[15] T. Pavlidis and C. J. Van Wyk. An automatic beautifier for drawings and illustrations. *Computer Graphics*, 19(3):225–234, July 1985.

[16] M. F. Plass and P. H. Hochschild. Optimal rendering of characters and images on discrete devices. to appear.

[17] P. van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Report 81-04, Math. Institute, Univ. of Amsterdam, 1981.

[18] C. K. Yap. An $O(n \log n)$ algorithm for the voronoi diagram of a set of simple curve segments. *Discrete and Computational Geometry*, 2:365–393, 1987.

is always room for improvement by adjusting weighting factors and adding new heuristics. Possible improvements include making better use of P-variables in single-character distortion functions and improving the treatment of approximate symmetry.

One advantage of the approach adopted here is that there is no need to write programs based on specific knowledge about character shapes. Instead of dealing with high-level concepts such as serifs, we use distortion measures to state concisely which low-level features should be preserved after the scan-conversion process. By using powerful techniques such as lattice basis reduction, we can treat competing distortion measures simultaneously to control complex features that arise from their interaction. This process is general enough that distortion measures given here are equally applicable to the Latin alphabet and Japanese Kanji.

References

[1] Adobe Systems Incorporated. Adobe type 1 font format, 1990.

[2] P. G. Apley. Automatic generation of digital typographic images from outline masters. Course note of ACM SIGGRAPH 88, Course 14: Digital Typography, 1988.

[3] L. Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[4] H. Blum and R. N. Nagel. Shape description using weighted symmetric axis features. *Pattern Recognition*, 10(3):167–180, 1978.

[5] S. Fortune. Sweepline algorithms for voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[6] M. Grötschel, L. Lovász, and A. Shrijver. Relaxation of vertex packing. *Journal of Combinatorial Theory, Series B*, 40(3):330–343, 1986.

[7] R. D. Hersch. Character generation under grid constraints. *Computer Graphics*, 21(4):243–251, 1987.

[8] J. D. Hobby. *Digitized Brush Trajectories*. PhD thesis, Dept. of Computer Science, Stanford University, 1985.

[9] J. D. Hobby. Rasterizing curved lines of constant width. *Journal of the ACM*, 36(2):209–229, April 1989.

[10] Apple Computer Inc. *Technical Introduction to the Macintosh Family*. Addison Wesley, Reading, Massachusetts, 1987.

[11] D. E. Knuth. *Computers and Typesetting*, volume D. Addison Wesley, Reading, Massachusetts, 1986.

[12] D. T. Lee. Medial axis transformation of a planar shape. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4:363–369, 1982.

[13] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[14] U. Montanari. Continuous skeletons from digital images. *Journal of the ACM*, 16(3):534–549, 1969.

where s_{ij} is the arc length associated with the jth t interval.

The distortion measures (7) in Section 3.3 are designed to give the arc-length integral of squared horizontal displacement. Doing this for a curved segment of the outline requires subdividing the curve and giving distortion measures like (33), but with (u_{ij}, v_{ij}) replaced by $(1, 0)$.

The last area where curved outlines present a problem is in the recognition of approximate symmetries. Since the algorithm in Section 3.5 is not readily adaptable to curved outlines, it should be applied to the polygonal approximations instead of the true outlines. The algorithm identifies intervals of the polygonal outlines that approximately match under a symmetry mapping $\mathcal{M}$ in the sense that $\mathcal{M}(P_1 \mathinner{..} Q_1)$ almost matches $P_2 \mathinner{..} Q_2$. It also finds a sequence of intermediate points A_1, A_2, A_3, ..., on $P_1 \mathinner{..} Q_1$ and a similar sequence B_1, B_2, B_3, ..., on $P_2 \mathinner{..} Q_2$ such that

$$\mathcal{M}(A_j) \approx B_j$$

for all j. We can then find one distortion measure for each j involving the points (X_{aj}, Y_{aj}) and (X_{bj}, Y_{bj}) on the adjusted versions of the curved outlines that correspond to points A_j and B_j on the polygonal outlines. Thus the distortion measure that replaces (16) is

$$\sqrt{\alpha_6 s_j}\,\frac{(v_j, -u_j) \cdot \big((X_{bj}, Y_{bj}) - \bar{\mathcal{M}}(X_{aj}, Y_{aj})\big)}{\sqrt{u_j^2 + v_j^2}},$$

where s_j is the arc length associated with B_j, and (u_j, v_j) is a measure of the direction tangent to the outline curve at B_j. The mapping $\bar{\mathcal{M}}$ is a version of $\mathcal{M}$ where the mapping parameter is replaced by a newly introduced variable F_k as defined by (15).

