View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne

Reverse Digital Typography

Tao C. Lee
LCAV, EPFL

July 14, 2014

Semester projects

Supervised by

Dr. Paolo Prandoni
Professor Martin Vetterli

Lausanne, academic year 2012 – 2013

Reverse PostScript

Tao C. Lee
Informatique, EPFL

June 20, 2012

Semester Project 1st year master

Supervised by

Dr. Paolo Prandoni
Professor Martin Vetterli

Lausanne, academic year 2011 – 2012

Abstract

Reverse engineering of typefaces has been an active research area of digital typography. However, past research works are limited to typographic parameters extraction and curve fitting, and no reverse-engineering equivalent of any classical digital typesetting software (e.g. METAFONT compiler, LaTeX package) exist to date. Interestingly, the rise of cloud computing has made large scale reverse typesetting of ancient books and artworks a real challenge, as the example of Google Books, which calls for reverse typesetting software. In this research report, we present the design and implementation of Reverse METAFONT compiler for reverse typesetting of raster-scanned fonts. We first give an introduction of reverse digital typography. Second, we report algorithms and techniques for generating METAFONT representations of raster-scanned fonts. Third, we report compression algorithms and techniques to reduce the number of control points in METAFONT representations. Two performance metrics are introduced to evaluate the compression algorithms: normalized control points reduction and global mean-squared-error. Finally, we evaluate the algorithms and techniques using Times New Roman font set. We present 52 reverse-engineered Times New Roman alphabets to give a demonstration.

Keyword: Typesetting; reverse digital typography; METAFONT; curve fitting; splines; parametric typographic representations

1 Introduction to reverse digital typography

1.1 Past, present and the future

Digital typography has been an all-time classic. Ever since the development of TeX system and METAFONT compiler by Donald E. Knuth, the past decades have been the golden ages of digital typography. The wide-spread influence of personal computers has completely changed the landscape of typesetting and publishing. Besides typesetting fonts, TeX system and METAFONT compiler are especially famous for typesetting complex mathematical formulae. Today, TeX systems and METAFONT compiler are used by amateur and professional users worldwide in the form of LaTeX package, and the basic ideas of METAFONT have been incorporated into successful commercial products such as PostScript of Adobe Inc. and TrueType of Microsoft.

Prior research works on digital typography can be largely classified into two categories. First, generation of new typefaces with new parameters [[Haralambous93], [Shamir96], [Hertz98], [Feng75], [Feng75−2]]: [Haralambous93] provides a research overview of the conversion between METAFONT representations and PostScript Type 1 representaions, and several ways to extract new parameters during the process. The existence of tools to convert between METAFONT representations and PostScript Type 1 representations establishes the equivalence between these two representations, meaning that generating one would imply the other. For this reason, we focus on the generation of METAFONT representations

in this research project. [Shamir96] reports methods of automatic typographic parameter extraction. [Hertz98] provides an effective way to extract typographic parameters by random sampling and windowing. [Feng75] provides an several useful algorithms on syntactic pattern generation. [Feng75−2] provides several useful algorithms on polygon outline generation, the algorithm of ordering of boundary points has been adopted by Reverse METAFONT compiler and is explained in later sections. Second, recovering typographic parameters of old typefaces [[Schneider90], [Matas95], [Shao96]]: [Schneider90] provides an effective method for automatic fitting of cubic splines. [Matas95] reports a median-filtered estimation method for tangent and curvature of digital lines. [Shao96] reports a robust method to identify critical points in typefaces. Algorithms and techniques developed by these research works have built solid foundations for reverse digital typography as we we show in later sections.

Cloud computing has emerged to become a dominant platform for digital media exploration. Google Books, for instance, has put large number of ancient books online. However, these ancient books are scanned images or OCR-ed texts and are not properly typeset. This situation has unfortunately degraded the quality of these electronic books. On the other hands, cloud computing provides not only potentially unforeseen computing power for large-scale signal processing of online media, but also a global-scale environment to deliver good perceptual experiences to worldwide users. In this regard, enhancing the quality of these electronic books has become a great opportunity for signal processing research. The idea is to carry out the classical typesetting process in reverse order: starting from rastered-scanned images, producing the parametric representations of book pages, and finally generating the re-typeset electronic books.

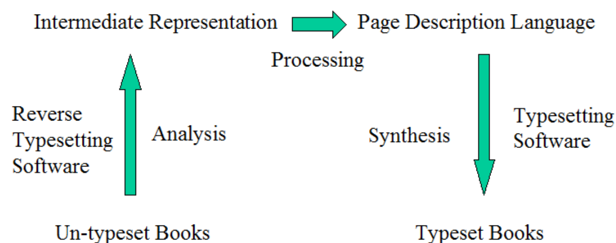


Figure 1: The Cycle of Electronic Publishing

We define the above process as *reverse digital typography*. Similar to its long-standing digital typography sibling, reverse digital typography also requires a set of software in similar roles as the classical METAFONT compiler and LaTeX package. Lacking this set of software has become an obstacle for large-scale re-typesetting of ancient books. Moreover, the reverse typesetting software can complement the classical typesetting software to form a complete cycle of electronic publishing as shown in Figure 1, and would enable further

signal processing research on different aspects of electronic publishing. To accomplish this goal, the scope of the software would cover the whole process of reverse compilation of images of book pages into page description languages. We choose METAFONT as our target language rather than PostScript or TrueType, for reasons that it is not a commercial standard and the conversion between METAFONT and PostScript is well known. Considering the architecture of classical typesetting software, we divide the software into several parts. The first and the most important part is the one that does reverse METAFONT modeling of font images. Therefore, this research project is devoted to the development of Reverse METAFONT compiler that can take raster-scanned fonts as input and generate their METAFONT representations as output.

The benefits of re-typesetting are not limited to better-quality electronic books, the METAFONT representations of fonts and pages are parametric and are thus scalable to all kinds of resolutions. This kind of scalability stems from the classical idea behind the invention of METAFONT. At that moment, the idea was developed to be able to adapt to printers of different resolutions. Today, this scalability is particularly useful for electronic publishing as zooming is often used during user exploration, and it essentially hides the walls of pixels to provide pixel-free perceptual experiences to end users.

1.2 Reverse METAFONT compiler

The architecture of Reverse METAFONT compiler is shown in Figure 2. Three building blocks represent the three important stages of the compilation process: image parsing, structure analysis and code generation.

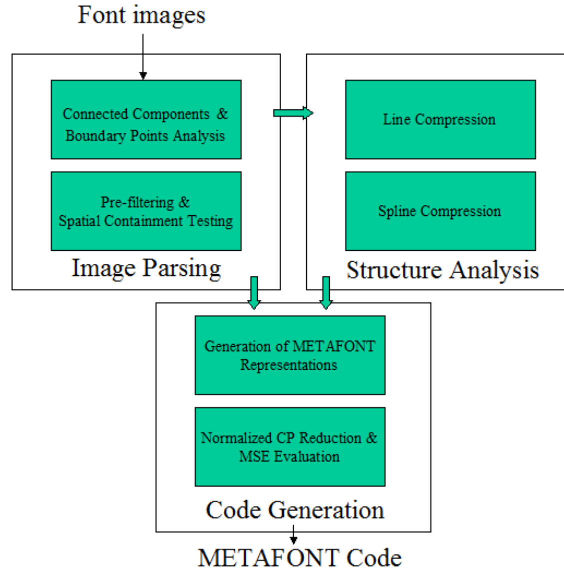


Figure 2: Reverse METAFONT Compiler Architecture

Image parsing is the image processing front-end of reverse METAFONT compiler. In this stage, font images are processed by image processing techniques to generate intermediate representations for later stages. While revious approaches to font pattern analysis mainly focused on gray-level segmentation or edge detection, we directly find all the connected components in the input images [Hesselink01]. This approach gives us the benefits of 1) using the most up-to-date algorithm and 2) avoiding unconnected edges. Using [Hesselink01] has the benefits of fast computation speed with higher-level of parallelism, while avoiding unconnected edges makes ordering of boundary points more straightforward.

After finding all the connected components, we proceed to order all the boundary points of each connected component. We develop an algorithm similar to [Feng75−2], but directly apply to connected components. We use 8-connectivity to label connected components, but only use 4-connectivity to label the boundary points of each connected component. This simplify the ordering algorithm and reduce the residual boundary points, but exclude the possibility of one-pixel wide typographic structures. We consider this as a reasonable assumption since admitting one-pixel wide typographic structures is vulnerable to quantization noise. After all the boundary points are ordered, we filter out small ordered groups of boundary points, and perform spatial containment testing to determine the containment relation between connected components. This is an important step as we use METAFONT *fill* and *unfill* commands to generate the fonts. The outer component is generated by *fill*, whereas the inner component is generated by *unfill*. Using the simulated-pen terminology of METAFONT, the reverse-engineered fonts can also be understood as generated by implicit pens.

After the determination of spatial containment relations, we can already use the obtained intermediate representations of font images to generate the METAFONT representations using METAFONT *fill* and *unfill* commands. However, experiments have shown this kind of primitive METAFONT representations is not efficient [Lee12] (e.g. for a 400x300-pixels image, over 2000 control points are used) and does not exclude quantization noise from the parametric representations. Therefore, toward more efficient font modeling, we perform structure analysis on the intermediate representations. In particular, we use robust finite differencing as a means to identify different typographic components (e.g. straight lines, splines, serifs, etc.), and perform a divide-and-conquer strategy to reduce the required number of control points.

Since the classical Computer Modern Typefaces and many other font sets are generated by different scales of straight line and splines, we pay particular attention to the modeling of these two components. In fact, more complicated typographic components such as serifs, can be modelled by finite combinations of these two basic constructs [Shamir96]. We introduce some parameters to trade-off the reduction of control points and accuracy. These parameters are fed to the filters to counteract quantization noise. Details of these filters are explained in later sections. For straight lines, we have developed a two-level threshold filtering scheme to control the estimation process of straight lines. For splines, we use the boundary points of straight lines as the end points of splines, and use a median filtering scheme to estimate the initial tangent vectors. Recursive spline fitting has been a long-standing research topic in numerical analysis, and we adopt the algorithm proposed by [Schneider90] to fit the digitized curves with splines (Bézier cubics).

After the modeling of straight lines and splines, reduction of control points can be achieved by replacing all digitized points along a straight line with just two end points, and replacing all digitized points along a spline with two end points and two spline control points [Lee12]. Through straight lines and splines modeling, more than 90% of control points reduction can be achieved [Lee12]. Nevertheless, the rate-distortion trade-off exists, and with higher level of control points reduction, the larger the deviation between the reversed-engineered fonts and the input font images. We use normalized Control Points (CP) reduction to denote the rate, and use Maximum Error (ME) and Mean-Squared-Error (MSE) to denote the distortion. The results of normalized CP reduction and MSE are shown in later sections.

We evaluate Reverse METAFONT compiler on a small benchmark of raster-scanned Times New Roman alphabets comprising of 52 font images of English upper-case and lower-case alphabets. The evaluation process is, first, to compile the font images with Reverse METAFONT compiler to generate METAFONT representations. Second, compile the generated METAFONT code with METAFONT compiler and convert the output to DVI format in order to render them with a DVI viewer. The process goes through the cycle of electronic publishing as shown in Figure 1. The results show efficient control

points usage with moderate MSE over most benchmarked fonts, still for some relatively complicated alphabets, larger MSEs are observed. Nevertheless, with lower-level of CP reduction, MSEs can be suppressed with the expense of using more control points. We provide detailed discussion in later sections.

2 Generating METAFONT representations: algorithms and techniques

2.1 Overview

As described in section 1, research works done in the past have laid the foundations of building a real Reverse METAFONT compiler. Nevertheless, it takes many design decisions in order to fuse previous research works and present goals together to produce a working prototype. In the following subsections, we present the algorithms and techniques for generating primitive METAFONT representations (without reduction of control points). After the presentation of the algorithms, a short demonstration of the reversed-engineered fonts using six Times New Roman alphabets is presented. Finally, we give a discussion on the quality of the reverse-engineered fonts.

2.2 Algorithms & techniques

The compilation process is illustrated in Algorithm 1. We adopt the algorithm proposed by [Hesselink01] for connected component analysis. The algorithm for ordering of boundary points is illustrated in Algorithm 2. Pre-filtering and spatial containment testing are relatively simple so we omit the algorithmic descriptions here. The generation of primitive METAFONT representations is algorithmically described in Algorithm 3.

Input: Raster-scanned font images
Output: METAFONT representations of font images

Connected component analysis [Hesselink 2001];
Ordering of boundary points;
Pre-filtering of small ordered groups generated by the ordering algorithm;
Spatial containment testing;
Structure analysis;
Generate METAFONT representations;

Algorithm 1: Reverse METAFONT compilation

Input: Connected Components labelled (1, 2, ...) on an Image(width, height)
Output: A list for each connected component:
List1: ((x11, y11), (x12, y12), (x13, y13), ..., (x11, y11)),
List2: ((x21, y21), (x22, y22), (x23, y23), ..., (x21, y21)), ...

for all labelled point Image(x, y) **do** {1st pass}
  if NB(8) $\neq$ Image(x, y) **then**
    mark (x, y) as boundary points;
  end if
end for
for all labelled point Image(x, y) **do** {2nd pass}
  if (x, y) is a boundary point **then**
    insert all its NB(4) neighboring boundary points in a List(x, y) = () in a counter-clockwise fashion
  end if
end for
{3rd pass}
S algorithm developed by [Feng 1975]

Algorithm 2: Ordering of boundary points by connected components

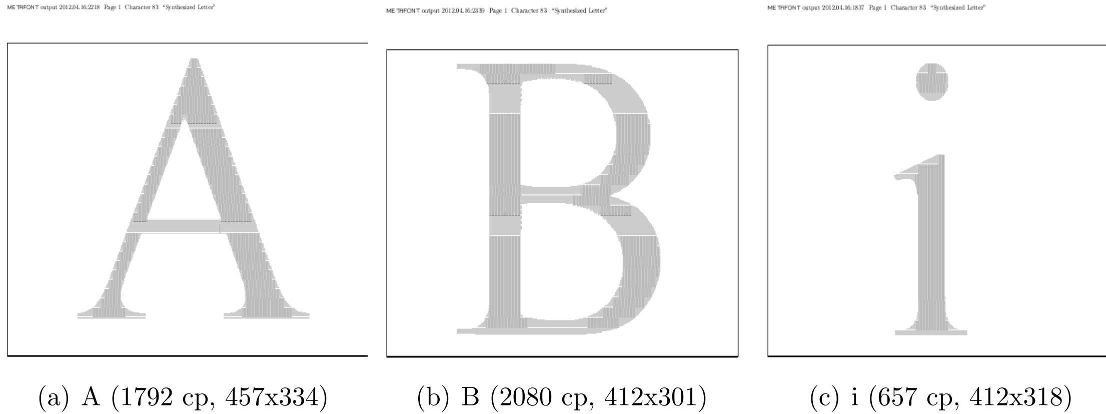
Input: Connected components with ordered boundary points, and marked with spatial containment relations
Output: Primitive METAFONT representations of font images

for all connected components **do**
  if marked as outer **then**
    write *fill* and all control points in a cyclic path;
  else
    write *unfill* and all control points in a cyclic path;
  end if
end for

Algorithm 3: Generation of primitive METAFONT representations

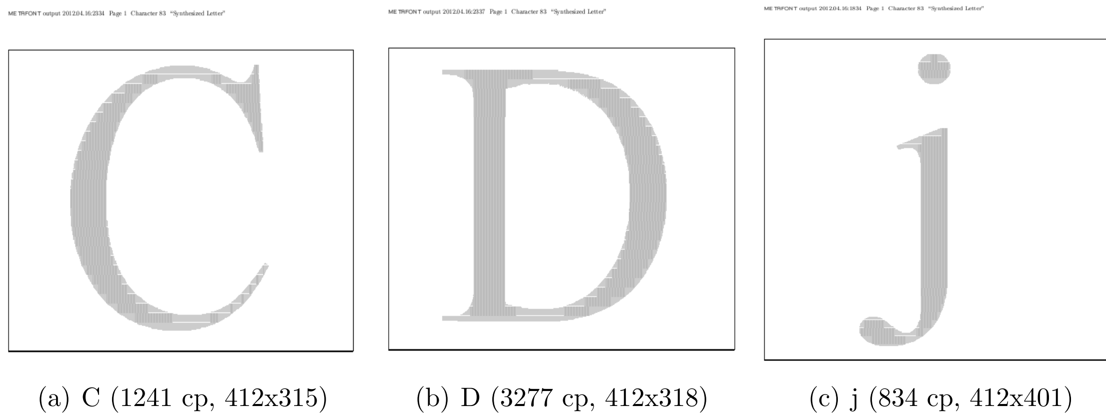
2.3 Experimental results

With the generated METAFONT representations, we can compile them with METAFONT compiler, and convert the output to DVI format in order to render them with a DVI viewer. A short demonstration of the reverse-engineered fonts are shown in Figure 3 and 4, with total number of control points for the given input image size. As we can see, the total number of control points are high. For example, alphabet B consumes over two thousands control points for an input size of 412x301.



