ELECTRONIC PUBLISHING, VOL. 8(4), 281–294 (DECEMBER 1995)

Random fonts for the simulation of handwriting

LUC DEVROYE AND MICHAEL McDOUGALL

School of Computer Science
McGill University
Montreal, Canada H3A 2A7

SUMMARY

We present several methods for creating a random printed handwriting font based upon a small sample from a person's own hand. The first method uses random interpolation and possibly random extrapolation. The second method is based upon the random selection of points from the minimal spanning tree formed by the sample. In a third method, we look at the controlled random perturbation of Bézier control points.

KEY WORDS random font; PostScript; Bézier curves; font design

1 INTRODUCTION

Random fonts allow users to come closer to simulating true handwriting. Both PostScript type 3 fonts and metafont allow one to create veritable random fonts, in which each instance of a character is rendered differently on the screen or printer. There is little need for random fonts in ordinary texts, but we believe that there are enormous possibilities such as in private mail, personalized advertisements, automatic form letter generators, mathematics texts in which one wants to emulate blackboard mathematics, captions in Tintin and comic strips in general, restaurant menus, the generation of test samples for handwriting character recognition systems, and all applications requiring a human touch.

In most modern computer fonts, characters are described directly or indirectly by outlines, which in turn are B-splines or Bézier curves or derivatives of such curves (see Karow [1,2], Hersch [3] or André [4] in general, or Adobe's Type 1 book [5] for PostScript). For example, an ordinary Bézier curve used in the PostScript command curveto, is defined by four two-dimensional control points, z_0, z_1, z_2, z_3, and the curve is the collection of points z with

$$z = (1-t)^3 z_0 + 3t(1-t)^2 z_1 + 3t^2(1-t) z_2 + t^3 z_3 \ , \ 0 \le t \le 1$$

(see, e.g., Su and Liu [6] or Farin [7]). A character outline consists of a sequence of Bézier curves smoothly joined together at the ends. One cannot just take the collection of control points and add small random perturbations in the hope of obtaining a random character. Very quickly, the characters' appearance becomes unacceptable and does not reflect what happens when humans produce handwritten characters. Figure 1 shows examples in Utopia-Regular and Tekton of what happens to fonts when control points are moved. The outcome depends heavily on the number of Bézier sections in an outline, and is in most cases unusable. Especially the irregular stroke width becomes a problem.

CCC 0894–3982/95/040281–14$17.50 Received 22 August 1994
©1995 by John Wiley & Sons, Ltd. Revised 16 January 1995

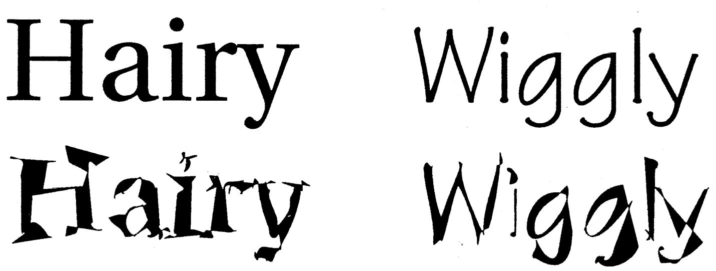

Figure 1.

One should really work with a constant pen with a deterministic (non-random) nib. In metafont, Knuth [8] introduced the idea of a pen and a nib, and he described character strokes in an analytic fashion. The outlines are constructed after a pen has been selected, thus guaranteeing pleasing constant strokes. On page 185 of Knuth [8], it is shown how one may carefully add noise to the control points so as to produce random results. Knuth's example is reproduced in Figure 2 for the sake of illustration.

The punk (metafont) font uses similar ideas to create random characters [9]. To produce a fully functional random font in this way would be interesting, but it is a painstaking job, as every character must be designed individually. Metafont may be used to create random characters but once they are created, they remain fixed: every occurrence of a symbol produces the same output. Finally, Knuth's example does not attempt to imitate one's handwriting. Sherman [10] (p. 164) shows one how a primitive, random, type 3 font might be coded. His interest was mainly at the software level and not the design of a functional font. In PostScript, it is possible to disable the font cache by using setcharwidth instead of setcachedevice and to randomize the characters before rendering. This was first pointed out by André and Borghi [11]. It was applied to the punk font by André and Ostromoukhov [12], and used for ransom fonts by Packard [13]. For other applications, we refer to the Beowolf font of van Blokland and van Rossum [14,15], or André's Scrabble font [16]. This approach is slow, as typical characters have more than 30 control points. Also, as shown above, raw independent random perturbations of the control points are not recommended as they create uneven strokes. Furthermore, at the junctures of Bézier segments, first and second order continuity are lost. Finally, as with Knuth's example, one arbitrarily extrapolates beyond a given character without regard to an individual's proneness to produce such extrapolations in their handwriting. For a survey of random (or dynamic) fonts, we refer to André [4] and André and Borghi [11].

Multiple master fonts interpolate between several fixed extreme fonts in a smooth manner. Haralambous [17] proposes certain parametrizations based upon metafont. In fact, our solution for handwriting fonts takes its cue from these ideas, but differs in that each character in the font may have a different number of prototype examples. Furthermore, we have the additional requirement that random combinations must look and behave alike.