This completes out study of how to construct distortion measures for curved outlines. All of them are affine functions of the scale factor, the I, P, and F variables, and the X and Y variables that give the Bézier control points for the adjusted outlines. These distortion measures and the font-wide measures from Section 4 can be encoded and used to find low-distortion solutions for the X and Y variables as described in Section 5. These give the control points that define the desired low-distortion curved outlines.

7. Conclusions

Numerous examples in preceding sections have shown the benefits that can be achieved by using adjusted outlines to generate bitmap fonts rather than simply scaling the original outlines. In addition, the encoded forms allow most of the difficult work to be done in advance so that the decoding algorithm involves simply reading off coefficients and evaluating linear combinations of previously-computed values.

Just how rapid is the decoding process? A prototype implementation on a VAX 8550 is fast enough to generate the adjusted contours in only twice the time required to write out the coordinates to a text file. It required nine seconds to generate adjusted contours for a 121 character Times Roman font and one to three seconds to scan convert them, depending on the scale factor. This is in stark contrast to the ninety minutes of processing time required to generate the 177,000 bytes of encoded character information for this font.

The 177,000 byte space requirement is not too bad considering that a wide range of sizes can be generated from a single encoded font description, but it is ten times the 17,600 bytes for the original outlines. (A single bitmap font of 121 characters with capitals 28 pixels high requires about 6,300 bytes). Encoded outlines for the same font based on Bézier cubic curves would probably requre about 90,000 bytes. One way to reduce the space requirement still further would be to modify the encoding to make use of common subexpressions when giving the linear combinations that describe coordinates of the adjusted contours.

Another important area where there is room for improvement is in the design of the distortion function. This heuristic function is the key to the quality of the bitmap fonts that result from decoding and scan conversion. Sections 3 and 4 describe a reasonable prototype for this function, but there

instead of (X_j, Y_j). After scaling but before grid adjustment, the control points are $(\bar{\xi}_j, \bar{\eta}_j)$, where $\bar{\xi}_j$ and $\bar{\eta}_j$ are affine functions of the scale parameter σ. The new X_j and Y_j values take the place of the vertex coordinates in the V vector of Section 5, and the distortion measures are required to be linear functions of V.

Most of the distortion measures given in Section 3 can be written in terms of the Bézier control points, although some care is required to keep them linear in V. A potential stumbling block is the need to generalize the algorithms for feature recognition to work with Bézier cubic outlines. For instance, the Voronoi Diagram is needed for identifying strokes and stroke-like features, but it is impractical to construct the Voronoi for Bézier cubics due to the tremendous complexity of the required medial axis curves. (See [18] for a discussion of how to compute Voronoi Diagrams for another family of curves).

A convenient way to avoid the difficulty in finding the Voronoi Diagram is to introduce an apporoximation that provides adequate medial axis information. Since stroke-like features are not very sensitive to small changes in the outlines, we can just find polygonal approximations to the Bézier cubics and use the approximations when finding the Voronoi Diagram and identifying stroke-like features. This polygonal approximation is used only for intermediate computations, not for the final adjusted outlines.

To find a polygonal approximation to the Bézier cubic (31), divide the t interval into some number k_i of equal pieces and take a polygonal line through the points obtained by evaluating (31) at

$$t = \frac{0}{k_i}, \frac{1}{k_i}, \frac{2}{k_i}, \ldots, \frac{k_i}{k_i}.$$

The number k_i may be selected by repeatedly subdividing the t interval until the polygonal line falls within some fractional-pixel error tolerance of the desired curve. The beauty of this scheme is that it provides a natural correspondance between points on the polygonal approximation and points on the true curve. For example, the point $\frac{1}{3}$ of the way from the $t = \frac{j}{k_i}$ vertex to the $t = \frac{j+1}{k_i}$ vertex corresponds to the value of (31) at

$$t = \frac{j + \frac{1}{3}}{k_i}.$$

When the distortion measures at the end of Section 3.2 need points A and B on the true outlines corresponding to a point P on the medial axis, we can identify A and B on the polygonal approximation as described in Section 3.2 and then take the corresponding points on the true outlines. We can then use A and B in (4) to define the stroke width at P.