(a) A (1792 cp, 457x334) (b) B (2080 cp, 412x301) (c) i (657 cp, 412x318)

Figure 3: Synthesized Times New Roman Alphabets



(a) C (1241 cp, 412x315) (b) D (3277 cp, 412x318) (c) j (834 cp, 412x401)

Figure 4: Synthesized Times New Roman Alphabets

2.4 Discussion

Up to this point, we have demonstrated that the METAFONT representations generated by Reverse METAFONT compiler can be synthesized to displayable fonts. While this might not sound like a great achievement, it is an important step that paves the way for further optimization of Reverse METAFONT compiler. The issue is that the required number of control points are high and quantization noise is not excluded from the METAFONT representations. In light of this, we focus on algorithms and techniques to reduce the required number of points in next section.

Nevertheless, to the best of our knowledge, Algorithm 2 and Algorithm 3 are not seen in any previous research. We find it interesting that these algorithms, although simple, can

really do reverse font modeling.

3 Reduction of control points: algorithms and techniques

3.1 Overview

As shown in section 2, Reverse METAFONT compiler can generate METAFONT representations successfully. Nevertheless, the issue is that the required number of control points are high and quantization noise is not excluded from the METAFONT representations. In the following subsections, we present the algorithms and techniques to perform effective reduction of control points. After the presentation of algorithms, a short demonstration of the reversed-engineered fonts using alphabet B is presented. Finally, we give a discussion on the quality of the reverse-engineered font. Two metrics, normalized control points reduction and mean-squared-error, are presented. The limitations of these two metrics are given in the end.

3.2 Algorithms & techniques

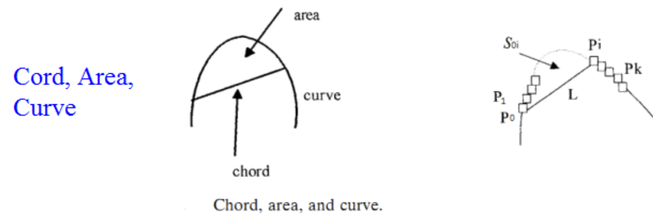
One of the foundations of structure analysis of digitized shapes is finite differencing [[Shao96], [Matas95]]. It is indispensable for the estimation of tangent and curvature along digitized curves. Nevertheless, a naive approach to finite differencing would be sensitive to quantization noise and would require a large number of samples to have stable results. Experiments have shown second-order finite differencing is more robust than first-order finite differencing. We adopt a second-order finite differencing scheme, cord-curve area response shown in Figure 5, as a front-end of structure analysis.

We use Algorithm 4 to compute cord-curve area response for every connected component. A motivating example of alphabet B is shown in Figure 6. With suitable choice of cord length, we can identify different typographic components. Although it is possible to estimate the suitable cord length by empirical formulae [[Shao96], [Lee12]], some hand tuning is required in practice. As we can see in Figure 6, line components have very small cord-curve area response. This is understandable as the cord-curve area of straight lines should be zero in theory. Nevertheless, we can still find some zero cord-curve area segments in splines. These segments can be understood as locally quantized straight lines and can be filtered out by threshold filters. We identify the components between straight lines as splines. The cord-curve area response of splines is noisy, but this so far does not affect the spline fitting process as we only require robust estimation of the start and end points of spline components. The only exception is the inflection point of B, since it marks the boundary of two splines. As we can see, the inflection point has a pulsed peak in its cord-curve area response, which requires one additional threshold to identify. But for all fonts in our selected benchmark, inflection points rarely appear. The cord length, threshold

parameters and filter taps are configurable parameters of Reverse METAFONT compiler and they are fed as input to Algorithm 4, 5 and 6. As we can see in section 4 and section 6, these parameters play important roles in the quality of reverse-engineered fonts.

Quantization noise is an annoying problem for robust fitting of digitized curves. Past research works have shown that this kind of noise cannot be effectively modelled as Gaussian noise. A more technical term for this kind of quantization noise is correlated quantization noise [Matas95], and experimental results have shown threshold and median filtering are particularly useful as schematically illustrated in Figure 7. We integrate these filtering schemes into structure analysis as illustrated in Algorithm 5 & 6. A two-level threshold filtering scheme is developed for robust estimation of line components, and we adopt the median filtering scheme propose by [Matas95] for robust estimation of initial tangent vectors of splines. We refine the spline fitting algorithm proposed by [Schneider90] with median-filtered estimation of tangent vectors in Algorithm 6.

The structure analysis part of Reverse METAFONT compiler is illustrated in Algorithm 5 & 6. Unlike previous approaches [[Shao96], [Schneider90]], we treat straight lines and splines separately. Although from a mathematical point of view, straight lines can be viewed as a degenerate case of splines. Considering the perturbation of quantization noise, experimental results have shown that it would more robust to identify straight lines first. Once a robust estimation of straight line components is achieved, we have implicitly identified the rough positions of spline components, recalling that the fonts are generated by only straight lines and splines as explained in section 1. The generation of efficient METAFONT represetations is a refined version of Algorithm 3 as illustrated in Algorithm 7. Depending on the level of compression (0: no compression; 1: compression of lines; 2: compression of spline; 3: compression of lines and splines), we perform different levels of control points reduction, and report metrics of quality for reverse-engineered fonts.



Formula to Compute the Area

$$S_{0i} = \begin{vmatrix} x_0 & x_1 & x_2 & \cdots & x_i & x_0 \\ y_0 & y_1 & y_2 & \cdots & y_i & y_0 \end{vmatrix}$$

which can be expanded to

$$S_{0i} = \sum_{p=0}^{i} (x_p y_{p+1} - x_{p+1} y_p) + x_i y_0 - x_0 y_i. \qquad \text{[Shao 1996]}$$

Figure 5: Concepts and computation of cord-curve area response

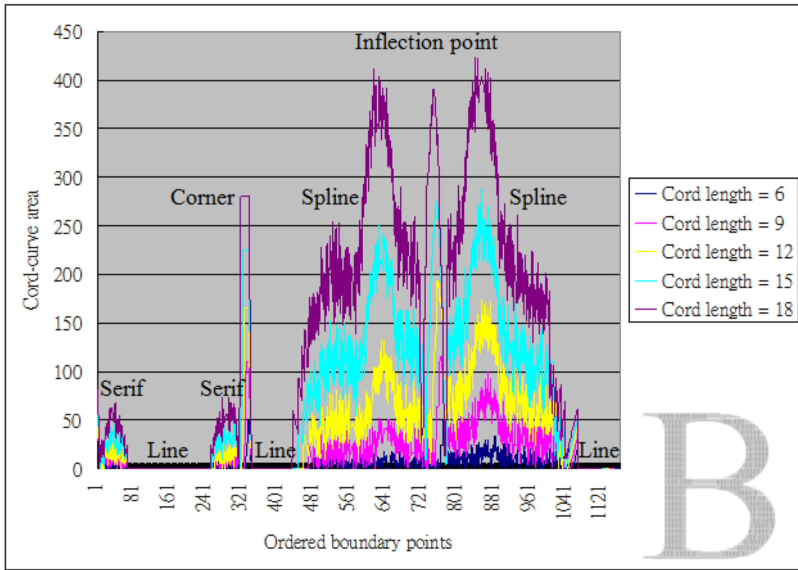
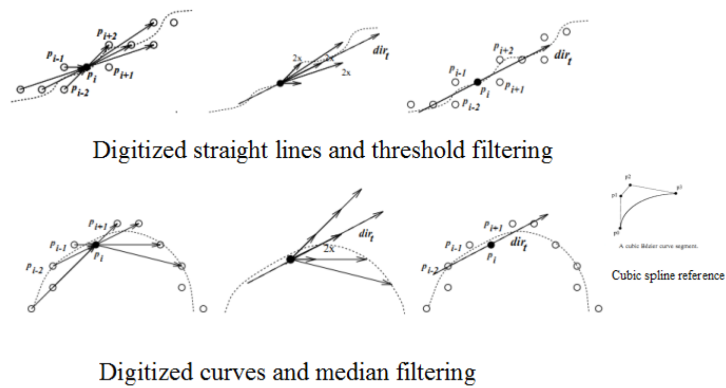


Figure 6: Cord-curve area response of the outer part of alphabet B



[Matas 1995, Shao1996]

Figure 7: Useful filters to counteract quantization noise

Input: A list for each connected component:
List1: ((x11, y11), (x12, y12), (x13, y13), ..., (x11, y11)),
List2: ((x21, y21), (x22, y22), (x23, y23), ..., (x21, y21)),
cord length L
Output: Intermediate representations marked cord-curve area
for all connected components **do** {1st phase}
set up data structures for finite differencing of ordered boundary points;
compute cord-curve area response with cord length L [Shao 1996];
end for

Algorithm 4: Computation of cord-curve area response

Input: A list for each connected component:
List1: ((x11, y11), (x12, y12), (x13, y13), ..., (x11, y11)),
List2: ((x21, y21), (x22, y22), (x23, y23), ..., (x21, y21)),
cord length L,
threshold θ, threshold δ
Output: Intermediate representations marked with straight line components

```
for all connected components do {1st phase}
  calculate cord-curve area response with cord length L;
  2-level threshold filtering of line segments;
  if cord-curve area ≤ threshold θ then
    counter++;
    if counter ≥ threshold δ then
      mark as LINE with START and END;
    end if
  end if
end for
```

Algorithm 5: Straight line analysis for compression of control points

Input: A list for each connected component:
List1: ((x11, y11), (x12, y12), (x13, y13), ..., (x11, y11)),
List2: ((x21, y21), (x22, y22), (x23, y23), ..., (x21, y21)),
cord length L,
median filter tap M
Output: Intermediate representations marked with spline components

```
for all connected components do {2nd phase}
  line analysis;
  if ∃ a set of points between LINE END and next LINE START then
    estimate SPLINE START and END based on cord-curve area response;
    tap-M median filtered estimation of initial tangent vectors;
    try to fit with a SPLINE [Schneider 1990];
    report maximum error and MSE;
  end if
end for
```

Algorithm 6: Spline analysis for compression of control points

Input: Connected components with ordered boundary points, and marked with spatial containment relations
Output: METAFONT representations of font images

```
for all connected components do
  if compression level 0 then
    write all control points;
  end if
  if compression level 1 then
    write LINE START and END for LINE; write all other control points;
  end if
  if compression level 2 then
    write SPLINE START, CONTROL POINT 1 & 2, SPLINE END for SPLINE;
    write all other control points;
  end if
  if compression level 3 then
    write LINE START and END for LINE; write SPLINE START, CONTROL
    POINT 1 & 2, SPLINE END for SPLINE; write all other control points;
  end if
end for
```

Algorithm 7: Generation of METAFONT representations

3.3 Experimental results

With higher levels of compression, we can obtain higher levels of CP reduction with the price of larger MSE. A short demonstration of the reverse-engineered alphabet B are shown in Figure 8 and 9, the normalized CP reduction and the associated ME and MSE. As we can see, the total number of control points are reduced significantly, while still maintaining acceptable MSE (pixel units). For level-1 compression, we achieve over 30% percent reduction of control points with ME = 0.5 and MSE = 0.0002. For level-3 compression, we obtain over 90% percent reduction of control points with ME = 260.04 and MSE = 16.17. We can see most errors are the contribution of spline fitting. Visible artefacts of sharp splines and smoothed serifs can be observed for the outer parts of alphabet B, and smoothed corners can be discerned for the inner parts of alphabet B. This motivates several future directions of improvement for Reverse METAFONT compiler and we address them in section 5.

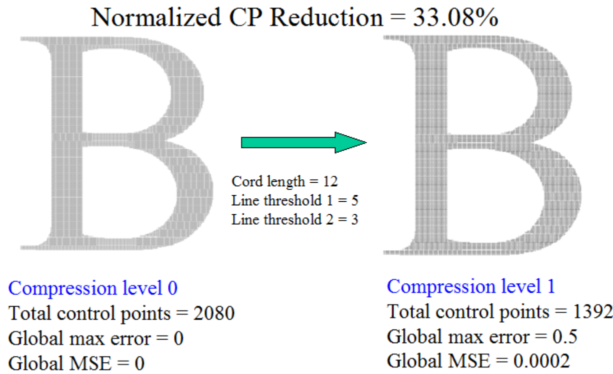


Figure 8: Compression of straight line components

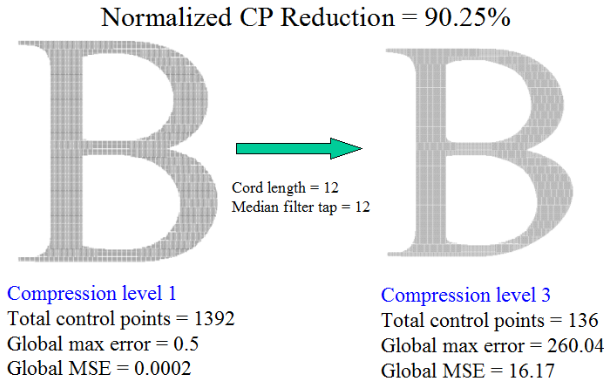


Figure 9: Compression of spline components

3.4 Discussion

Till this point, we have demonstrated that Reverse METAFONT compiler can generate efficient METAFONT representations, with normalized CP reduction over 90%. For the highest level of compression, the corresponding ME and MSE are sometimes high due to spline fitting. The issue is that the fewer control points we use to model the fonts, the larger deviation we might get comparing to the original font images. We can understand this as a trade-off between efficient representations and accuracy, an analogy of the classical rate-distortion relation of source coding.