Handwriting fonts have been available for a long time. A small list of static (non-random) PostScript fonts with sample printouts is given in Table 1. These were created from scratch through programs created by Luc Devroye and Sandro Mazzucato at McGill University in 1995.

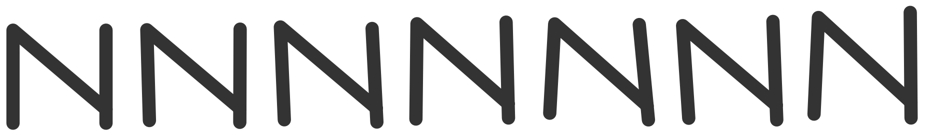

Figure 2.

There are several software houses that make customized PostScript fonts based upon one's handwriting. A program that takes scanned input and produces a type 1 PostScript font without any human intervention has been developed at our School by Sandro Mazzucato [18]. Auto-tracing algorithms are described by Plass and Stone [19], Schneider [20], Gonczarowski [21] and Itoh and Ohno [22]. A sample of a non-random handwriting font called TropDePoils produced by Mazzucato's program is shown in Figure 3.

Some might find it helpful to have a general filter that would take a type 1 font as input and construct an acceptable random type 3 PostScript font as output. This could be done for example by using `ghostscript` and playing with the PostScript operator pathforall, which allows one to play back paths and obtain the control points. Unfortunately, such a filter would have to extrapolate beyond a sample without having sufficient information regarding the person's real handwriting. What we propose here is a methodology for making random fonts. In a nutshell, we consider each sample character as a vector in Euclidean space. For a given character, we have thus a sample of n vectors independently drawn from an unknown distribution — each person has a different distribution. The objective is to draw further random samples (the characters to be rendered) from that unknown distribution. At the end of the paper, a few technical details and some font samples are provided.

2 METHODOLOGY

For each character in the font encoding vector, we need a few samples from the individual's hand. In each character, certain key points are marked. For example, in the letter A, we have 8 points as shown in Figure 4.

Typically, we mark the extreme points where strokes end, and the points where the

Garmugia	3.49	Buccellato di Lucca	6.49
Minestra di farro	3.49	Torta co'bischeri	6.99
Acquacotta Maremmana	4.39	Ciliege al vino rosso	7.99
Zuppa di fagioli di Montalcino	3.99	Crema zabaione al vinsanto	11.99
Penne alla Toscana	6.49	Meringato fiorentino	8.99
Grandinina o orzo coi piselli	7.99	Crostata di uva	7.49
Pasta alle olive	7.49	Brutti ma buoni	2.99
Pezze della nonna	7.99	Necci	3.99
Pappardelle ai pepperoni	7.99	Torta di maronni al cioccolato	8.99
Maccheroni stirate alla Lucchese	8.49	Bomboloni livornesi	7.49
Risotto al basilico	10.49	Zuccotto all'Alkermes	11.99

Figure 3.

Binkom	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Bunsbeek	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Bost	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Grimde	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Hoegaerden	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Houtem	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Meldert	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Oorbeek	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Oplinter	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Wommersom	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Waaiberg	ABCDEFGHIJKLMNOPQRSTUVWXYZ

Table 1.

derivative is horizontal or vertical. Furthermore, on each segment, at least one intermediate point is selected to create some curvature. For a particular character, a number of such point sets are obtained so that we have a representative collection for that person. One should attempt to have a set of characters that is correctly positioned and of about the same size. However, our method is remarkably robust to crazy outliers in samples. Each point set is considered as a vector in R^d for an appropriate dimension d, which may be different from character to character. These will be called the n representative vectors. A random point that will describe a random character is defined as a point uniformly distributed in the random polyhedron with the n representative vectors as vertices. When $n \le d+1$, this polyhedron may be projected down to a simplex of R^{n-1} (thus having n vertices). Random variate generation in a simplex is well-known. It suffices for example to generate $n-1$ independent uniformly distributed random variables on $[0, 1]$, sort them, and call the sorted sample $U_{(1)} < U_{(2)} < \cdots < U_{(n-1)}$. The spacings $U_{(1)}$, $U_{(2)} - U_{(1)}$, . . ., $1 - U_{(n-1)}$ are called $S_1, \ldots, S_n$. If $z_1, \ldots, z_n$ are the representative vectors, define the random point by

$$Z = \sum_{i=1}^{n} S_i z_i \ .$$

This point has the interesting property that it is uniformly distributed in the said simplex (see Devroye [23], Rubinstein [24], or Smith [25]). The interpretation for handwriting is equally interesting, as Z represents a character that looks like but is different from all representative characters. As Z is forced to fall in the convex polyhedron, the character that is produced cannot possibly misbehave. Our experiments, in fact, show a remarkable robustness. In Figure 5 we show four representative A's, together with a sample of random A's obtained by our method.