Now that we have the medial axis information and a way of finding points A and B on the character outlines corresponding to a point P on a medial axis, the distortion measures in Sections 3.2, 3.4, 3.6, and 3.7 make sense for Bezier cubic outlines. However, this is not the case for Section 3.1 where the distortion measures (1) require the outlines to be made up of straight line segments. The easiest way to get around this problem is to recall that the distortion measures were chosen to contribute to the distortion function a weighting factor α_1 times the integral with respect to arc length of squared perpendicular displacement. A reasonable approximation to this integral can be obtained by sampling the curve at k_i points and having one distortion measure for the perpendicular displacement of the adjusted outline at each sample point.

Let (X_{ij}, Y_{ij}) be the value of (31) at

$$t = \frac{j + \frac{1}{2}}{k_i}; \tag{32}$$

let (ξ_{ij}, η_{ij}) and $(\bar{\xi}_{ij}, \bar{\eta}_{ij})$ be the corresponding points on the original and scaled outlines; and let (u_{ij}, v_{ij}) be the derivative of (31) with respect to t evaluated at (32). Then the distortion measure for the jth t interval is

$$\frac{v_{ij}(X_{ij} - \bar{\xi}_{ij}) - u_{ij}(Y_{ij} - \bar{\eta}_{ij})}{\sqrt{u_{ij}^2 + v_{ij}^2}} \sqrt{\alpha_1 s_{ij}}, \tag{33}$$

3. For each character, find the medial axes and combine these with the integer adjustment data to find the single-character distortion measures. As each character is processed, do the QR factorization and save the font-wide distortion measures.

4. Use the cluster data to find font-wide distortion measures and combine these with the font-wide distortion measures already saved away. Then do the QR factorization and find the font-wide encoding, saving away the σ expressions for P-variables used in the encoding.

5. Scan each character again, finding the medial axes and the distortion measures and doing the QR factorization as in Step 3. As each character's R matrix is produced, refer to the σ expressions for P-variables and find the character encoding.

Once we have the font-wide encoding and all the character encodings, they can be used to generate low-distortion outlines for any scale factor σ between 1 and σ_{hi}. First, we use the font-wide encoding to find the P-variables, then we use the character encodings to generate the vertex coordinates for the low-distortion outlines.

The font-wide encoding starts with a sequence of linear combinations involving σ and previous results. The first few results are rounded to integers to give transformed P-variables and the final P-variables are given as linear combinations of these. For single-character encodings, the first few linear combinations are rounded to integers to give I-variables which then appear in subsequent linear combinations whose values give vertex coordinates for the adjusted outlines.

6. Curved Outlines

Since character outlines usually contain smooth curves, they are most naturally and concisely represented as splines. When given character outlines expressed as smooth curves, it would be nice to adjust them to fit the pixel grid without having to replace the curves by polygonal approximations. Since a polygonal outline needs many segments to approximate a curve, significant space savings generally result from representing the curves directly.

It is difficult to make direct comparisons, but the polygonal outlines like those in Figure 13 appear to have about six vertices for every Bézier cubic curve segment in commercially available outlines given in Adobe Type 1 format. [1] Since Bézier cubics require three coordinate pairs per curve segment, the number of coordinates is cut in half. In fact the overall savings would be somewhat less than this estimate since there are no savings on straight parts of a character outline.

Suppose piecewise cubic outlines do reduce the number of coordinates to encode by a factor of two. If we can generalize the distortion function and the encoding techniques of Section 5 so they apply to such outlines, we may be able to cut the size of the encoding in half as well. This would reduce the encoding space for the Times Roman example of Section 5.2 from 177,465 bytes to about 89,000 bytes. For comparison, an Adobe Type 1 font typically requires about 25,000 bytes.

How do we generalize the distortion function to work with curved outlines? The key ideas are that a curved outline is determined by a sequence of control points and any point on the outline is a linear function of the control points. For polygonal outlines, the control points are the vertices and points on the outline are obtained by interpolating between adjacent vertices. For Bézier cubic outlines, every third point is the junction between two cubic curves, and points on the ith cubic curve are given by the well-known formula

$$(1-t)^3(X_{3i},Y_{3i})+3t(1-t)^2(X_{3i+1},Y_{3i+1})+3t^2(1-t)(X_{3i+2},Y_{3i+2})+t^3(X_{3i+3},Y_{3i+3}). \tag{31}$$

For each t between zero and one, this gives a point on the curve as a linear function of four control points.