There are several different ways to calculate ME and MSE for curve fitting. We adopt the parametrized method proposed by [Schneider90]. One benefit of this method is that slight deviation of tangent vectors can be reflected by large errors, which is good for metrics of curve fitting. However, the error terms are sometimes unreasonably large when the

parametrization process produces a very small term in the denominator and this may be confusing. Using MSE could be a good choice as errors are averaged among all control points, but this has the disadvantage that locally large errors might appear small globally.

As a remark, to the best of our knowledge, Algorithm 5 and Algorithm 6 are not seen in any previous research. We find it interesting that these algorithms, although simple, can really do efficient reverse font modeling.

4 Evaluation

4.1 Overview

In section 2 & 3, we have shown Reverse METAFONT compiler can generate METAFONT representations and perform optimization with trade-off between efficiency and accuracy. Nevertheless, we have only shown the results for limited cases. In the following subsections, we enlarge our scope and present the results for six alphabets in our selected benchmark. We first show the normalized CP reduction breakdown for different compression levels. Second, we show the MSE breakdown for different compression levels. We omit ME because it is difficult to have meaningful comparisons of ME for different alphabets. The results are obtained by using an iterative process to find suitable parameters [Lee12]. These parameters are different for each alphabet, and the searching process is a different story for each alphabet. We present this reverse engineering process in algorithmic form in Algorithm 8. This reverse engineering process is presently manual, but we envision that it could become automatic in the future.

Input: Font images
Output: METAFONT representations of font images

```
while results not good do
  Compile font images with Reverse METAFONT compiler, version 0.000001;
  Synthesize with METAFONT compiler, version 2.718281;
  See the synthesized alphabets;
  if results not good then
    // tune the parameters using some experience rules
    Tune code length L, increase L for complex typographic structures;
    Tune threshold θ, increase θ if line is not perfect horizontal or vertical;
    Tune threshold δ, increase δ if line boundary is noisy;
    Tune median filter tap M, increase M if tangent is slowly varying;
  else
    break;
  end if
end while
Ship the METAFONT representations;
```

Algorithm 8: Reverse engineering process of fonts

We coarsely classify alphabets into three representative groups to have a qualitative understanding of the performance metrics of different groups of alphabets. For line-and-spline dominated group, we select alphabets having large line and spline components, and alphabets A and B are chosen. For spline dominated group, we select alphabets having large spline components, and alphabets C and D are chosen. For disconnected group, we select alphabets having disconnected components, and alphabets i and j are chosen.

4.2 Normalized control points reduction

As shown in Figure 10, normalized control points reduction are significant for all three groups. With level-1 compression, group 1 achieves 35% to 40% reduction, group 2 achieves 5% to 35% and group 3 achieves 25% to 31% reduction. With level-2 compression, group 1 achieves 51% to 61% reduction, group 2 achieves 55% to 78% and group 3 achieves 43% to 46% reduction. With level-3 compression, group 1 achieves 88% to 95% reduction, group 2 achieves 83% to 91% and group 3 achieves 68% to 78% reduction.

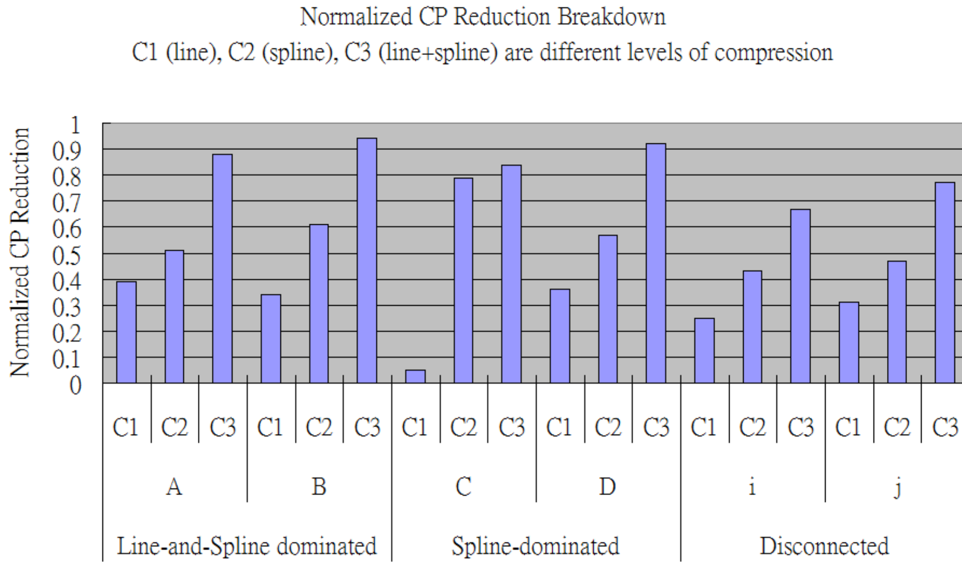


Figure 10: Normalized CP reduction breakdown for different compression levels

4.3 Maximum error & mean-squared-error

As shown in Figure 10, MSEs (pixel units) are growing as we use higher levels of compression for all three groups. With level-1 compression, group 1 has MSE almost 0, group 2 has MSE 0 and group 3 has MSE 0. With level-2 compression, group 1 has MSE 2.19 to 16.6, group 2 has MSE 19.61 to 51.68 and group 3 has MSE 8.54 to 13.26. With level-3 compression, group 1 has MSE 2.28 to 16.6, group 2 has MSE 19.62 to 51.68 and group 3 has MSE 8.54 to 13.26.

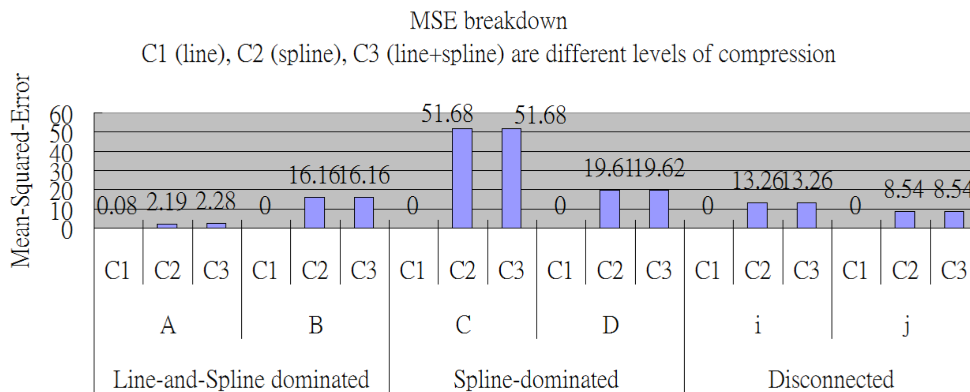


Figure 11: MSE breakdown for different compression levels

4.4 Discussion

As we see in Figure 10 & 11, normalized CP reduction and MSE for three different groups largely follow the same trend: higher levels of compression of control points yields larger MEs and MSEs. Alphabet C is an interesting case, as it has large fractions of boundary points covered by splines. With compression of splines, the reduction of control points is very significant, whereas the compression of lines only contributes to 5% reduction. Also notice that MSE of alphabet C is also the largest. This can be understood as errors are contributed by large factions of control pints covered by splines during spline fitting. In addition, using level-3 compression, we find the normalized CP reduction for the disconnected group (alphabets i and j) is less than the other two groups. This can be explained by large lines and splines are not found in this group and circles are hard to model with spline fitting.

5 Conclusions and future work

Reverse digital typography is a promising new-born field. Growing out of the success of classical digital typography, reverse digital typography adds new blends of signal processing and cloud computing applications. This new field can complement classical digital typography to form a complete cycle of electronic publishing, and provides a new infrastructure for signal processing research on different aspects of electronic publishing. In addition, parametric representations as the output of reverse digital typography can provide scalability to all kinds of resolutions and enhance perceptual experiences to a new level by hiding the walls of pixels from users.

Central to the software stack of reverse digital typography is the Reverse METAFONT compiler that can take raster-scanned font images as input and generate their METAFONT representations as output. In this research report, we present the design and implementation of Reverse METAFONT compiler. We report the architecture of Reverse METAFONT compiler, and its performance on the Times New Roman benchmark. Reverse METAFONT compiler offers a set of parameters to fine tune the reverse engineering process, and different levels of compression to make trade-off between efficiency and accuracy. Experimental results have shown that with suitable selection of parameters, 15% to 45% control points reduction can be achieved with level-1 compression, and 65% to 95% reduction of control points can be attained with level-3 compression.

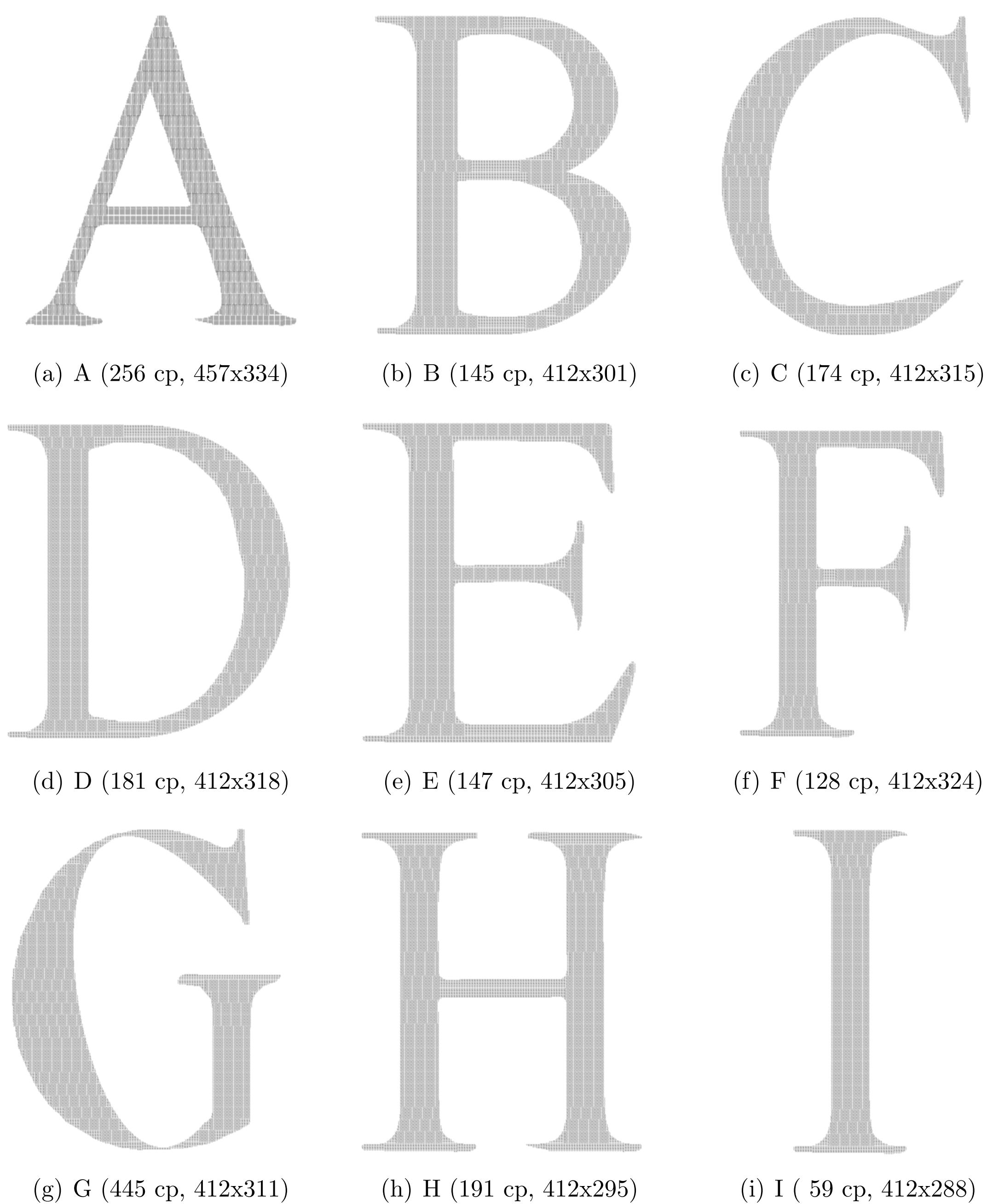
Nevertheless, MSEs of level-3 compression are sometimes high due to spline fitting. Several research directions are likely to further minimize MSE. First, feature point detection algorithms might be useful as important feature points such corners and inflection points mark the start and end of lines and splines. Thus a robust estimation of these feature points might aid the compression algorithms to do more precise boundary segmentation. Second, recursive and weighted spline fitting might be useful as past research works have

shown this is an effective way to optimize splint fitting. Third, although Reverse METAFONT compiler provides a set of parameters to tune the reverse engineering process. The process is currently manual and is time consuming. Thus, automatic closed-loop optimization of parameters might ease the time-consuming parameter tuning process. Finally, all configurable parameters in current Reverse METAFONT compiler are set globally, and it might be useful to investigate the possibilities of developing a multi-resolution analysis and synthesis framework to perform local adaptation of these parameters.

Current Reverse METAFONT compiler can successfully perform reverse modeling of fonts by generating METAFONT representations of the font images. These METAFONT representations can then be compiled by classical METAFONT compiler or convert to PostScript representations for a wide range of applications. A natural extension of the current Reverse METAFONT compiler would be to consider the reverse modeling of pages, namely, taking page images as input and generate their LaTeX representations as output. This extension is very promising and deserves serious investigations in the future.

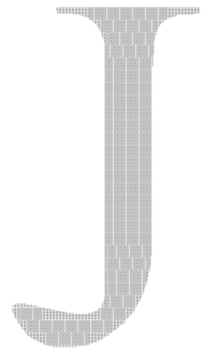
6 The reversed-engineered Times New Roman alphabets

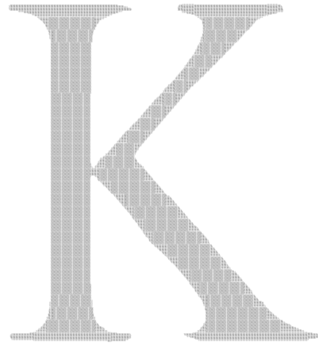
In this section, we list the complete 52 reverse-engineered Times New Roman alphabets using level-3 compression. Also, we list the number of control points and input image size under each alphabet. The scripts for generating METAFONT representations and for DVI rendering are enclosed in the release package.