From a random Z, we construct a collection of connected or unconnected Bézier curves, forcing us to write a different procedure for each character in the encoding. For example, for the A, the first Bézier visits z_1, z_2 and z_3, while starting off vertically and ending horizontally. There are 9 degrees of freedom (eight Bézier coefficients and the value of t at z_2). The derivatives remove two degrees of freedom, and the point restrictions six, leaving

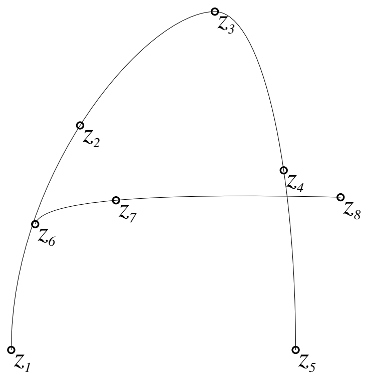

Figure 4.

one degree of freedom. Among all Bézier curves, we pick one that has approximately the smallest curvature, by performing one iteration of a second order Newton–Raphson search for a minimum. We do the same thing for z_3, z_4, z_5. Finally, an unconnected Bézier curve is added for the horizontal stroke of the character, visiting z_6, z_7 and z_8. There remain three degrees of freedom, which are removed by insisting that the Bézier curve is quadratic in t and that z_7 is visited for $t = 1/2$. The way the control points are transformed into Bézier curves is the key tool for compressing the information in the characters. For the sake of completeness, we list the six building blocks of our characters in Figure 6, omitting trivial symmetries.

A brief description of each piece follows. The point here is that we are sticking to rather simple-minded atomic components that are easy to understand and control.

(a) Horizontal to vertical curve. The function draws a curve that starts out horizontally at one endpoint, passes through a second point (at unknown t), and ends vertically at the other endpoint. These conditions leave one degree of freedom which is resolved by choosing the path with the least curvature of all the possible paths satisfying these conditions. The curvature is estimated by Simpson's rule. Optimization is performed by one iteration of the Newton–Raphson method.
(b) Horizontal to horizontal arc. The function draws a curve which begins horizontally, curves until it is vertical at the second point, and continues to curve until it is again horizontal at the third point.
(c) Horizontal to horizontal slope. The function draws a curve which begins horizontally, curves in one direction, reverses the curvature and levels out to be horizontal at the third point. This leaves two degrees of freedom. The first is resolved by making the curve pass through the second point, and the last is resolved by making the x values of the second and third control points of the Bézier curve equal.
(d) Horizontal tail. The curve begins horizontally and curves to end at the second point. The path is selected so that it is a quadratic Bézier curve, leaving one degree of freedom. This is resolved by setting the x value of the second control point of the Bézier to be exactly halfway between the x values of the endpoints.
(e) Horizontal to straight. The curve begins horizontally and curves to end at the second point. The path is selected so that it is a quadratic Bézier curve that resolves the last degree of freedom by minimizing the curvature. This curve always has the property

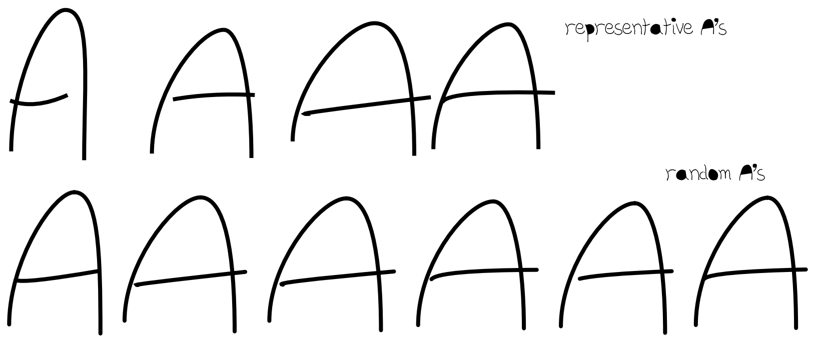

Figure 5.

that the x value of the second control point of the Bézier is 1/3 of the way from the first endpoint to the second, giving the curve a parabolic shape. This curve turns tightly at first and straightens out as it approaches the second endpoint.

(f) Simple curve. The function draws a curve between the first and third points so that the path follows a quadratic Bézier and the second point is the point on the curve where $t = 0.5$. This point is also the point which is furthest from the line segment joining the endpoints. The distance from the line segment at this point is equal to $3/4$ the distance from the second and third control points of the curve to the line joining the endpoints.

When one prints unconsciously (as when filling out a bureaucratic personal data form), very little information goes into each character. It is therefore useless to try to model the distribution of a character in a high-dimensional space. What matters more is the tightness of one's stroke. The latter problem is not dealt with in this short note, but deserves some attention.

Note that we apply the same convex combination to all components of each representative vector (vertex of the polyhedron). If we were not to do this, the characters would appear totally chaotic. For example, we would lose robustness with respect to translations in representative vectors. Take for example representatives z_1 and z_2. A convex combination is given by $z = \lambda z_1 + (1-\lambda) z_2$ with $\lambda \in (0, 1)$. If z_2 is shifted by a vector δ (as may happen when the data are not perfectly aligned), then z is shifted by $(1 - \lambda)\delta$. In other words, the shape of z is not altered! We would not be able to deduce this, had the components of z_1 and z_2 been mixed by different weights. Because of this invariance, we are virtually forced to combine vertices as we are currently doing.