Equation (31) gives the outlines after they are adjusted to fit the pixel grid. With the naming scheme from Section 3, the original outlines are similar, but with control points of the form (ξ_j, η_j)

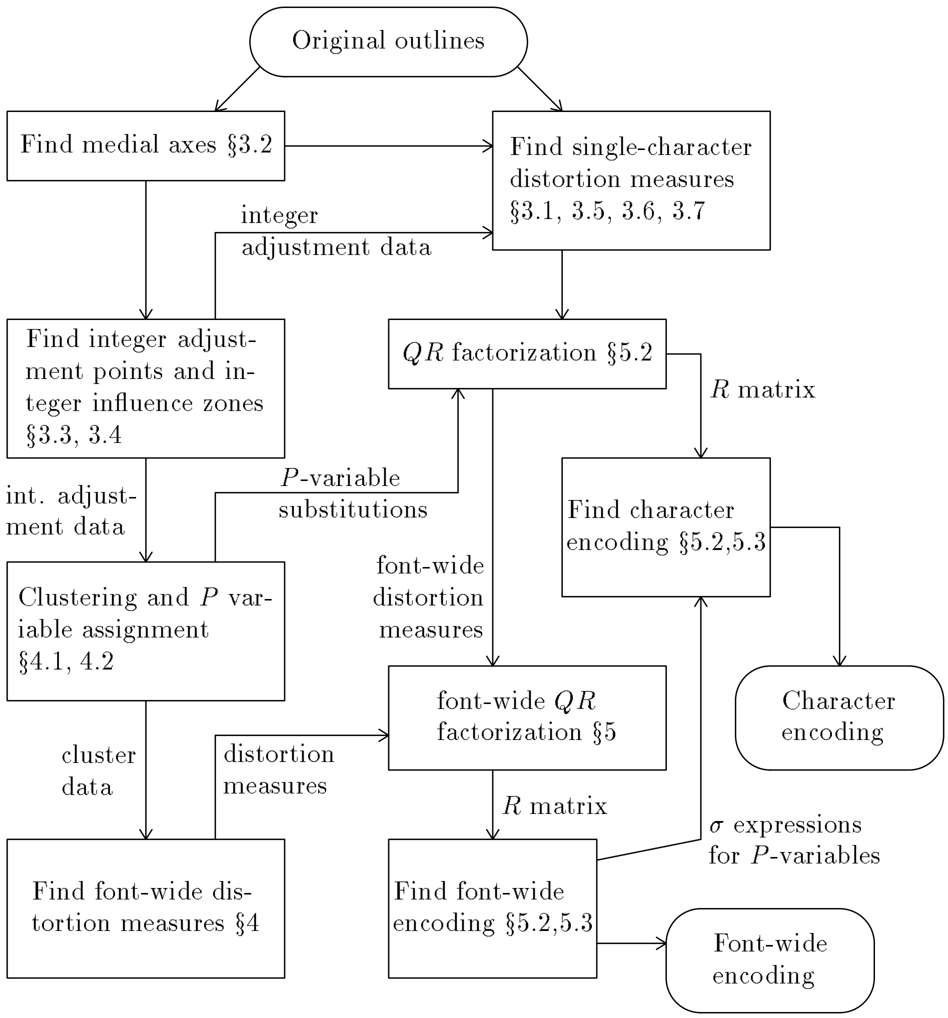


Figure 41: The flow of intermediate results when going from outlines to the encoded forms.

cannot be treated as zero unless $U_i\,|c_i|$ is less than the error bound ϵ_1, but it may be that $d\,|c_i|$ is sufficiently small while $U_i\,|c_i|$ is not. In this case (30) can be simplified to $c'_a\sigma + c'_b$, where $c'_a = c_a + ac_i$ and $c'_b = c_b + bc_i$.

Thus the strategy is to obtain an expression of the form $a\sigma + b \pm d$ for each variable used in the linear combinations to be encoded, and use the uncertainty d to decide when terms can be removed by adjusting the σ coefficients and constant terms. The simplified linear combinations can then be used in the encoded forms discussed in Sections 5.1 and 5.2, thus reducing their sizes considerably. The reduction from 217,798 nonzeros to 129,316 for the Times Roman example was achieved with the error bound $\epsilon_1 = 0.05$

The task of finding an expression of the form $a\sigma + b \pm d$ for each variable reduces to defining arithmetic operations on such σ *expressions*. Addition and scalar multiplication are easy to define:

$$(a_1\sigma + b_1 \pm d_1) + (a_2\sigma + b_2 \pm d_2) = (a_1 + a_2)\sigma + (b_1 + b_2) \pm (d_1 + d_2),$$
$$c(a\sigma + b \pm d) = (ac)\sigma + bc \pm cd.$$

The only other operation that is needed is rounding to the nearest integer. This could be implemented by just adding $\frac{1}{2}$ to the uncertainty d, but the pseudo-code in Figure 40 shows how to get better results by taking advantage of the fact that σ is known to be in the range $1 \ldots \sigma_{\mathrm{hi}}$.

