid12] Henrik Skov Midtiby: todonotes. http://ctan.org/pkg/todonotes, 2012.

[NPS10] Martin Nöllenburg, Valentin Polishchuk, and Mikko Sysikaski: Dynamic one-sided boundary labeling. In *Proc. 18th SIGSPATIAL Int. Conf. Adv. Geogr. Inform. Syst. (ACM-GIS'10)*, pages 310–319. ACM, 2010.

[RV14] Juan Rada-Vilela: easy-todo. http://ctan.org/pkg/easy-todo, 2014.

[Tan] Till Tantau: PGF and Ti*k*Z – Graphic systems for TeX. http://sourceforge.net/projects/pgf/.

[Ver13] Didier Verna: Fixme. http://ctan.org/pkg/fixme, 2013.

[Wol13] Alexander Wolff: Graph drawing and cartography. In Roberto Tamassia (editor): *Handbook of Graph Drawing and Visualization*, chapter 23. CRC Press, 2013.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 19. August 2014

. . . . . .
Fabian Lipp