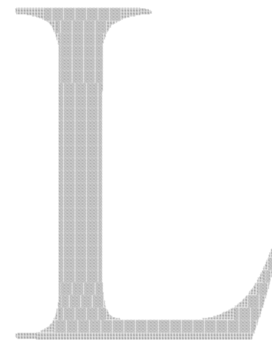


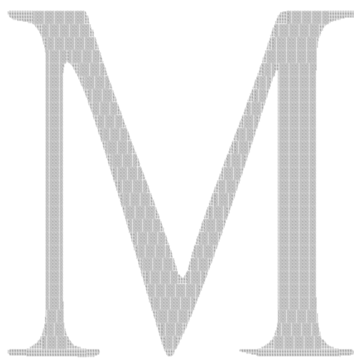
(a) A (256 cp, 457x334)
(b) B (145 cp, 412x301)
(c) C (174 cp, 412x315)
(d) D (181 cp, 412x318)
(e) E (147 cp, 412x305)
(f) F (128 cp, 412x324)
(g) G (445 cp, 412x311)
(h) H (191 cp, 412x295)
(i) I (59 cp, 412x288)

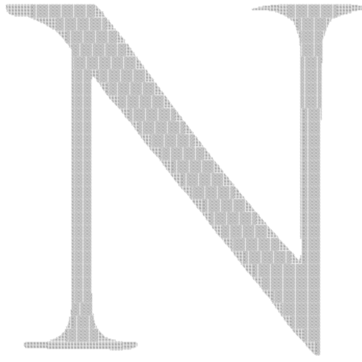
Figure 12: The reverse-engineered Times New Roman Alphabets (A-I)

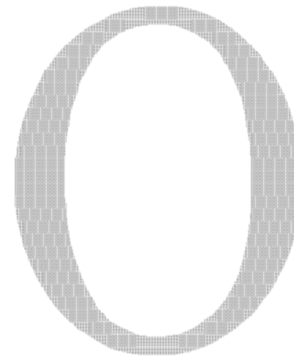

(a) J (94 cp, 412x319)

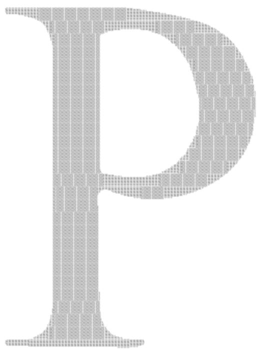

(b) K (177 cp, 412x313)

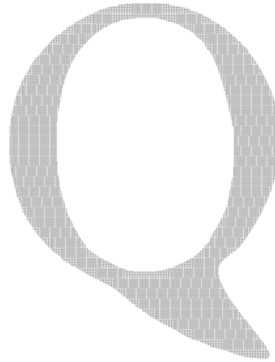

(c) L (74 cp, 412x319)


(d) M (206 cp, 412x305)


(e) N (254 cp, 303x297)


(f) O (490 cp, 412x306)


(g) P (145 cp, 412x304)


(h) Q (594 cp, 412x384)

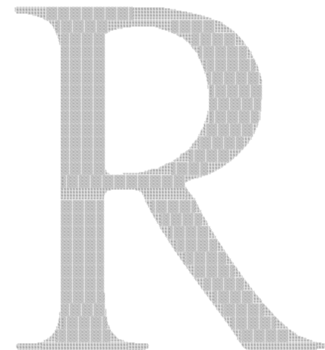
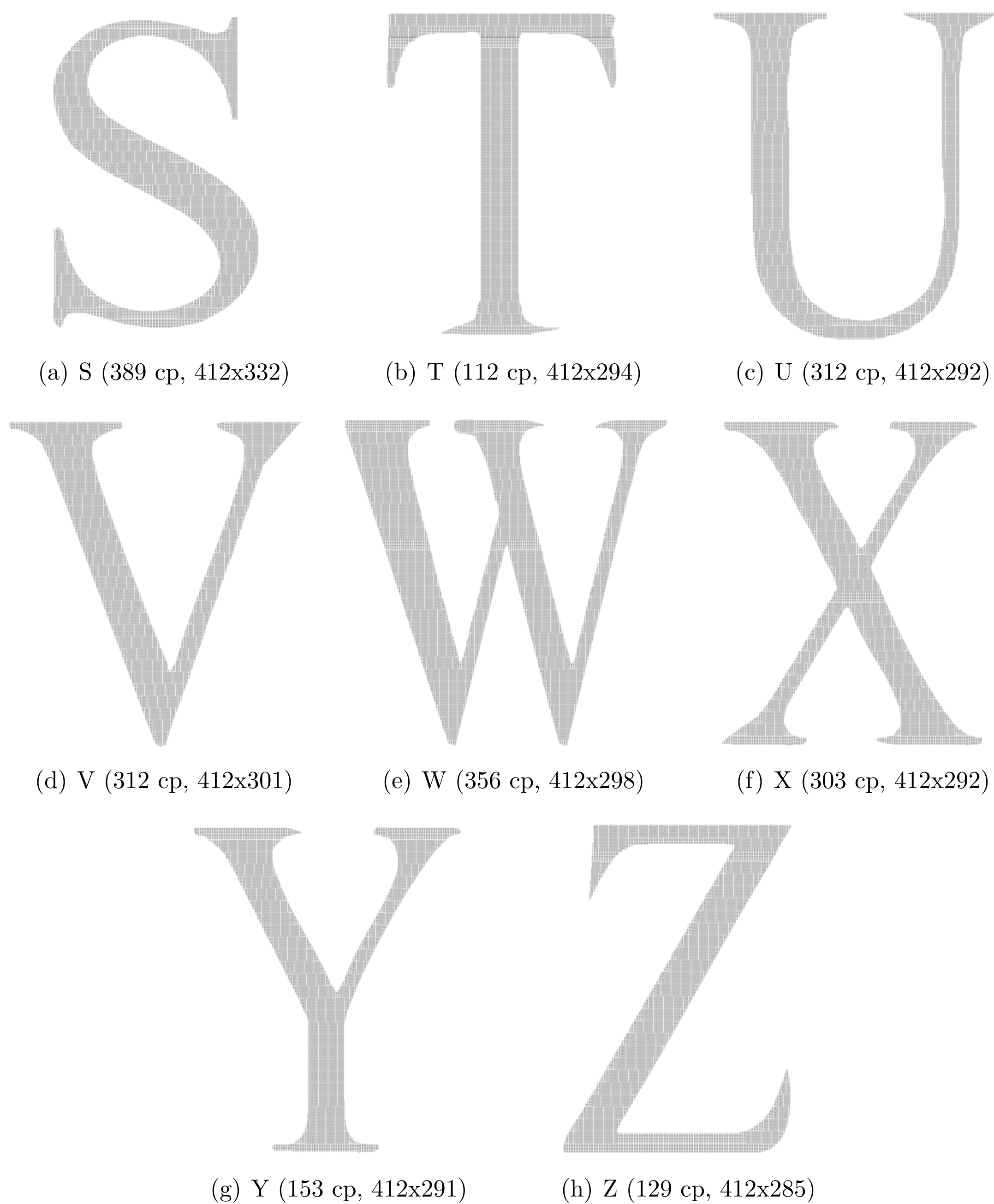

(i) R (192 cp, 412x306)

Figure 13: The reverse-engineered Times New Roman Alphabets (J-R)

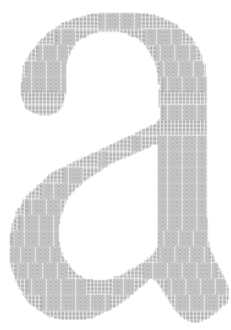


(a) S (389 cp, 412x332) (b) T (112 cp, 412x294) (c) U (312 cp, 412x292)

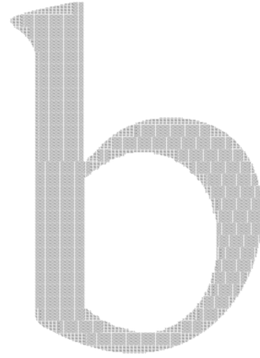
(d) V (312 cp, 412x301) (e) W (356 cp, 412x298) (f) X (303 cp, 412x292)

(g) Y (153 cp, 412x291) (h) Z (129 cp, 412x285)

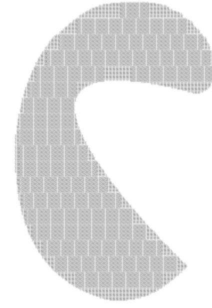
Figure 14: The reverse-engineered Times New Roman Alphabets (S-Z)



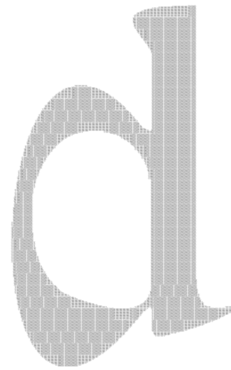
(a) a (395 cp, 412x279)



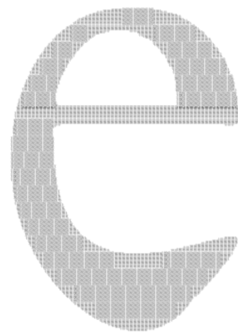
(b) b (481 cp, 412x312)



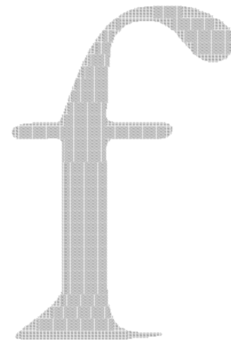
(c) c (149 cp, 412x279)



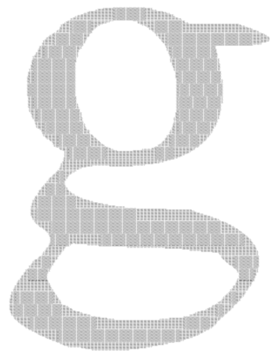
(d) d (224 cp, 412x327)



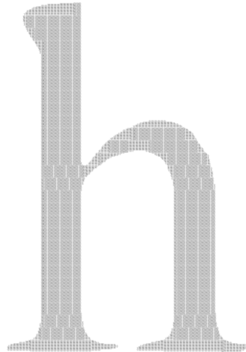
(e) e (286 cp, 412x279)



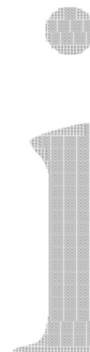
(f) f (252 cp, 412x348)



(g) g (401 cp, 412x319)

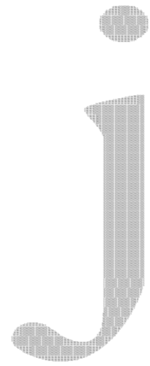


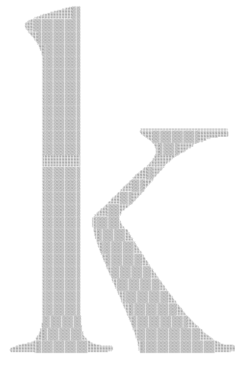
(h) h (207 cp, 412x309)

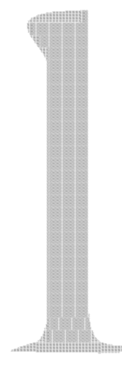


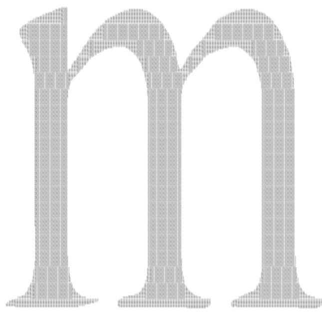
(i) i (254 cp, 412x318)

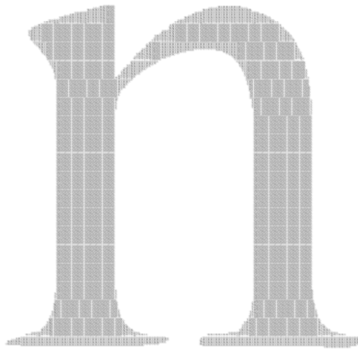
Figure 15: The reverse-engineered Times New Roman Alphabets (a-i)

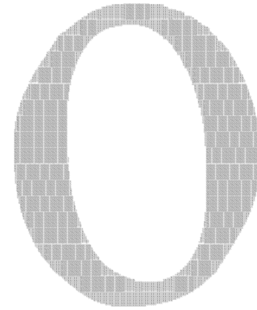

(a) j (231 cp, 412x401)

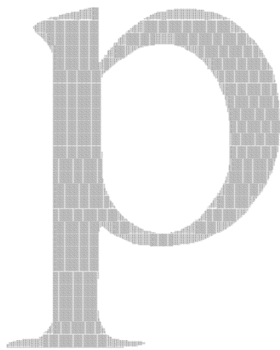

(b) k (273 cp, 412x314)

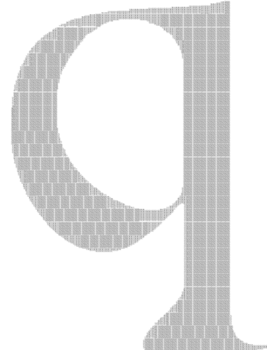

(c) l (106 cp, 412x318)


(d) m (345 cp, 457x279)


(e) n (264 cp, 212x214)


(f) o (298 cp, 412x279)


(g) p (385 cp, 412x330)


(h) q (225 cp, 412x296)

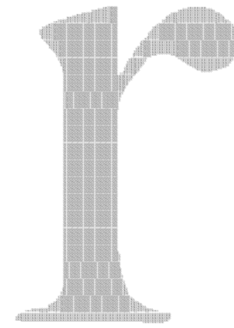
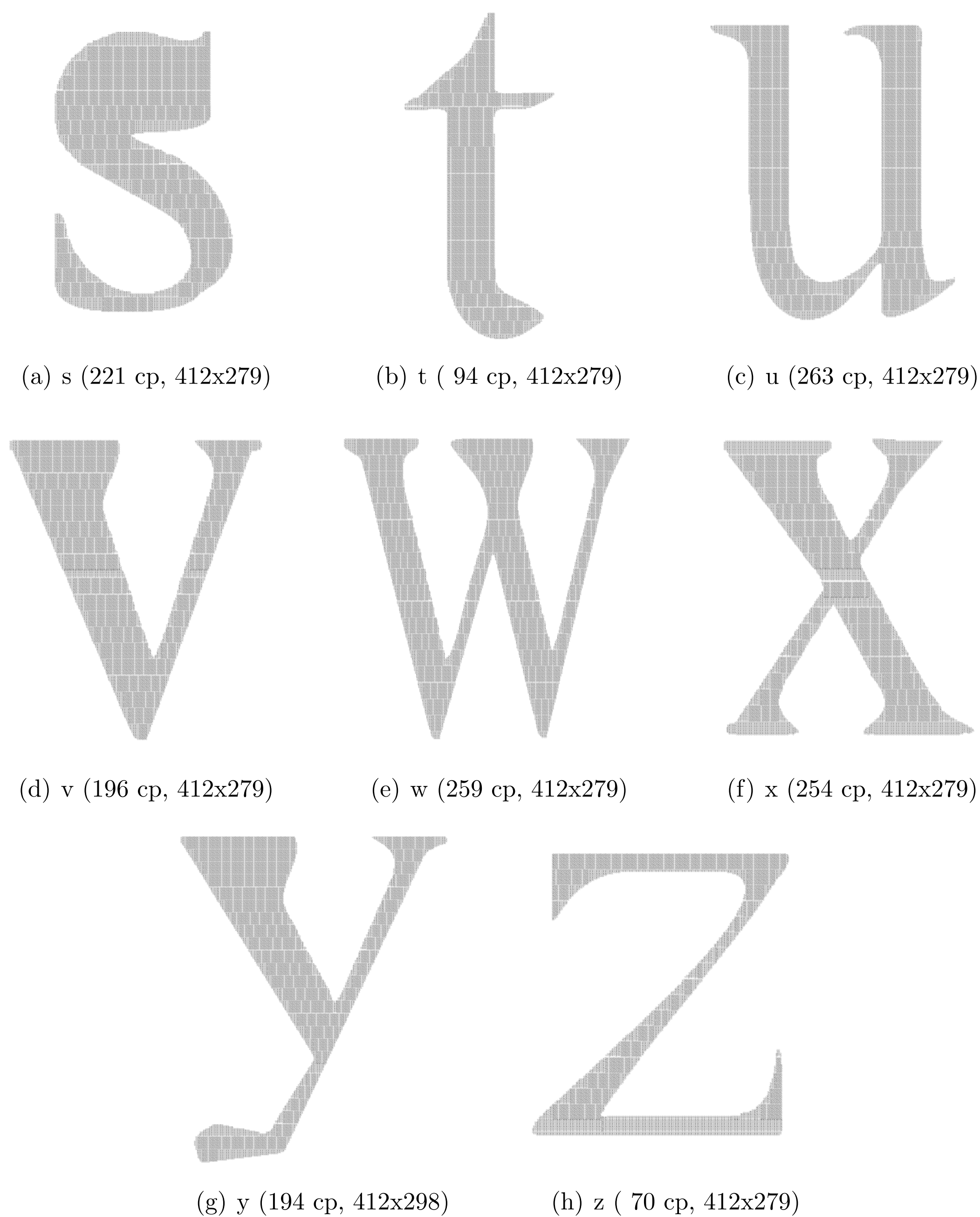

(i) r (211 cp, 412x279)

Figure 16: The reverse-engineered Times New Roman Alphabets (j-r)



(a) s (221 cp, 412x279) (b) t (94 cp, 412x279) (c) u (263 cp, 412x279)

(d) v (196 cp, 412x279) (e) w (259 cp, 412x279) (f) x (254 cp, 412x279)

(g) y (194 cp, 412x298) (h) z (70 cp, 412x279)

Figure 17: The reverse-engineered Times New Roman Alphabets (s-z)

References

[Haralambous93] Y. Haralambous. *Parametrization of PostScript fonts through METAFONT — an alternative to Adobe Multiple Master fonts.* Electronic Publishing, Vol.