$x_{\mathrm{lo}} \leftarrow a + b;$
$x_{\mathrm{hi}} \leftarrow a\sigma_{\mathrm{hi}} + b;$
$i_{\mathrm{lo}} \leftarrow \mathbf{round}(x_{\mathrm{lo}} - d);$
$i_{\mathrm{hi}} \leftarrow \mathbf{round}(x_{\mathrm{hi}} + d);$
$\mathbf{if}\ i_{\mathrm{hi}} = i_{\mathrm{lo}}\ \{\ a \leftarrow 0;\ d \leftarrow 0;\ b \leftarrow i_{\mathrm{hi}}\ \}$
$\mathbf{else}\ d \leftarrow \mathit{min}(d + \frac{1}{2},\ \mathit{max}(i_{\mathrm{hi}} - x_{\mathrm{lo}},\ x_{\mathrm{hi}} - i_{\mathrm{lo}}));$

Figure 40: The algorithm for rounding a σ expression $a\sigma + b \pm d$ to the nearest integer.

This completes the set of operations necessary to perform Steps 4 and 5 of Figure 36 or Figure 39 by using arithmetic on σ expressions. It is then a simple matter to take each linear combination and find the uncertainty d_i in the σ expression for each term of the form c_iP_i or c_iI_i. Whenever this is less than the tolerance ϵ_1, we can set $c_i \leftarrow 0$ and add a_i to the σ coefficient and b_i to the constant term.

5.4. Summary of the Encoding Process

It is now time to summarize the steps required to go from the original outlines to the encoded output. This involves a number of different types of intermediate results that are generated by one process and used by another. This is diagramed in Figure 41 with the processes shown as boxes and the flow of intermediate results indicated by arrows going from one box to another. Many of the processes are adequately identified by the names shown in the boxes, but further information can be found by referring the indicated sections.

Since each process needs its input to be computed before it can run, the arrows represent constraints on the order in which the various processes can be run. In particular, clustering and P-variable assignment cannot be done until after scanning all of the character outlines for the whole font. Since the resulting P-variable substitutions are needed before any of the character encodings can be found, there clearly have to be multiple passes over the input data. Some of the intermediate results can be saved away, but the large, dense R matrices are best recomputed. This suggests the following algorithm for the complete encoding process.

1. For each character, find the medial axes, the integer adjustment points, and the integer influence zones.
2. Use the resulting integer adjustment data for clustering and P-variable assignment.

To count the coefficients to encode for Step 5 of Figure 39, note that (28) makes each of the $2N_X$ elements of V_X linear in $\bar{V}_I$, V_P, and V_1. This gives $N_I + N_P + 2$ coefficients to encode for each element of V_X. Adding the coefficient count from Step 4 gives the total

$$(2N_X + N_I)(N_I + N_P + 2) - \frac{N_I(N_I+1)}{2}. \tag{29}$$

Once all these coefficients have been encoded, they give an intermediate form that can be used to find the low-distortion character outlines whose vertices are given by V_X. To do this, use the font-wide encoding discussed in Section 5.1 to find V_P, then use the coefficients from Step 4 of Figure 39 to evaluate the linear combinations whose rounded values give the elements of $\bar{V}_I$. The remaining coefficients give linear combinations for the elements of V_X. A small amount of additional information is then needed to determine which elements of V_X belong to which character outlines.

Consider an example taken from a set of Times Roman outlines with a cap height of 17.5 and $\sigma_{\mathrm{hi}} = 4$. (This size is appropriate for a 6-point font on a 300 dot/inch device). The "a" had $N_X = 80$ vertices and the distortion measures used $N_I = 14$ I-variables and $N_P = 13$ P-variables. Of the 4941 coefficients accounted for by (29), only 1935 turned out to be nonzero. For all 121 characters in the font, the total number of coefficients was 591,922 or which 217,798 or 37% were nonzero. Clearly, there is much to be gained by encoding only the nonzero coefficients.

This provides the motivation for the next section where we see how to adjust the coefficient vectors so that they have significantly fewer nonzero elements yet produce almost the same low-distortion character outlines. Applying these techniques in the above example reduces the number of nonzero coefficients from 217,798 to 129,316. The actual number of bytes occupied by the encoded form of these coefficients depends on the encoding scheme, but a prototype implementation managed to use only 177,465 bytes for the whole font. This also includes the auxiliary information necessary to determine which linear combinations give vertex coordinates for adjusted character outlines.