3 SOME TECHNICAL DETAILS

In PostScript's type 3 fonts, users have access to virtually all regular PostScript functions, including the random number generator `rand`. Type 1 fonts are more restricted, but are now the standard because of the possibility of describing hints. Type 3 fonts remain nevertheless useful for special effects (see McGilton and Campione [26] for more details on the PostScript language). In a type 3 font, we may use the `stroke` operator to produce a stroked (as opposed to a filled) font, thus ensuring constant line widths. This is why we picked the type 3 format in our limited experiment. We are aware that variable width

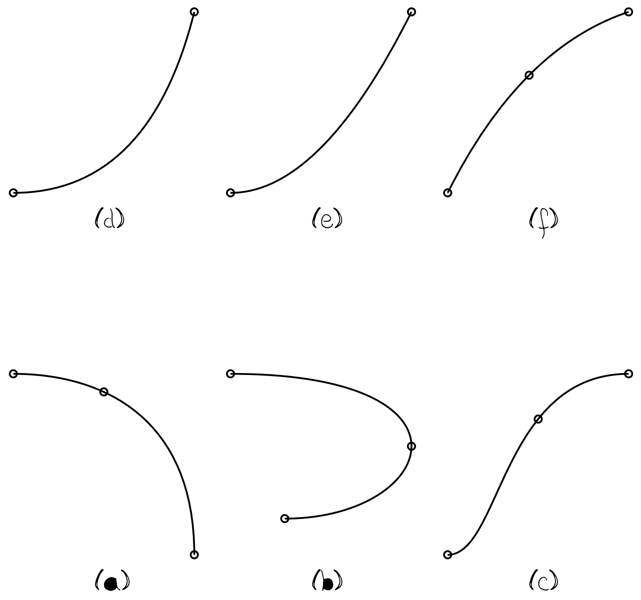

Figure 6.

strokes are preferable in many applications — metafont is ideally suited for producing such fonts through the choice of a pen nib and the operator penpos. For recent work on variable width splines, see Klassen [27].

On average, each character is described by $n = 4$ representative vectors of R^d, where d is on average about 16. For a font set with 200 characters, 12 000 numbers must be stored in a dictionary in the font. This is undoubtedly the main drawback of the present method. While the storage by itself is not exorbitant — the code occupies about 56k bytes which is not outrageous by PostScript standards — the input process is formidable and difficult to automate. We entered the data by hand!

One of the advantages of the convex combination method is that no parameters have to be picked except possibly for the thickness of the lines for stroke. This is in most cases an unbelievable luxury. Thicknesses of 0.1, 0.3, 0.8 and 1.6 points are illustrated in Figure 8 for the Tekla font.

To some extent, one would like to control the amount of randomness. This may be achieved by replacing the uniform distribution by a non-uniform one. For example, a Dirichlet process would be helpful. For definitions, see Wilks [28], Aitchison [29], Basu and Tiwari [30], Devroye [23] (chapter XIV.4), or Narayanan [31]. The spacings, barring normalization so that they sum to one, are now distributed as independent gamma (a) random variables, where $a > 0$. At $a = 1$, we obtain the uniform distribution in the polyhedron. As $a \to \infty$, we obtain the distribution that concentrates all its mass at the arithmetic center of the polyhedron (this corresponds to the least amount of randomness). As $a \downarrow 0$, we more or less pick a vertex uniformly and at random, and introduce the maximal possible variability in the outlines. Hence, a may be adjusted to achieve a given desired effect. For efficient gamma generators, we refer to Devroye [23]. A similar effect may be obtained if we use for the spacings independent vectors distributed as $E^{1/a}$, where $E = -\log U$ is exponentially distributed and U is uniform $[0, 1]$. If time is at a premium,

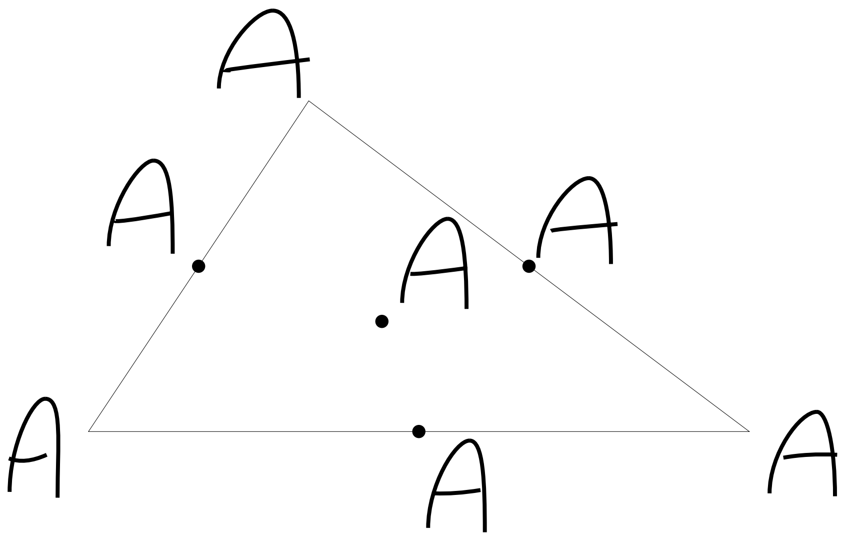

Figure 7.

one might also use $U^{1/a}$ for the spacings. Figure 9 shows how a triangle gets filled by random points for various choices of a, while Figure 10 shows characters for values of a equal to 0.1, 1 and 7.