6(3), 145–157, September 1993.

[Shamir96] A. Shamir and A. Rappoport. *Extraction of Typographic Elements from Outline Representations of Fonts.* Computer Graphics Forum, Vol. 15, 259-268, 1996.

[Hertz98] J. Hertz, C. Hu, J. Gonczarowski, and R. D. Hersch *A Window-based Method For Automatic Typographic Parameter Extraction.* Electronic Publishing, Artistic Imaging and Digital Typography, (Eds. R.D. Hersch, J. Andre, H. Brown), LNCS 1375, Springer-Verlag, 44-54, 1998.

[Feng75] H. Y. F, Feng and T. Pavlidis. *The Generation of Polygonal Outlines of Objects from Gray Level Pictures.* IEEE Transactions on Circuits and Systems, Vol. CAS-22, No. 5, May 1975.

[Feng75−2] H. Y. F, Feng and T. Pavlidis. *Decomposition of Polygons Into Simpler Components: Feature Generation for Syntactic Pattern Recognition.* IEEE Transactions on Computers, Vol. c-24, No. 6, June 1975.

[Hesselink01] W. H. Hesselink, A. Meijster and C. Bron. *Concurrent Determination of Connected Components.* Sci. Comp. Programming, 41, pp. 173-194, 2001.

[Schneider90] P. J. Schneider. *An Algorithm for Automatically Fitting Digitized Curves.* Graphics GEMS I Edited by A. S. Glassner.

[Matas95] J. Matas, Z. Shao, and J. Kittler. *Estimation of Curvature and Tangent Direction by Median Filtered Differencing.* 8th Int. Conf. on Image Analysis and Processing, San Remo September, 1995.

[Shao96] L. Shao and H. Zhou. *Curve Fitting with Bézier Cubics.* Graphical Models and Image Processing, Vol. 58, No. 3, May, 223–232, 1996.

[Knuth86] D. E. Knuth. *Computer Modern Typefaces.* Addison Wesley Publishing Company, 1986.

[Lee12] T. C. Lee. *Reverse PostScript.* Project Presentation Slides, EPFL, 2012.

Reverse Typography

Tao C. Lee
Informatique, EPFL

January 20, 2013

Semester Project 2nd year master

Supervised by

Dr. Paolo Prandoni
Professor Martin Vetterli

Lausanne, academic year 2012 – 2013

Abstract

Reverse digital typography addresses the problem of high-quality document reproduction in the print-scan cycle. Central to the technical challenges of reverse digital typography is the resolution-scalable edge reconstruction. The conventional approach to edge reconstruction is Global Spline Fitting (GSF): estimation of control points followed by spline fitting between the control points, which is computation intensive and fragile to the print-scan degradation. In this research, we propose a new approach to edge reconstruction: Quadtree Spline Fitting (QSF), a combination of quadtree segmentation and spline fitting. We demonstrate that this new approach can achieve the same baseline performance compared to GSF in the noise-free case, and can outperform GSF in the noisy cases. We optimize QSF using the prune-join algorithms. and give a Rate-Distortion (R-D) analysis. The denoising power of QSF is explored in the case of blurry edges, and future applications of QSF are discussed.

Keyword: Edge reconstruction, spline fitting, quadtree segmentation, blurry edges, denoising

1 The edge reconstruction problem

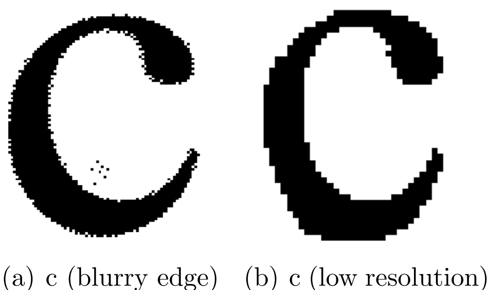
Geometrical images, a class of images that contain most information on the edges, attracted the attention of the signal processing community in two aspects: (i) wavelets perform poorly on the edges [Donoho99, Friedrich04], and (ii) geometrical images are commonly found in several image processing applications [Shukla04, Shukla05]. One approach to geometrical image processing is based on the quadtree segmentation techniques: segmenting the images into different scales of squares, and each square with edge length great than one can have four subsquares of equal size as children. This approach essentially transformed a representation problem into segmentation and optimization problems, and it was proven that this approach could provide suboptimal (nearly min-max) estimation of the edges [Donoho99]. Efficiency and the approximation properties of the quadtree segmentation techniques were studied [Friedrich04], and R-D optimized quadtree algorithms and their denoising and compression applications were reported [Shukla04]. Nevertheless, quadtree segmentation techniques have not found many practical applications outside the academic circle.

Cartoon images, a subset of geometrical images that have edges modeled by piecewise cubic splines, are commonly found in the electronic publishing industry. They are usually associated with fonts, math formulas, and graphics that can scale to different resolutions across peripheral devices. Reconstructing the edges of cartoon images have practical applications in digital typography and have been widely studied [Feng75, Feng75−2, Shamir96, Schneider90, Matas95, Shao96, Lee12]. The mainstream emphasis of edge reconstruction here is GSF. By using a few configurable parameters, GSF first locates the approximate positions of control points, then proceeds to initial tangent estimation, and finally starts the recursive fitting of splines between control points. In essence, GSF targets a global

optimization approach, and the fitting process iterates until the fitting error is below the acceptable threshold. GSF technology was commercialized by Adobe Systems Inc. in the Illustrator's live tracing plugin [Illustrator], and used by worldwide users.

The performance of GSF is highly dependent on the Signal-to-Noise Ratio (SNR) along the edges. If the edges are nearly noise-free, GSF works quite well. On the other hand, if the edges are noisy, GSF breaks down easily. This unfortunate fact poses limitations to the use of GSF. For example, it is widely known that users of Illustrator's live tracing plugin need to clean the edges before running live tracing. Cleaning the edges by pre-filtering might help but inevitably modify the edges. Also, since GSF targets a global optimization approach and all the parameters are configured globally, tracing edge structures at different scales becomes difficult. For instance, it is common that either small structures are sacrificed during live tracing, or unwanted structures induced by the noise are traced. In any case, lacking the ability to counteract edge noise and the flexibility to adapt to multiresolution structures make GSF brittle, and a more robust and effective edge reconstruction framework is desired.

The real situation of edge reconstruction in the print-scan cycle is much more demanding than some toy examples presented by the previous academic research works [Feng75, Feng75−2, Shao96]. Notably, there are two main artefacts induced by the print-scan degradation: (i) blurry edges and (ii) low resolutions. Blurry edges are caused by scanning, where the edge formation process can be modeled by the edge signals convoluted with a point-spread function [Smith98]. The low resolutions come from the resolution limitations of scanning and the size of the scanned materials. These two artefacts make edge reconstruction more challenging as shown in Figure 1, and noise resilience and resolution enhancement are more than required for an effective edge reconstruction framework in the print-scan cycle.

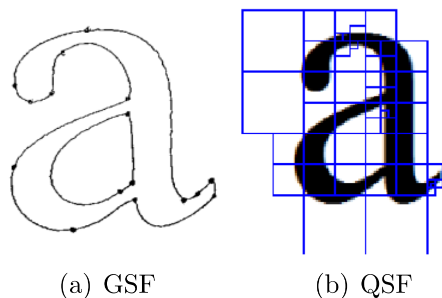


(a) c (blurry edge) (b) c (low resolution)

Figure 1: Edge reconstruction in the print-scan cycle

A rethinking of the edge reconstruction problem leads us to look at the techniques we have with new eyes. Quadtree segmentation and spline fitting are both viable methods for

edge reconstruction, yet with different strengths. In this sense, they are the two faces of the same coin: quadtree segmentation provides a multiscale approximation of the edges and has proven properties in denoising and compression applications, whereas spline fitting provides a vectorized representation of the edges and has proven properties in resolution scaling and parametric synthesis. A tempting question would be how to combine the two techniques together? Can we go from one to the other? When to use quadtree segmentation, when to use spline fitting, and when to use both? These questions do not have a simple answer, and we try to answer these questions using QSF in this research.



(a) GSF (b) QSF

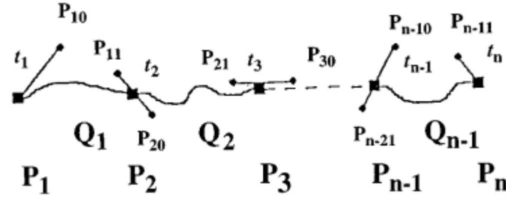
Figure 2: GSF and QSF

This report is organized as follows. We review the benefits and drawbacks of GSF in section 2, and introduce QSF in section 3. In section 4, we present a denoising experiment using both GSF and QSF on scanned images with blurry edges. In section 5, we summarize our results and present future work. In section ??, we present the reconstructed 52 Times New Roman alphabets using QSF.

2 Global spline fitting

2.1 Introduction

A complete description of GSF is out of the scope of this report, and many classical references related to the theory of splines can be found [Schneider90]. A brief description of GSF can be illustrated in 2.1: given a set of discrete points, the task is fit piecewise cubic splines that best approximate these points. The conventional approach to GSF is to first estimate the discontinuities (i.e. control points) between splines, and then fit splines in between. Apparently, the technical challenges of GSF lies in the precise estimation of the control points, which define the position of the discontinuities and the tangents on them.



(a) GSF on discrete points

2.2 Illustrator's live tracing plugin

Illustrator is the flagship software product of Adobe Systems Inc. for computer peripheral applications, including graphics design, image processing, electronic publishing, etc. The live tracing plugin is Illustrator's toolkit to vectorize pixel-based images, using a toolbox of tracing algorithms based on GSF, and some configurable parameters, including *path tightness*, *corner angle*, *minimum area* to control the spline fitting process. There are default values for these parameters, but manual tuning of these parameters is sometimes necessary and experience-based. In theory, the optimal spline fitting can be attained by optimizing these parameters, but automatic optimization is not currently offered by Illustrator.

(b) Noise-free c (c) Recon. c (Illustrator)

Figure 3: GSF using Illustrator (simple tracing, path tightness = 2, corner angle = 20, minimum area = 10)

The result of live tracing Times New Roman letter *c* is shown in Figure 3 with the parameters listed. Letter *c* is chosen on purpose for its simple structure, and its difficulty to trace the inner and outer splines precisely. As we can see, the control points are not precisely estimated by Illustrator, leading to the reconstructed inner and outer edges slightly different from the original ones.

2.3 The reverse METAFONT compiler

Since GSF has been widely studied [Feng75, Feng75−2, Shamir96, Schneider90, Matas95, Shao96], a good practice is to retrod the steps of the previous research works and get the hands dirty, in order to really understand the strengths and limitations of GSF. For this purpose, we implemented the reverse METAFONT compiler that could compile raster scanned font images to METAFONT representations using GSF technology [Lee12].

The first version of the reverse METAFONT compiler, although not being completely satisfactory, serves as a baseline for improvement. We implemented the QSF algorithms in the new reverse METAFONT compiler, by reusing several components in the old reverse METAFONT compiler. For this reason, it is worthwhile to mention some of the existing algorithms in [Lee12]. Some of the experimental results in [Lee12] are also cited here for a discussion on the strengths and limitations of GSF.

2.4 Algorithms

Algorithm 1 to 7 outline the compilation process of reverse METAFONT compiler. Part of Algorithm 1, 2, 3 will be reused later by QSF. Here we pay particular attention to Algorithm 4, 5, and 6, because they represent the basic elements of GSF. Algorithm 4 is fo the estimation of the control points based on [Shao96]. This algorithm plays the equivalent role of corner detection with parameter *corner angle* in the Illustrator's live tracing plugin. Algorithm 5 and 6 play the equivalent role of spline fitting with parameter *path tightness* and *minimum area* in the Illustrator's live tracing plugin. We separate the fitting of lines from splines in Algorithm 5 for technical reasons. Although lines can be seen as a degenerative case of splines, tiny mismatches of initial tangents could make lines slightly curved, which would make the reconstructed fonts bad-looking. This observation was not taken care of in Illustrator's live tracing plugin, as experiments showed live tracing often produced curved lines in place of lines.