It is instructive to compare the 177,465 bytes for the encoded form of the font with what it would take to store the original character outlines. There are a total of 7804 vertices in all the character outlines, and each vertex coordinate is a nine-bit integer. Thus the total space requirement for the original outlines is about 17,600 eight-bit bytes, and the cost of storing encoded instructions for generating low-distortion character outlines is approximately a factor of ten. Hint-based schemes such as the Adobe Type 1 font format [1] are not as expensive but they are also not as flexible and they are difficult to generate automatically. Since hint-based schemes generally do not use polygonal character outlines, further discussion of their relative space efficiency is delayed until Section 6.

5.3. Increasing the Sparsity

Consider the linear combinations that go into the encoded forms described in Sections 5.1 and 5.2. If it were not for the fact that some of the encoded linear combinations are to have their results rounded to integers, all of the linear combinations could be reduced to expressions of the form $a\sigma + b$ for real numbers a and b, where σ is the scale factor. This would mean that all of the variables in the vector V could be expressed in the form $a\sigma + b$. In fact, the variables can be so expressed only approximately, but there can still be times when it is profitable to use a linear expression in σ instead of referring to a variable. The purpose of this section is to use this idea to reduce the number of nonzero terms in the linear combinations that need to be encoded. As mentioned above, the savings are significant, reducing the number of nonzero coefficients in the encoding for our Times Roman example from 217,798 to 129,316.

Suppose a linear combination contains the terms

$$c_i P_i + c_a \sigma + c_b, \tag{30}$$

where P_i is known to be in an interval bounded by $a\sigma + b \pm d$. The upper bound U_i for variable P_i is the maximum of $a\sigma + b + d$ for $1 \le \sigma \le \sigma_{\mathrm{hi}}$, where σ_{hi} is an upper bound on σ. The coefficient c_i

hence can be chosen so that

$$\begin{aligned} V_F &= -R_{FF}^{-1}(R_{FX}V_X + R_{FI}V_I + R_{FP}V_P + R_{F1}V_1) \\ V_X &= -R_{XX}^{-1}(R_{XI}V_I + R_{XP}V_P + R_{X1}V_1), \end{aligned}$$

This zeros out the first $N_F + 2N_X$ rows of the product RV leaving only the problem of finding an integer vector V_I so as to try to minimize

$$||R_{II}V_I + R_{IP}V_P + R_{I1}V_1||$$

This is essentially the same as (24) except that the matrix is R_{II} instead of R_{PP}, the variable vector is V_I instead of V_P, and the constant vector is $R_{IP}V_P + R_{I1}V_1$ instead of $R_{P1}V_1$. Thus the methods used to make (24) small can be applied here. The first step is to use Lovász basis reduction to find a transformed version $R_{II}T$ of R_{II} and substitute $T^{-1}\bar{V}_I$ for V_I. Next we use QR factorization to find an upper triangular matrix $\bar{R}$ so that $R_{II}T = \bar{Q}\bar{R}$. The remaining approximation problem can then be written $\bar{R}\bar{V}_I \approx Z$, where

$$Z = -\bar{Q}^T(R_{IP}V_P + R_{I1}V_1). \tag{27}$$

This allows $\bar{V}_I$ to be found by back substitution with rounding as shown in Figure 35. The adjusted character outlines are then obtained by evaluating

$$V_X = -R_{XX}^{-1}(R_{XI}T^{-1}\bar{V}_I + R_{XP}V_P + R_{X1}V_1), \tag{28}$$

thus concluding the distortion minimization algorithm that is summarized in Figure 39

1. Find the matrix R from the QR-factorization of A.

2. Apply the Lovász basis reduction algorithm to the subblock R_{II}, keeping track of the transformation matrix T used to create the reduced basis matrix $R_{II}T$.

3. Find the QR factorization $R_{II}T = \bar{Q}\bar{R}$ and use it to evaluate (27).

4. Use back substitution with rounding to find an integer vector $\bar{V}_I$ where $\bar{R}\bar{V}_I \approx Z$ as shown in Figure 35, except with $\bar{V}_I$ and N_I instead of $\bar{V}_P$ and $N_V - 2$.

5. Use (28) to evaluate V_X and obtain the X and Y-variables that define the adjusted contours.

Figure 39: An algorithm for finding low-distortion character outlines for the distortion function determined by the matrix A.

Since σ and the P-variables are not needed until step 4 of Figure 39, it makes sense to produce encoded instructions for completing Steps 4 and 5 once σ and the P-variables are chosen. This encoding problem is very much like the font-wide encoding discussed in Section 5.1, except that the linear combinations to be encoded involve $\bar{V}_I$, V_P, and V_1 instead of just V_P and V_1.