In the input process, baselines were shown. Our program does not care whether the characters are moved or translated in the x direction, however, making the input process even smoother. The font has a dictionary for the data and for five main parameters: sidebearing, shear, thickness, exponent (the a from above), and scale (which is the δ introduced below when extrapolating beyond one's handwriting; $\delta = 1$ is the default). In setcharwidth, we report just the extremal values of the x co-ordinates of the data (after translation to line up their leftmost co-ordinates). The FontBBox is set to zero, as it is not used.

Figure 11 shows one ridiculous combination of three entirely different B's, a script B, an art nouveau B, and a typewriter B. The example is included to illustrate the robustness of our method, as well as the genetic contents of the random B's.

We may also extrapolate beyond the convex hull of the representative characters. This is achieved as follows. Let c be the center of the convex hull, obtained by averaging. Let Z be a random combination of the vertices obtained as before. Now return $c + \delta(Z - c)$, where $\delta \geq 1$ is an extrapolation parameter. For $\delta = 1$, we just return S. For $\delta > 1$, there is a positive probability of returning values outside the convex hull. In Figure 12 the first five lines have $\delta = 1$ for Tekla, while the second group of five lines corresponds to the large value of $\delta = 2.5$. Note that extrapolation creates characters that definitely have the look and feel of a stranger's hand. Some strokes will look unnatural to the person whose handwriting the font is based on.

4 RANDOM FONT VIA MINIMUM SPANNING TREES

As soon as the number of prototypes exceeds the dimension of the space by more than one, the method above is not guaranteed to produce points that are uniformly distributed inside the convex hull. A different yet exciting kind of randomization may be obtained as follows. In a preprocessing step, one first finds the minimal spanning tree of the data points (see,

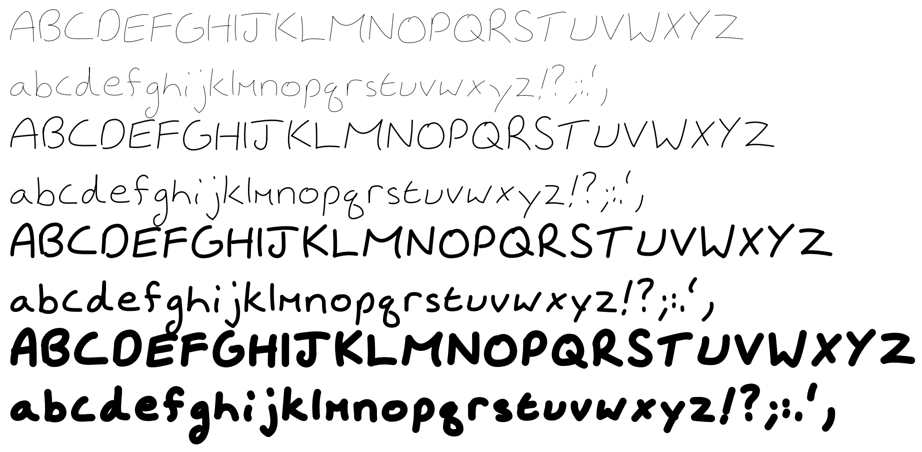
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz!?;:.',
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz!?;:.',
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz!?;:.',
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz!?;:.',

Figure 8.

e.g., Cormen, Leiserson and Rivest [32]). These trees have the desirable feature that they form the skeleton of data clouds, and thus aid in pinning down the distribution (see Figure 13).

When a character is rendered, an edge is picked at random, and a point is generated uniformly and at random on the edge. Such procedures produce characters that are closer to the original data. When $n \leq d + 1$, the edges are convex hull edges as well, and therefore, the characters should appear crisper. Figure 14 shows a minimum spanning tree suitably projected to the plane, with edge distances of the minimal spanning tree intact. The dark characters are the data, and the light characters are the random interpolations. At the right, a random sample is shown.

5 FONT SAMPLES

Figures 15 and 16 are two examples of our Tekla font, Figure 15 for running text, and Figure 16 for an Italian menu. Both are shown with character widths of 0.3 point and sidebearing of 0.35 point. The shear components are 0 and 8 respectively. The extrapolation parameter δ is 2 in the first example and 2.5 in the second one. No on-the-fly kerning was done.

6 THE KERNEL METHOD

A random sample of size n of the same character from a person's hand represented in R^d as indicated above yields a cloud with a certain structure. Above, we modeled this cloud by a family of distributions supported in the convex hull of the cloud, the uniform distribution on the convex hull being the most prominent. What is really needed is a way of generating a new point with the same (unknown) distribution as the data. This is dealt with in chapter 14 of Devroye [23] in a general statistical setting, and crucial connections with density estimation are made there. In the kernel method, for example, one generates a new point by picking a data point uniformly at random and adding a perturbation to it, say hW, where W is a random vector of R^d with a fixed density (say, the standard normal in R^d) and $h > 0$ is a given smoothing factor that needs to be adjusted in a certain way. The density