Input: Raster-scanned font images
Output: METAFONT representations of font images

Connected component analysis [Hesselink 2001];
Ordering of boundary points;
Pre-filtering of small ordered groups generated by the ordering algorithm;
Spatial containment testing;
Structure analysis (straight line or spline);
Generate METAFONT representations;

Algorithm 1: Reverse METAFONT compilation

Input: Connected Components labelled (1, 2, ...) on an Image(width, height)
Output: A list for each connected component:
List1: ((x11, y11), (x12, y12), (x13, y13), ..., (x11, y11)),
List2: ((x21, y21), (x22, y22), (x23, y23), ..., (x21, y21)), ...

for all labelled point Image(x, y) **do** {1st pass}
  if NB(8) $\neq$ Image(x, y) **then**
    mark (x, y) as boundary points;
  end if
end for
for all labelled point Image(x, y) **do** {2nd pass}
  if (x, y) is a boundary point **then**
    insert all its NB(4) neighboring boundary points in a List(x, y) = () in a counter-clockwise fashion
  end if
end for
{3rd pass}
S algorithm developed by [Feng 1975]

Algorithm 2: Ordering of boundary points by connected components

Input: Connected components with ordered boundary points, and marked with spatial containment relations
Output: Primitive METAFONT representations of font images

for all connected components **do**
  if marked as outer **then**
    write *fill* and all control points in a cyclic path;
  else
    write *unfill* and all control points in a cyclic path;
  end if
end for

Algorithm 3: Generation of primitive METAFONT representations

Input: A list for each connected component:
List1: ((x11, y11), (x12, y12), (x13, y13), ..., (x11, y11)),
List2: ((x21, y21), (x22, y22), (x23, y23), ..., (x21, y21)),
cord length L
Output: Intermediate representations marked cord-curve area

for all connected components **do** {1st phase}
  set up data structures for finite differencing of ordered boundary points;
  compute cord-curve area response with cord length L [Shao 1996];
end for

Algorithm 4: Computation of cord-curve area response

Input: A list for each connected component:
List1: ((x11, y11), (x12, y12), (x13, y13), ..., (x11, y11)),
List2: ((x21, y21), (x22, y22), (x23, y23), ..., (x21, y21)),
cord length L,
threshold θ, threshold δ

Output: Intermediate representations marked with straight line components

for all connected components **do** {1st phase}
  calculate cord-curve area response with cord length L;
  2-level threshold filtering of line segments;
  if cord-curve area $\leq$ threshold θ **then**
    counter++;
    if counter $\geq$ threshold δ **then**
      mark as LINE with START and END;
    end if
  end if
end for

Algorithm 5: Straight line analysis for compression of control points

Input: A list for each connected component:
List1: ((x11, y11), (x12, y12), (x13, y13), ..., (x11, y11)),
List2: ((x21, y21), (x22, y22), (x23, y23), ..., (x21, y21)),
cord length L,
median filter tap M

Output: Intermediate representations marked with spline components

for all connected components **do** {2nd phase}
  line analysis;
  if $\exists$ a set of points between LINE END and next LINE START **then**
    estimate SPLINE START and END based on cord-curve area response;
    tap-M median filtered estimation of initial tangent vectors;
    try to fit with a SPLINE [Schneider 1990];
    report maximum error and MSE;
  end if
end for

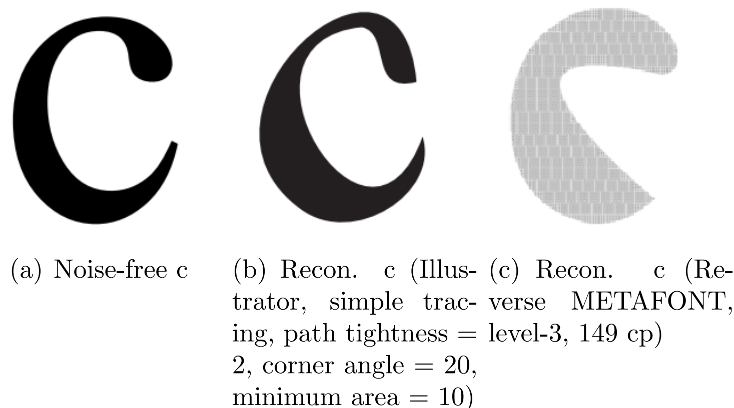
Algorithm 6: Spline analysis for compression of control points

Input: Connected components with ordered boundary points, and marked with spatial containment relations
Output: METAFONT representations of font images

for all connected components **do**
  if compression level 0 **then**
    write all control points;
  end if
  if compression level 1 **then**
    write LINE START and END for LINE; write all other control points;
  end if
  if compression level 2 **then**
    write SPLINE START, CONTROL POINT 1 & 2, SPLINE END for SPLINE; write all other control points;
  end if
  if compression level 3 **then**
    write LINE START and END for LINE; write SPLINE START, CONTROL POINT 1 & 2, SPLINE END for SPLINE; write all other control points;
  end if
end for

Algorithm 7: Generation of METAFONT representations

2.5 Experimental results



(a) Noise-free c

(b) Recon. c (Illustrator, simple tracing, path tightness = 2, corner angle = 20, minimum area = 10)

(c) Recon. c (Reverse METAFONT, level-3, 149 cp)

Figure 4: GSF using Illustrator and Reverse METAFONT

Complete results of edge reconstruction using GSF can be found in [Lee12]. Here we only compare the reconstruction of letter c using GSF with Illustrator's live tracing plugin and the reverse METAFONT compiler as shown in 4. As we can see, different implementations of GSF can produce different reconstructions as the case of letter c and others [Lee12].

For letter c reconstruction, the GSF implementation of Illustrator outperforms that of the reverse METAFONT in the estimation of control points, as the inner edges are better fitted in Figure 4 (b) than (c). Both reconstructions, however, are not faithful compared to the original one in Figure 4 (a), as the inner and outer splines do not conform to the original ones and the round shape typefaces of the joints between inner and outer edges are distorted.

Thus we see even in this almost noise-free case, it is still not an easy thing to produce faithful reconstructions of the typefaces using GSF. It is of course possible that we can scan all possible range of the configurable parameters to see if we can get more faithful reconstructions, but the manual optimization of these parameters is tedious, and even if an automatic optimization procedure is implemented, the worst complexity would be proportional to the number of edge points to the power of number of configurable parameters. Besides the inherent complexity of optimization, a more serious problem is defining visual-quality metrics as the objectives of optimization, often such metrics are typeface-dependent, and no simple object functions serve for all typefaces.

$$complexity \propto O(\text{number of edge points}^{(\text{number of configurable parameters})}) \tag{1}$$

One explanation of this inherent difficulty comes from the nature of GSF: a global optimization problem. GSF uses control points estimation as the first step of reconstruction, but the estimation is based on certain heuristics and different parameters would produce different results. Iterative optimization of the estimation process is highly dependent on the visual-quality objectives and proportional to the number of edge points considered. Usually there is no global guarantee on the convergent speed of this optimization problem.

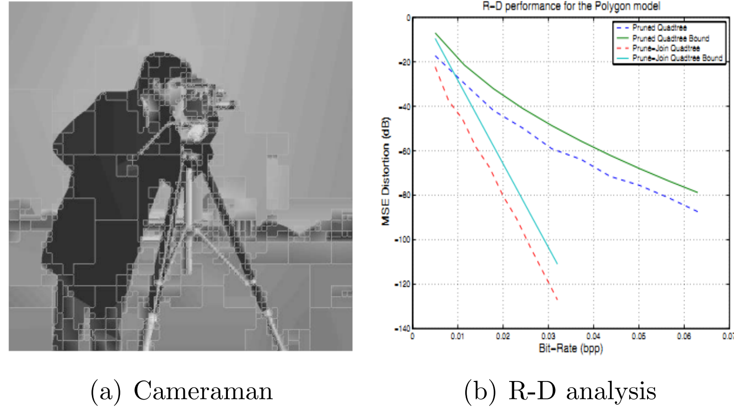
Following this line of reasoning, if we can decompose the global optimization problem into suboptimal local optimization problems, then we have a natural reduction on the complexity. Also, the distortion can be bounded locally, and R-D optimization strategies can be studied. These observations lead to the idea of QSF.

3 Quadtree spline fitting

3.1 Introduction

Quadtree segmentation algorithms and R-D analysis

Quadtree segmentation was proposed as a way to represent images [Donoho99, Friedrich04, Shukla04]. Its approximation properties can be studied in the R-D analysis as illustrated Figure 5. Typically, with a given rate (bit budget), the lower distortion the algorithm can achieve the better. It was reported that for geometrical images, quadtree segmentation algorithms could outperform wavelets-based algorithms (e.g. JPEG2000) [Friedrich04, Shukla04].

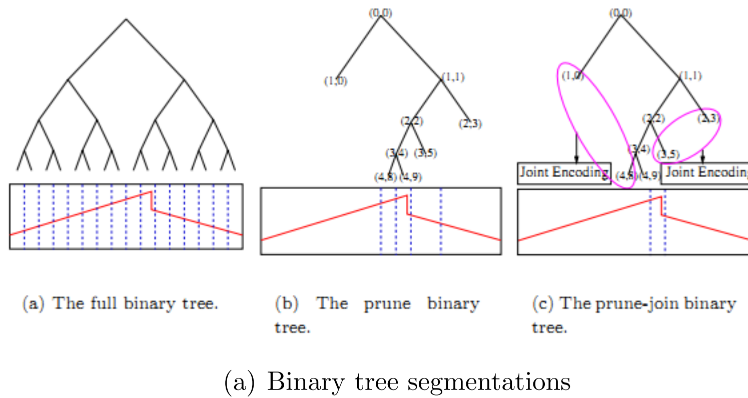


(a) Cameraman (b) R-D analysis

Figure 5: Quadtree segmentation algorithms and R-D analysis

Towards R-D optimized algorithms

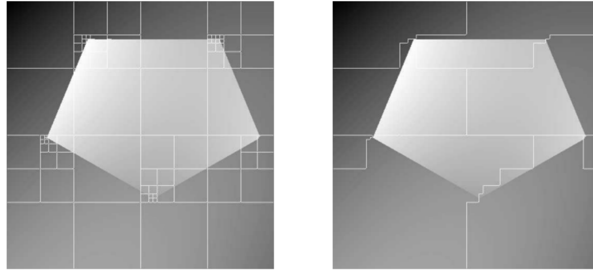
Improvement of quadtree segmentation algorithms was focused on the optimization of the R-D performance. Typically, a tree-based algorithm can be improved by pruning its redundant nodes, or joining similar nodes. The binary tree case is illustrated in Figure 6, and an example for quadtree R-D optimization is shown in Figure 7. For detailed descriptions of these algorithms, see [Shukla04].



(a) The full binary tree. (b) The prune binary tree. (c) The prune-join binary tree.

(a) Binary tree segmentations

Figure 6: Prune and join optimization in the binary tree case



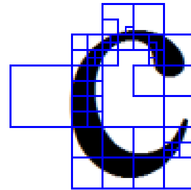
(a) Pruned and joined quadtree images

Figure 7: R-D optimization using prune and prune-join algorithms

The idea of quadtree spline fitting

The idea of QSF comes from the fusion of quadtree segmentation and spline fitting, where quadtree segmentation gives an approximate shape of the typefaces, and spline fitting further gives a local approximation on each quadtree partition. One benefit of this approach is that there is no need to estimate the control points, the positions of the control points are approximated by the quadtree segmentation, and the criterion to determine whether the approximation is good enough is to try to fit a spline at each level of the quadtree, and see if the fitting error at each partition is small enough. Once an acceptable approximation at a certina level is found, spline fitting at that partition is also done. Otherwise, the four quadtree partitions at the next level are considered, and the recursive process continues until the fitting error is small enough or the 1-pixel leaves are reached.

The suboptimal nature of quadtree segmentation gives QSF more control points than GSF, and this is where R-D optimization comes to rescue. In the *join* phase of QSF, we try to join neighboring splines into one spline iteratively, eliminating redundant control points in the process. In principle, the *join* phase reduces the rates but increases the distortion, and good *join* algorithms make trade-offs between the rates and the distortion.



(a) QSF

Figure 8: QSF of letter c (size=128x128, th=5, prune)

3.2 Algorithms

Algorithm 8 describes the main operations of QSF. (i) In initialization, connected component analysis, labeling and ordering of boundary points 2, and quadtree construction are performed. (ii) In threshold-based pruning, we prune the quadtree by eliminating redundant nodes, and fit cubic splines at each node. At each node, we prune the four children if the fitting error is less than the given threshold. Details of spline fitting include tangent scan and straight line fitting on each partition. (iii) In quadtree-to-connected-component mapping, we mark the fitted the splines at the right positions of the connected components 8. (iv) In joining neighboring splines, we join neighboring splines iteratively until no more splines can be joined. (v) In generating METAFONT representations, we output the definitions of the control points and the associated pen path defined by METAFONT *fill* and *unfill* commands 7, 9.

Input: An input font image,
Error threshold δ
Output: METAFONT representations of the font images!
Step 1. Initialization;
Connected component analysis;
Labeling and ordering of boundary points;
Quadtree construction;
Step 2. Threshold-based pruning;
Discard nodes without edge points;
Fit cubic splines (tangent scan) at each node;
Fit a line at each node;
Pick the best fitting (among splines and straight lines) with min. error;
Compare the fitting error with error threshold δ;
if error less than δ **then** mark this node **else** go to next level;
Step 3. Quadtree to connected component mapping
Step 4. Joining neighboring splines (or straight lines)
Fit cubic splines across neighboring splines
if error less than δ **then** pick the new spline; and mark the old splines as joined;
Iterate until no neighboring splines are joined;
Step 5. Generating METAFONT representations

Input: Pruned quadtree, connected components with ordered boundary points, marked with spatial containment relations
Output: connected components marked with quadtree splines

```
for all quadtree node do
  save SPLINE START and SPLINE END;
  for all connected components do
    find the positions of SPLINE START and SPLINE END; mark as the START and
    END of quadtree splines;
  end for
end for
```

Algorithm 8: Quadtree to connected component mapping

Input: Connected components with ordered boundary points, marked with spatial containment relations, marked with quadtree splines
Output: METAFONT representations of font images

```
for all connected components do
  write SPLINE START, CONTROL POINT 1 & 2, SPLINE END for quadtree
  splines; write all other control points;
end for
```

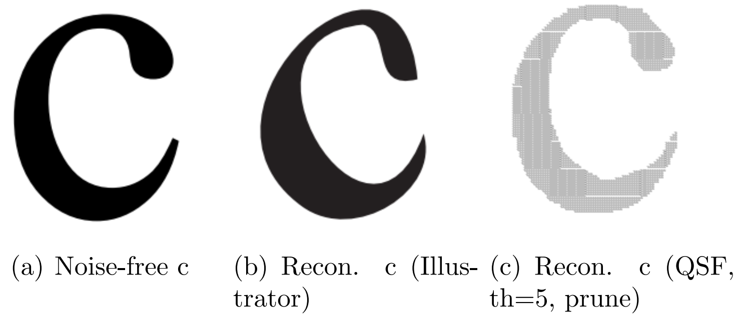
Algorithm 9: Generation of METAFONT representations

3.3 Experimental results

Overview

We classify the experimental results into five categories in order to illustrate the different facets of QSF. (i) In noise-free case, we revisit the reconstruction of letter *c* using GSF and QSF. (ii) In error threshold scan, we show the results of edge reconstruction with different error thresholds. Different error thresholds give different pruned quadtrees, and they in turn determine the distortion of the reconstruction. (iii) In line fitting, we show the visual effects of fitting with lines. This mode is very useful in some line dominated typefaces as discussed in Section 2.4. (iv) In joining, we show the visual effects of joining neighboring splines. This step is crucial to the improvement of R-D performance. (v) In R-D analysis, we compare the R-D performance of letter *c* reconstruction using prune-only and prune-join algorithms with different error thresholds.