Since (27) makes the Z vector linear in V_P and V_1, the element $Z[k]$ in the second line of Figure 35 has to be encoded as a linear combination of V_P and V_1. Using N_I instead of $N_V - 2$ in Figure 35 gives $N_I - k$ terms in the summation for a total of $N_P + 2 + N_I - k$ coefficients in the linear combination to be encoded. Taking into account the loop in Figure 35 gives a total of

$$\sum_{k=1}^{N_I} N_P + 2 + N_I - k = N_I(N_I + N_P + 2) - \frac{N_I(N_I+1)}{2}$$

coefficients to encode in Step 4 of Figure 39.

In this case, the vector V that gives the variables and constant parameters is significantly more complicated than corresponding vector for the font-wide problem: it contains the F-variables introduced in Section 3.5 followed by the X and Y-variables that describe the adjusted outlines, the integer-valued I-variables, the P-variables that are fixed in advance to ensure font-wide uniformity, and the constant one and a scale factor σ. As shown in Figure 37, the font-wide vector V is the tail end of the vector V used in this section.

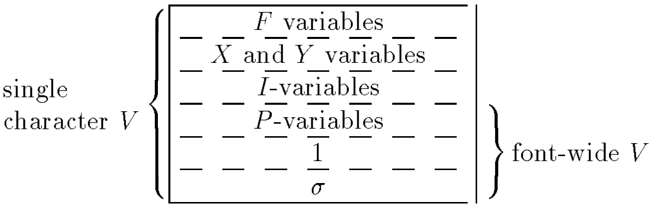


Figure 37: The structure of the vector V

Choosing V to minimize the distortion is a least-squares problem complicated by the constraint that the I-variables must have integer values. The basic approach is to find an upper-triangular matrix R such that $||AV||^2 = ||RV||^2$ for all V. As in the font-wide problem, this can be done by finding the QR factorization of A or by computing A^TA and then finding the Cholesky factorization $A^TA = R^TR$, where the latter method is faster because A is very sparse and has many more rows than columns.

Since there is no need to consider elements of the vector RV that are fixed once the scale parameter σ is known, we can restrict our attention to the first $N_F + 2N_X + N_I + N_P$ rows of R, where N_F, N_X, N_I, and N_P are the numbers of F, X, I, and P-variables respectively. Thus the remaining problem has the block structure shown in Figure 38 where V_P and V_1 are the known parts of V and V_F, V_X, and V_I are to be chosen to make $||RV||^2$ small.

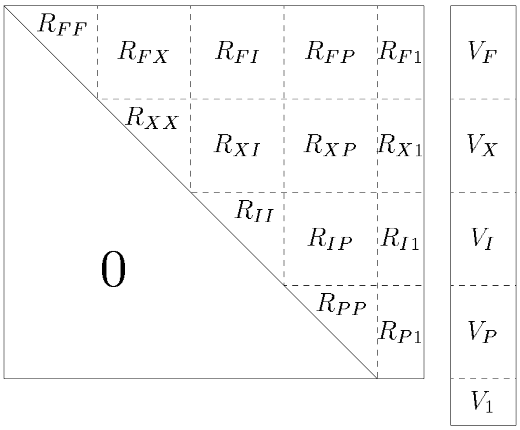


Figure 38: Block structure for the problem remaining after choosing σ and the P-variables. The vector of F-variables in V is V_F; the X and Y-variables are in V_X; V_I and V_P contain the I-variables and P-variables respectively; and $V_1 = (1\ \sigma)^T$.

Once V_P is known, the $R_{PP}V_P + R_{P1}V_1$ is fixed and the minimization of $||RV||^2$ can be based on the rest of RV. However, it is relevant to the font-wide problem that $||R_{PP}V_P + R_{P1}V_1||^2$ is a lower bound for $||RV||^2$. Adding this lower bound to the font-wide distortion function as suggested in Section 4 involves simply letting the elements of $R_{PP}V_P + R_{P1}V_1$ written as expressions in P-variables be font-wide distortion measures.

The problem of trying to minimize $||RV||^2$ is further simplified by noting that with the block structure shown in Figure 38, the variables in V_F and V_X are not constrained to be integers and

The simplest way to encode linear combinations is just by listing the coefficients in some fixed order. After investigating that, we can see about saving space by not representing zero coefficients explicitly. The goal is to get a good estimate of the space required without going into the details of exactly what encoding to use. The first step is to take a closer look at the coefficients and see exactly how many of them need to be encoded.