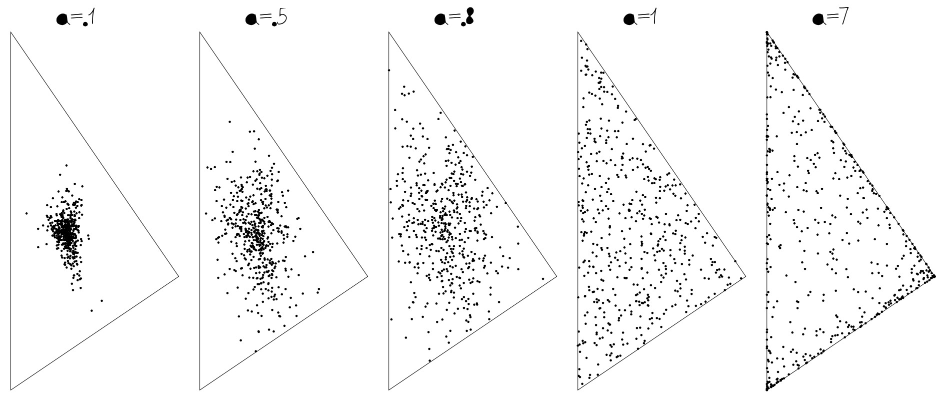

Figure 9.

of W again forces unwanted and possibly disastrous extrapolations on us, leading to ugly unnatural characters. Figure 17 shows a fixed character represented as described above, and shows random perturbations of this character with linearly increasing values of the smoothing factor h, from left to right. The first few characters are usable, but for most of the range, the noise hurts the legibility. The only possible use of the noisiest characters is in the simulation of children's first attempts at writing.

7 REMAINING PROBLEMS

We may get information from a handdrawn character with respect to the tension in the curves of one's stroke. This may then be used to add tension to the Bézier curves in the spirit of the tension parameter in metafont. In fact, we may learn a thing or two from the principles used in optical character recognition— see Suen, Berthod and Mori [33] or Zurada [34]. Recent advances in character recognition based upon the skeletonization of characters followed by neural network training have been remarkable.

On the practical side, we would like to automate the input process, which seems a phenomenal task in its own right.

Kerning random characters must be done on the fly. Hence the need for fast ways of kerning while characters are being produced. afm or tfm files become virtually useless for characters whose width changes every time. In pure PostScript, we get around this by changing the stringwidth of each character as it is generated.

Finally, random handwriting fonts in which all characters are smoothly linked create special problems that must be overcome. We have started attempts by putting information about the last character in the font itself, and using this to generate an extra linking stroke that has the correct derivatives at beginning and end. Also, the position of the link is context sensitive. This research will be reported elsewhere. A careful attempt in this direction is given in Kokula [35].

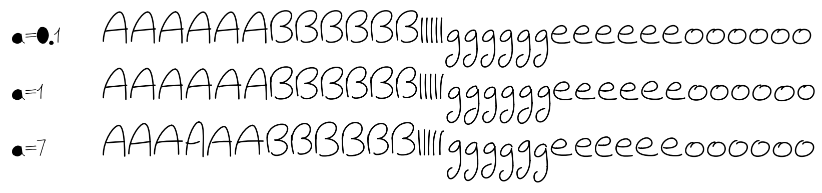

Figure 10.

ACKNOWLEDGEMENTS

Research of the authors was sponsored by NSERC Grant A3456 and by FCAR Grant 90-ER-0291. We would like to thank the three referees for their helpful remarks.

REFERENCES

1. P. Karow, *Digital Typefaces*, Springer-Verlag, Berlin, 1994.
2. P. Karow, *Font Technology*, Springer-Verlag, Berlin, 1994.
3. R. Hersch, *Technical and Visual Aspects of Fonts*, Cambridge University Press, 1993.
4. J. André, *Création de fontes et typographie numérique*, IRISA, Campus de Beaulieu, Rennes, 1993.
5. Adobe Systems Inc, *Adobe Type 1 Font Format*, Addison-Wesley, Reading, MA, 1990.
6. B.-Q. Su and D.-Y. Liu, *Computational Geometry—Curve and Surface Modeling*, Academic Press, Boston, 1989.
7. G. Farin, *Curves and Surfaces in Computer-aided Geometric Design*, Academic Press, 1990.
8. D. Knuth, *Metafont*, Addison-Wesley, Reading, MA, 1984.
9. D. Knuth, 'A punk meta-font', *TUGboat*, **9**, 152–168, (1988).
10. J. F. Sherman, *Taking Advantage of PostScript*, Wm. C. Brown Publishers, Dubuque, IA, 1992.
11. J. André and B. Borghi, 'Dynamic fonts', in *Raster Imaging and Digital Typography*, J. André and R. D. Hersch, eds, Cambridge University Press, 1989, pp. 198–204.
12. J. André and V. Ostromoukhov, 'Punk: de Metafont à PostScript', *Cahiers GUTenberg*, **4**, 23–28, (1989).
13. T. Packard, 'Ransom fonts', *The PostScript Journal*, **2**, 44–45, (1989).
14. E. van Blokland and J. van Rossum, 'Random code—the Beowolf random font', *The PostScript Journal*, **3**(1), 8–11, (1990).
15. E. van Blokland and J. van Rossum, 'Different approaches to lively outlines', in *Raster Imaging and Digital Typography II*, R. A. Morris and J. André, eds., Cambridge University Press, 1991, pp. 28–33.
16. J. André, 'The Scrabble font', *The PostScript Journal*, **3**(1), 53–55, (1990).
17. Y. Haralambous, 'Parametrization of PostScript fonts through METAFONT—an alternative to Adobe Multiple Master fonts', *Electronic Publishing—Origination, Dissemination and Design*, **6**(3), 145–157, (September 1993).
18. S. Mazzucato, *Optimization of Bézier outlines and automatic font generation*, M.sc. thesis, McGill University, Montreal, School of Computer Science, 1994.
19. M. Plass and M. Stone, 'Curve fitting with piecewise parametric cubics', *Computer Graphics*, 229–239, (1983). siggraph.
20. P. J. Schneider, 'An algorithm for automatically fitting digitized curves', in *Graphics Gems*, A. S. Glassner, ed, Academic Press, San Diego, CA, 1990, pp. 612–626.
21. J. Gonczarowski, 'A fast approach to auto-tracing (with parametric cubics)', in *Raster Imaging and Digital Typography II*, R. A. Morris and J. André, eds., Cambridge University Press, 1991, pp. 1–15.
22. K. Itoh and Y. Ohno, 'A curve fitting algorithm for character fonts', *Electronic Publishing—Origination, Dissemination and Design*, **6**(3), 195, (September 1993).
23. L. Devroye, *Non-uniform Random Variate Generation*, Springer-Verlag, New York, 1986.
24. R. Y. Rubinstein, 'Generating random vectors uniformly distributed inside and on the surface of