Noise-free case

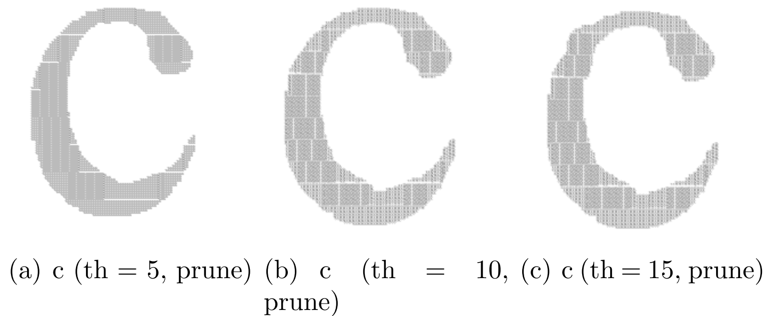


(a) Noise-free c (b) Recon. c (Illustrator) (c) Recon. c (QSF, th=5, prune)

Figure 9: Noise-free reconstruction of letter c with GSF and QSF

As we can see in Figure 9, QSF obviously provides a more faithful reconstruction than GSF. As the inner and outer edges are faithfully reconstructed, and the joint typefaces are preserved. However, we reconstruct this QSF using a rather small error threshold 5, this would produce more control points and result in less resolution scalability at the junctions of small splines.

Edge Reconstruction with Different Error Thresholds

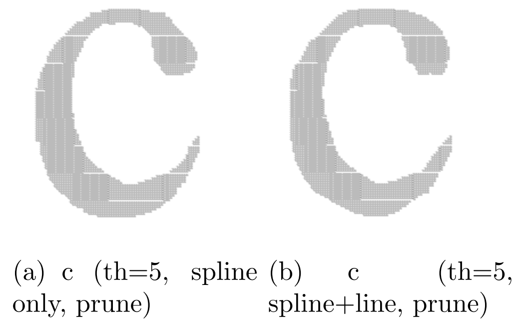


(a) c (th = 5, prune) (b) c (th = 10, prune) (c) c (th = 15, prune)

Figure 10: Edge reconstruction with different error thresholds

As we can see in Figure 10, different error thresholds produce slightly different reconstructions of letter c. The larger the error threshold, the larger the quadtree partitions are retained (larger splines are fitted). However, the distortion is also proportional to the error threshold, as we see the distortion starts to become unpleasant when threshold = 15. In the case of letter c, threshold = 10 is still acceptable with a few bumps along the edges, but most typographic details are still well reconstructed.

Edge Reconstruction with Line Fitting



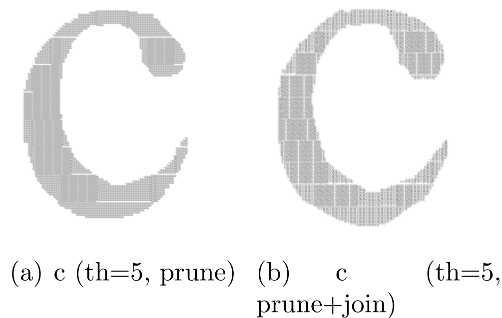
(a) c (th=5, spline only, prune)

(b) c (th=5, spline+line, prune)

Figure 11: Edge reconstruction with straight line fitting

When line fitting is performed on every quadtree partition, its fitting error is compared to that of spline fitting, and it is selected if its error is smaller. As we can see in Figure 11, spline fitting produces some clipping artefacts on the edges. In the case of letter *c*, this is not desirable since there is no line segments on the edges. But for other typefaces with line segments (e.g. letter *A*), line fitting is desirable.

Edge Reconstruction with Join



(a) c (th=5, prune)

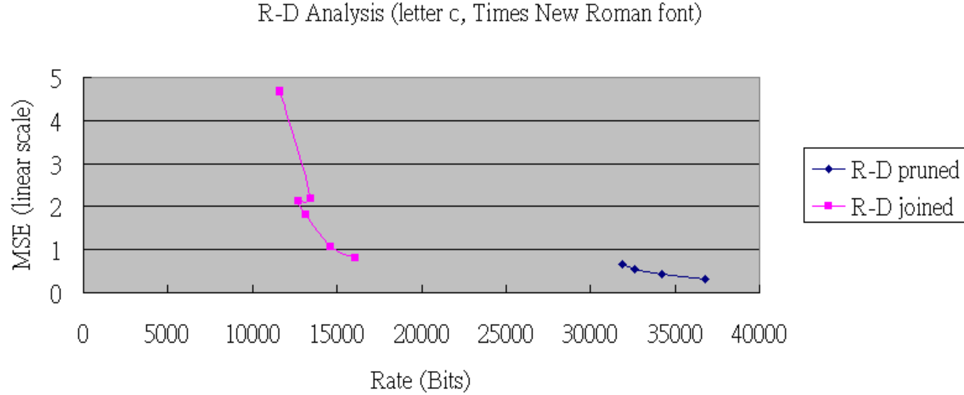
(b) c (th=5, prune+join)

Figure 12: Synthesized Times New Roman Alphabets

As we can see in Figure 12, the *join* algorithm produces more regulated edges as neighboring splines are joined, yet the faithful reconstruction of letter *c* is still preserved. In our current implementation of the *join* algorithm, only very simple join conditions are considered: (i) joining neighboring splines using the left tangent of the left spline and the right tangent of the right spline, (ii) joining neighboring lines using the better tangent of the left line or the right line, and (iii) spline (lines) are joined if the fitting error of the joined splines (lines) is

less than the given error threshold. This simple implementation could be improved in the future as more demanding *join* requirements arise.

R-D Analysis of Edge Reconstruction



(a) R-D analysis (prune, prune+join)

Figure 13: R-D performance of prune and prune-join algorithms (error threshold = 30 to 3, step = 3.3, spline only)

In Figure 13 we show the R-D performance of the prune and the prune-join algorithms in Algorithm 8 with different error thresholds. Distortion is calculated by Mean-Squared-Error (MSE) in linear scale of pixels. The bit rate is calculated as Equation 2. The factor 98 is chosen as an estimation of the bits required to store the definition of control points and the associated pen path.

$$\begin{aligned} \text{bit rate} &\propto O(\text{number of control points}) \\ &= \text{number of control points} * 98 \end{aligned} \tag{2}$$

Figure 13 might not look like an ordinary R-D curve [Friedrich04, Shukla04]. The problem is not now we cannot perform a linear scan on the rates, and linear scan of error thresholds produce nonlinear distribution on the rates. Also we have not derive tight bounds on the R-D performance of QSF. A very loose R-D bound of QSF can be estimated as Equation 3. But experiments showed this bound was so loose compared to the prune and prune-join algorithms that we omit it here.

$$D(R(\delta)) \leq \sum \sqrt{2} * (\text{length of each pruned quadtree node}) \tag{3}$$

where

$$\delta = \text{error threshold}$$

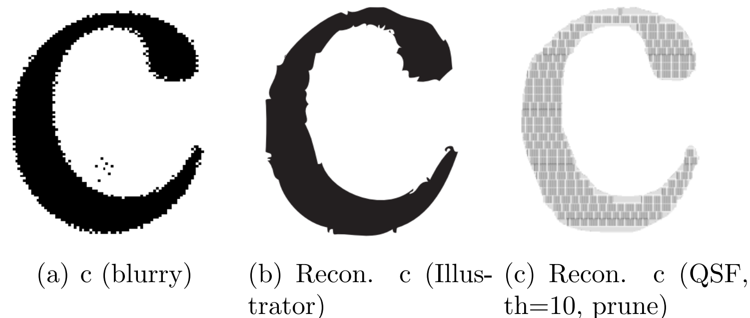
We can see in Figure 13 that the MSEs for letter c using the prune algorithm are below 1, where as MSEs for the prune-join algorithm varies from 5 to less than 1, all with more than 50% of bit rate reduction. This suggests that the optimal R-D operating point is the minimum error threshold 3, using the prune-join algorithm.

4 Denoising experiments: blurry edges

4.1 Introduction

Denoising was a hot topic for image representation research [Friedrich04, Shukla04], partly because it was a good testing ground for the approximation power of the bases under investigation. The fundamental principles of image denoising was studied in the direction that quantization leads to denoising [Natarajan95]. In the case of edge reconstruction, no explicit quantization is presented in GSF, so it is no surprise that GSF is fragile to noisy edges as widely known. For QSF, however, quadtree segmentation gives an implicit quantization before spline fitting, so it is natural that QSF has more capability to combat noisy edges. In the following we perform edge reconstruction on a scanned image with blurry edges to understand the denoising power of QSF.

4.2 Experimental results



(a) c (blurry) (b) Recon. c (Illustrator) (c) Recon. c (QSF, th=10, prune)

Figure 14: Reconstruction of letter c with blurry edges

In Figure 14 we see the reconstructions of letter c using GSF and QSF. Clearly, QSF with error threshold = 10 outperforms GSF. GSF has no power to resist blurry edges, and the estimation of control points is strongly interfered by the blurry edges, which produces funny curly shapes along the edges. QSF, on the other hand, regulate the blurry edges, and retain

a faithful reconstruction of the edges.

We demonstrate the denoising power of QSF in this simple experiment in the hope that denoising using QSF can become a topic that can attract more careful study in the future. To the best of author's knowledge, denoising in edge reconstruction is still an open problem, and QSF has good potential to become an effective solution.

5 Conclusions and future work

This research addresses the deficiencies of GSF in the context of resolution-scalable edge reconstruction, and proposes QSF as a new viable solution. The inherent weaknesses of GSF are studied in two aspects: (i) computation complexity, and (ii) resiliency to noisy edges. Experiments showed that GSF had limitations in achieving faithful edge reconstructions in the noise-free case and noisy cases. Such limitations come from the difficulties of precise estimation of the control points, and lacking quantization schemes to counteract noisy edges.

QSF, a combination of quadtree segmentation and spline fitting, is designed to take advantage of the strengths of both schemes. Quadtree segmentation provides a suboptimal estimation of the control points, and the implicit quantization scheme that can be exploited to denoise the blurry edges. Experiments demonstrated edge reconstruction using QSF under different conditions: (i) different error thresholds, (ii) hybrid line and spline fitting. Optimization of QSF was done by the prune-join algorithm, a R-D analysis of the prune and prune-join algorithms was performed, and an optimal R-D operating point was identified for the example presented.

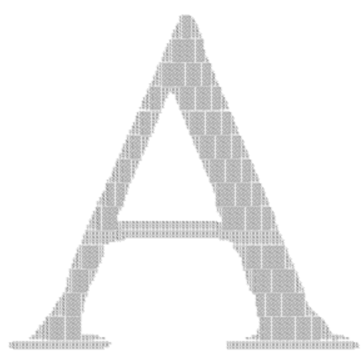
Reverse digital typography targets high-quality document reproduction in the print-scan cycle. For this purpose, we proposed QSF as a new approach to resolution-scalable edge reconstruction, and implemented QSF in the second version of the reverse METAFONT compiler. Experiments showed that QSF achieved the same baseline performance as GSF in the noise-free case, and outperformed GSF in the noisy cases. Benchmarking showed QSF performed well on the Times New Roman fontset. We expected further research in QSF would produce more fruitful results.

Several interesting directions are worth exploring in the future. (i) Improving the efficiency and accuracy of QSF is worth investigating. (ii) More efficient *join* algorithms to improve R-D performance, and tighter R-D bounds of QSF can be explored. (iii) Denoising power of QSF in the print-scan cycle should be further studied, in particular for the print-scan degradation such as blurry edges and low resolutions. (iv) Applying QSF to the reconstruction more complex cartoon images would be an interesting experiment with potential research payback. (v) Using QSF in non-photorealistic rendering might bring a new direction in computer graphics and art design. (vi) Conversion interface for other

vector graphics formats such as PostScript and PDF would make the reverse METAFONT compiler more accessible to a larger audience.

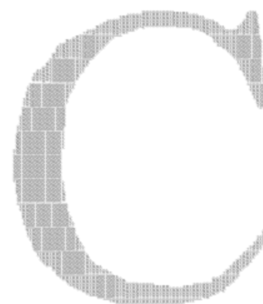
6 The reconstructed Times New Roman alphabets

In this section, we list the complete 52 reconstructed Times New Roman alphabets using QSF. The error thresholds and other parameters can be found in the compilation scripts. The scripts for compiling font image, generating METAFONT representations and DVI rendering are enclosed in the release package.



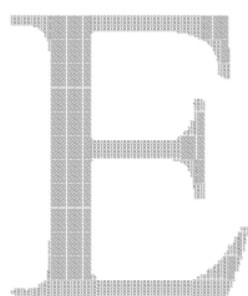
(a) A

(b) B

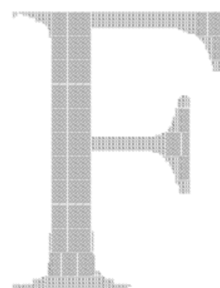


(c) C

(d) D

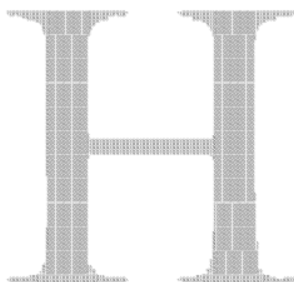


(e) E

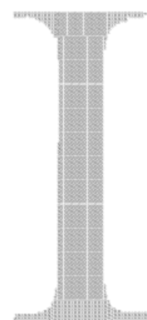


(f) F

(g) G

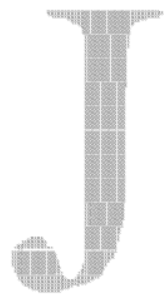


(h) H

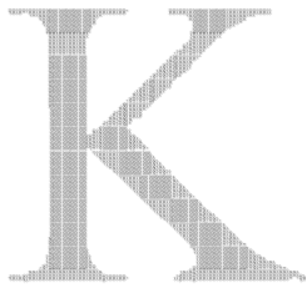


(i) I

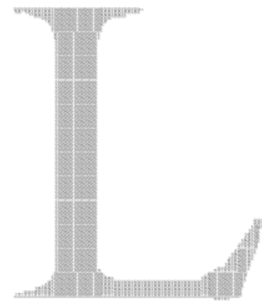
Figure 15: The reconstructed Times New Roman Alphabets (A-I)