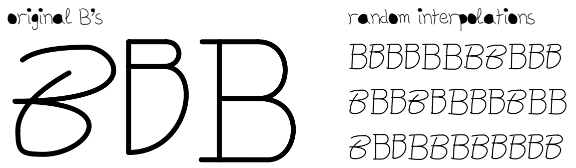

Figure 11.

different regions', *European Journal of Operations Research*, **10**, 205–209, (1982).

25. R. L. Smith, 'Efficient Monte Carlo procedures for generating points uniformly distributed over bounded regions', *Operations Research*, **32**, 1296–1308, (1984).
26. H. McGilton and M. Campione, *PostScript by Example*, Addison-Wesley, Reading, MA, 1992.
27. R. V. Klassen, 'Variable width splines: a possible font representation?', *Electronic Publishing—Origination, Dissemination and Design*, **6**(3), 183–194, (September 1993).
28. S. S. Wilks, *Mathematical Statistics*, Wiley, New York, 1962.
29. J. Aitchison, 'Inverse distributions and independent gamma-distributed products of random variables', *Biometrika*, **50**, 505–508, (1963).
30. D. Basu and R. C. Tiwari, 'A note on the Dirichlet process', in *Statistics and Probability: Essays in Honor of C.R. Rao*, G. Kallianpur, P. R. Krishnaiah and J. K. Ghosh, eds., North-Holland, 1982, pp. 89–103.
31. A. Narayanan, 'Computer generation of Dirichlet random vectors', *Journal of Statistical Computation and Simulation*, **36**, 19–30, (1990).
32. T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Boston, 1990.
33. C. Y. Suen, M. Berthod, and S. Mori, 'Automatic recognition of handprinted characters: the state of the art', in *Proceedings of the IEEE*, volume 68, pp. 469–487, (1980).
34. J. M. Zurada, *Introduction to Artificial Neural Systems*, West Publishing Company, St. Paul, 1992.
35. M. Kokula, 'Automatic generation of script font ligatures based on curve smoothness optimization', *Electronic Publishing—Origination, Dissemination and Design*, **7**(4), 217–229, (December 1994).

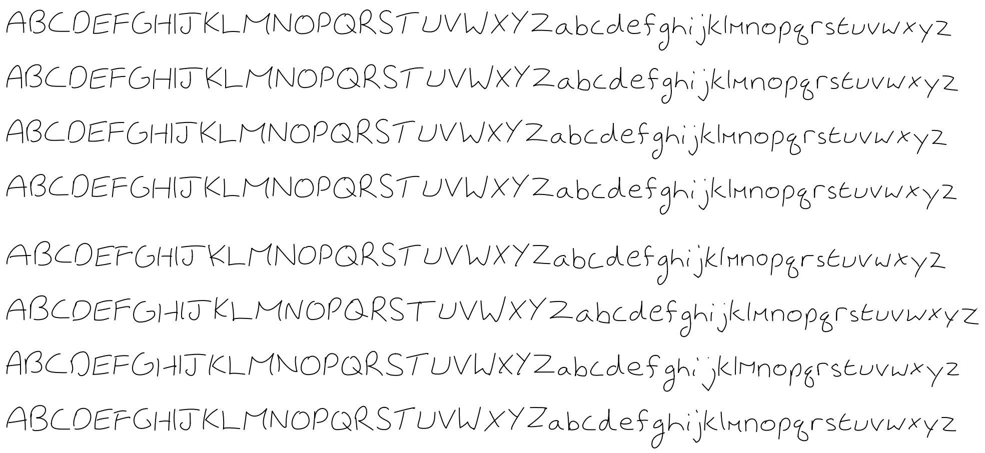
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

Figure 12.

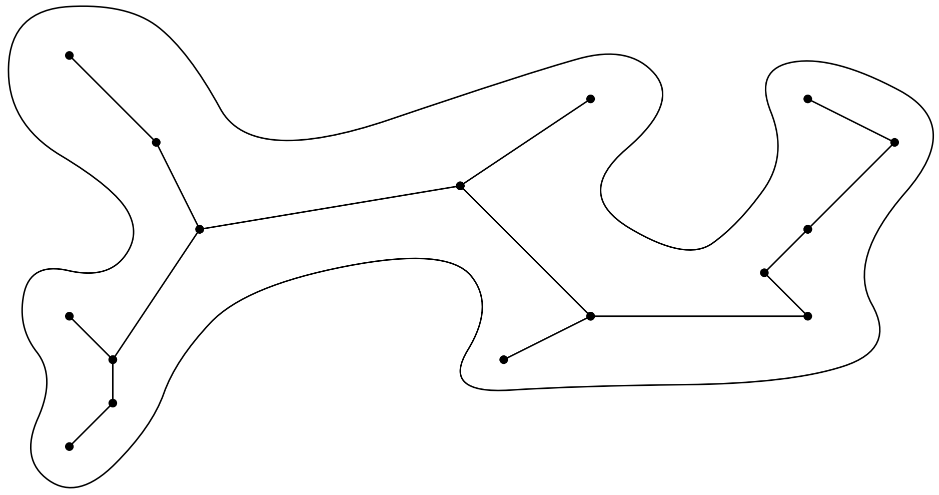

Figure 13.

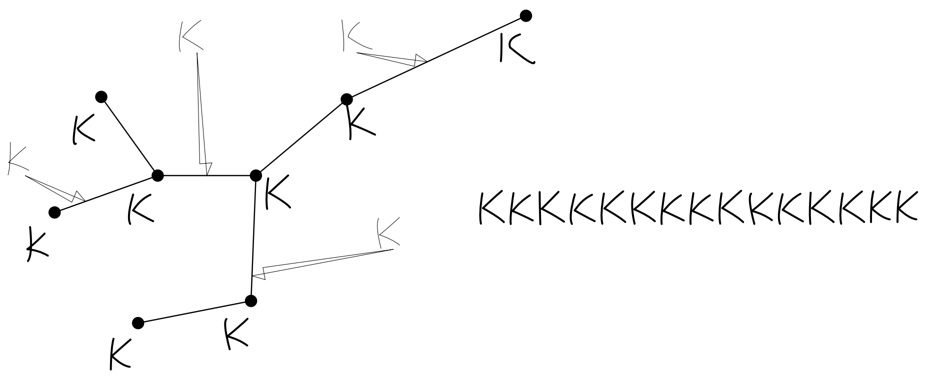

Figure 14.

Lists, such as contents pages and recipes, are opportunities to build architectural structures in which the space between the elements both separates and binds. The two favorite ways of destroying such an opportunity are setting great chasms of space that the eye cannot leap without help from the hand, and setting unlightening rows of dots (dot leaders, they are called) that force the eye to walk the width of the page like a prisoner being escorted back to its cell.

Robert Bringhurst, 1992

Figure 15.

Garmugia	3.49	Buccellato di Lucca	6.49
Minestra di farro	3.49	Torta co'bischeri	6.99
Acquacotta Maremmana	4.39	Ciliege al vino rosso	7.99
Zuppa di fagioli di Montalcino	3.99	Crema zabaione al vinsanto	11.99
Penne alla Toscana	6.49	Meringato fiorentino	8.99
Grandinina o orzo coi piselli	7.99	Crostata di uva	7.49
Pasta alle olive	7.49	Brutti ma buoni	2.99
Pezze della nonna	7.99	Necci	3.99
Pappardelle ai pepperoni	7.99	Torta di maronni al cioccolato	8.99
Maccheroni stirate alla Lucchese	8.49	Bomboloni livornesi	7.49
Risotto al basilico	10.49	Zuccotto all'Alkermes	11.99

Figure 16.

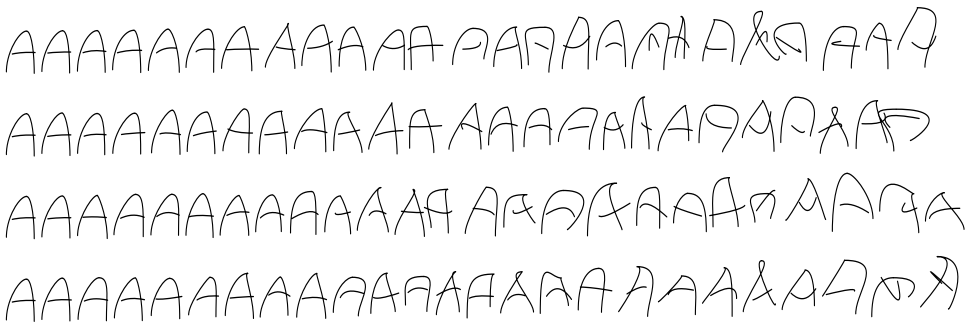

Figure 17.