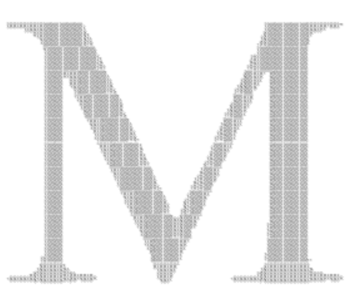
(a) J



(b) K

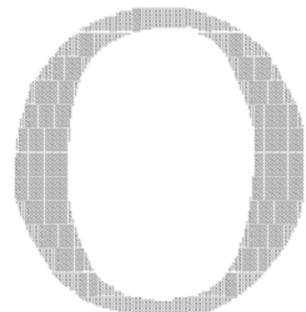


(c) L

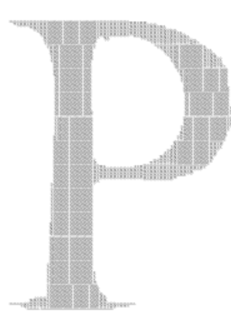


(d) M

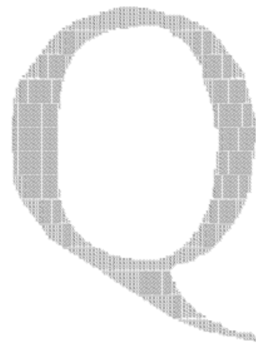
(e) N



(f) O



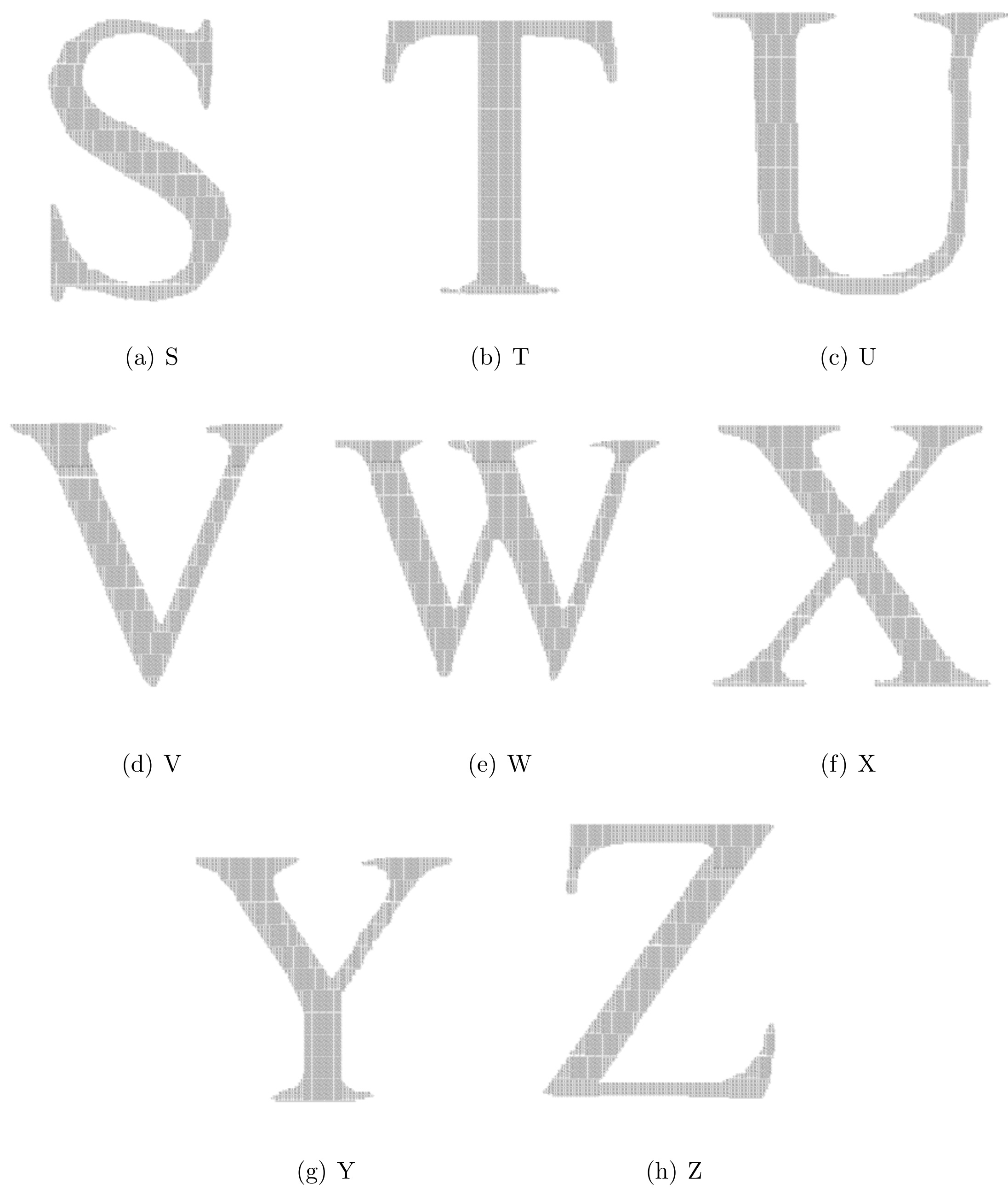
(g) P



(h) Q

(i) R

Figure 16: The reconstructed Times New Roman Alphabets (J-R)



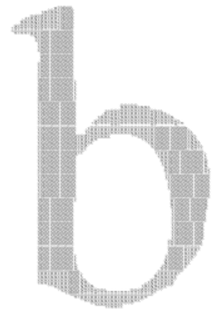
(a) S (b) T (c) U

(d) V (e) W (f) X

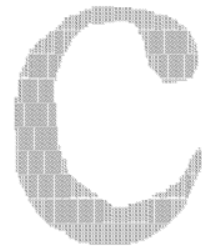
(g) Y (h) Z

Figure 17: The reconstructed Times New Roman Alphabets (S-Z)

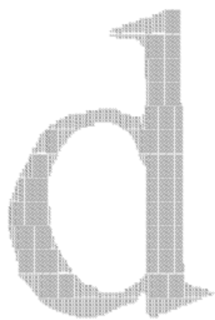
(a) a



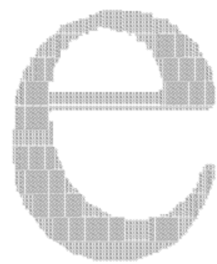
(b) b



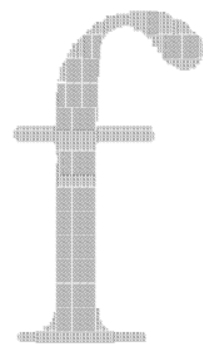
(c) c



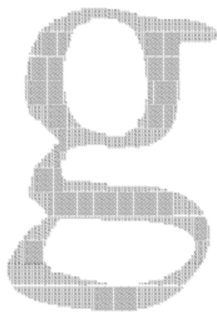
(d) d



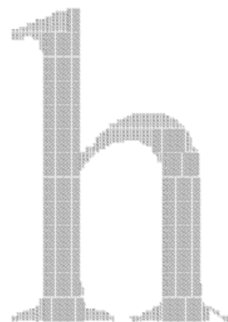
(e) e



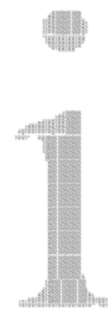
(f) f



(g) g

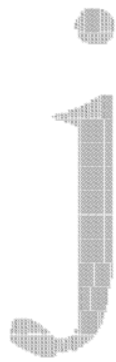


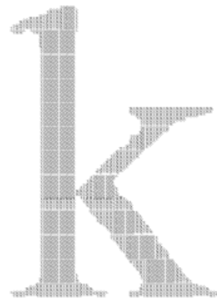
(h) h

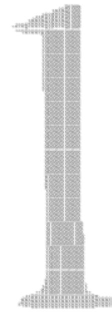


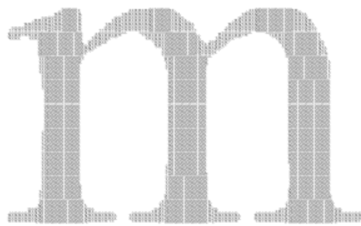
(i) i

Figure 18: The reconstructed Times New Roman Alphabets (a-i)

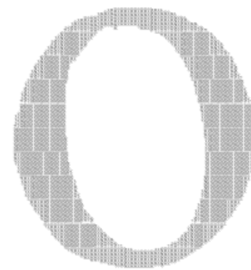

(a) j

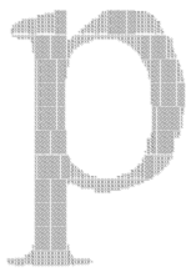

(b) k

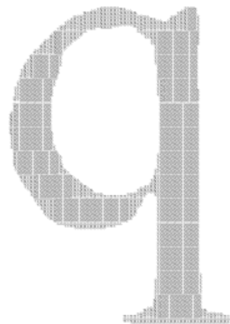

(c) l


(d) m

(e) n


(f) o


(g) p


(h) q

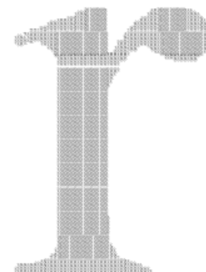
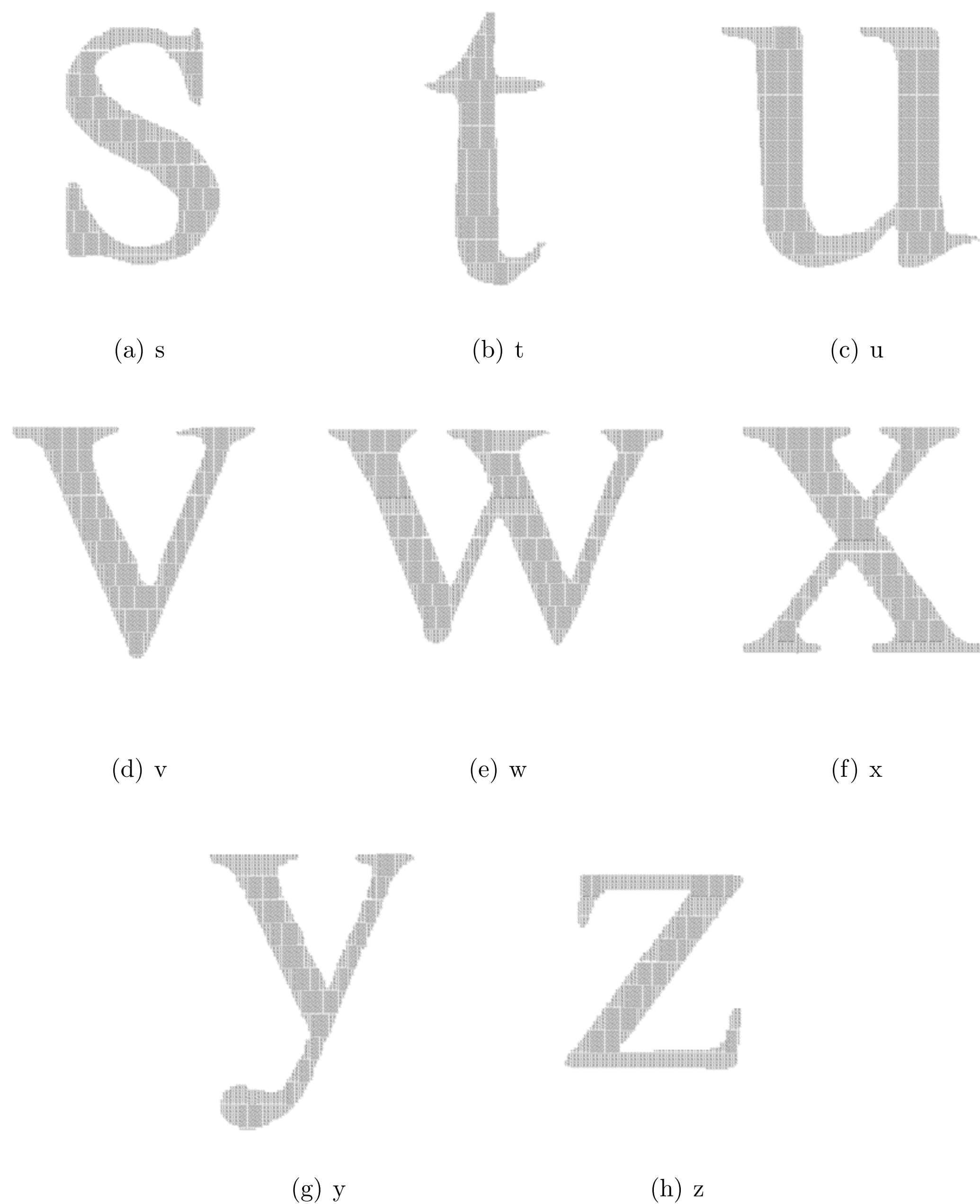

(i) r

Figure 19: The reconstructed Times New Roman Alphabets (j-r)



(a) s (b) t (c) u (d) v (e) w (f) x (g) y (h) z

Figure 20: The reconstructed Times New Roman Alphabets (s-z)

References

[Haralambous93] Y. Haralambous. *Parametrization of PostScript fonts through METAFONT — an alternative to Adobe Multiple Master fonts.* Electronic Publishing, Vol.

6(3), 145–157, September 1993.

[Shamir96] A. Shamir and A. Rappoport. *Extraction of Typographic Elements from Outline Representations of Fonts.* Computer Graphics Forum, Vol. 15, 259-268, 1996.

[Hertz98] J. Hertz, C. Hu, J. Gonczarowski, and R. D. Hersch *A Window-based Method For Automatic Typographic Parameter Extraction.* Electronic Publishing, Artistic Imaging and Digital Typography, (Eds. R.D. Hersch, J. Andre, H. Brown), LNCS 1375, Springer-Verlag, 44-54, 1998.

[Feng75] H. Y. F, Feng and T. Pavlidis. *The Generation of Polygonal Outlines of Objects from Gray Level Pictures.* IEEE Transactions on Circuits and Systems, Vol. CAS-22, No. 5, May 1975.

[Feng75−2] H. Y. F, Feng and T. Pavlidis. *Decomposition of Polygons Into Simpler Components: Feature Generation for Syntactic Pattern Recognition.* IEEE Transactions on Computers, Vol. c-24, No. 6, June 1975.

[Hesselink01] W. H. Hesselink, A. Meijster and C. Bron. *Concurrent Determination of Connected Components.* Sci. Comp. Programming, 41, pp. 173-194, 2001.

[Schneider90] P. J. Schneider. *An Algorithm for Automatically Fitting Digitized Curves.* Graphics GEMS I Edited by A. S. Glassner.

[Matas95] J. Matas, Z. Shao, and J. Kittler. *Estimation of Curvature and Tangent Direction by Median Filtered Differencing.* 8th Int. Conf. on Image Analysis and Processing, San Remo September, 1995.

[Shao96] L. Shao and H. Zhou. *Curve Fitting with Bézier Cubics.* Graphical Models and Image Processing, Vol. 58, No. 3, May, 223–232, 1996.

[Knuth86] D. E. Knuth. *Computer Modern Typefaces.* Addison Wesley Publishing Company, 1986.

[Lee12] T. C. Lee. *Reverse PostScript.* Project Report, EPFL, 2012.

[Donoho99] D. L. Donoho. *Wedgelets: Nearly Minimax Estimation of Edges.* Annals of Statistics , 27(3):859–897, 1999.

[Friedrich04] Felix Friedrich. *Complexity Penalized Segmentations in 2D.* PhD thesis, TU Munich, 2004.

[Shukla05] R. Shukla, P. L. Dragotti, M. N. Do, and M. Vetterli. *Rate-distortion optimized tree-structured compression algorithms for piecewise polynomial images.* IEEE Transactions on Image Processing, March 2005.

[Shukla04] R. Shukla. *Rate-Distortion Optimized Geometrical Image Processing.* PhD thesis, EPFL, 2004.

[Prandoni99] P. Prandoni. *Optimal Segmentation Techniques for Piecewise Stationary Signals.* PhD thesis, EPFL, 1999.

[Illustrator] Adobe Illustrator *Illustrator's Live Tracing Plugin.* Illustrator User Manual, Adobe Systems Inc.

[Smith98] E. H. B. Smith. *Characterization of Image Degradation Caused by Scanning.* Pattern Recognition Letters, Elsevier Science B.V., 1998.

[Natarajan95] B. K. Natarajan. *Filtering Random Noise from Deterministic Signals via Data Compression.* IEEE Transaction on Signal Processing, vol. 43(11), pp. 2595-2605, Nov. 1995.