For the expression that gets rounded in the second line of Figure 35, there are $N_V - 2 - k$ coefficients of the form

$$\frac{\bar{R}[k,j]}{\bar{R}[k,k]}$$

plus whatever coefficents arise from

$$\frac{Z[k]}{\bar{R}[k,k]}.$$

Since (25) defines the vector Z to be linear in V_1, the element $Z[k]$ is really a function of V_1 with one coefficient for each of the two entries in V_1. This makes $N_V - k$ coefficients in the expression to be rounded in the second line of Figure 35. Summing this over the indicated values of k produces a total of

$$\sum_{k=1}^{N_V-2} N_V - k = N_V(N_V - 2) - \frac{(N_V-1)(N_V-2)}{2}$$

coefficients to be encoded in Step 4 of Figure 36.

Step 5 of the algorithm in Figure 35 can be avoided if we are willing to settle for $\bar{V}_P$ instead of V_P, but this optimization turns out to be of marginal importance. The $N_V - 2$ linear combinations needed to evaluate $T^{-1}\bar{V}_P$ have a total of $(N_V - 2)^2$ coefficients, namely the entries of T^{-1}. Thus the entire encoding requires

$$(N_V - 2)^2 + \left(N_V - \frac{N_V - 1}{2}\right) = \frac{3}{2}(N_V - 2)(N_V - 1) \tag{26}$$

coefficients.

Consider an example based on a 121-character Times Roman font with original outlines designed for 6-point at 300 dots/inch. With $\sigma_{\rm hi} = 4$, the range of sizes covered is 6 to 24 points. This font requred 13 P-variables so that $N_V = 15$. For this example, (26) says that 273 coefficients are required to encode the final steps of Figure 36. The coefficients that come from the T^{-1} matrix are small integers, and the coefficients from the expression in Figure 35 are real numbers most of which have absolute value less than or equal to one. These coefficients need to be represented to enough precision so that the resulting error in each linear combination is much less than one.

While the space required to encode 273 coefficients of modest precision is not excessive, it is instructive to consider the savings obtainable from the sparseness of the coefficient vectors. In the Times Roman example, there are 104 real-valued coefficients from Step 4 of Figure 36, and 61 of them are nonzero. A more significant saving is obtained with the T^{-1} matrix from Step 5: only 23 of the 169 coefficients are nonzero. The actual savings will not be quite as large as these statistics indicate since additional information must be encoded to indicate which coefficients are nonzero. Even so, the savings from using sparseness are significant and they will be more so when we consider how to encoding rules for finding low-distortion character outlines.

5.2. The Encoded Form of a Character

The font-wide encoding discussed in Section 5.1 can be used to find good values for the P-variables once the scale parameter is known, but this leaves the problem of finding low-distortion adjusted character outlines. This is very much like the font-wide problem in that the distortion can also be written $\|AV^2\|$, where A is a sparse, rectangular matrix that defines the distortion function.

$$\begin{aligned}&\textbf{for } k \leftarrow N_V-2, N_V-3, \ldots, 1\\ &\textbf{do } \bar{V}_P[k] \leftarrow \textbf{round}\Big(\frac{1}{\bar{R}[k,k]}\Big(Z[k] - \sum_{j=k+1}^{N_V-2} \bar{R}[k,j]\bar{V}_P[k]\Big)\Big)\end{aligned}$$

Figure 35: How to use back substitution with rounding to find an N_I-element integer vector $\bar{V}_P$ so that $\bar{R}\bar{V}_P \approx Z$.

large, but it is faster in practice to find an alternative that takes full advantage of the fact that A has only a constant number of nonzeros per row. This can be done by computing $A^T A$ and then taking the Cholesky factorization, resulting in an $O(M + N_V^3)$ time bound.[6] One of these alternatives is likely to be fast enough so that the overall running time is dominated the Lovász basis reduction in Step 2. Fortunately, this algorithm is reasonably fast for practical values of the problem size N_V even though the time bound given in [13] is worse than $O(N_V^5)$.

1. Find the matrix R from the QR-factorization of A.

2. Apply the Lovász basis reduction algorithm to the subblock R_{PP}, keeping track of the transformation matrix T used to create the reduced basis matrix $R_{PP}T$.

3. Find the QR factorization $R_{PP}T = \bar{Q}\bar{R}$ and use it to evaluate (25).

4. Use back substitution with rounding to find an integer vector $\bar{V}_P$ where $\bar{R}\bar{V}_P \approx Z$ as shown in Figure 35.

5. Use the relation $V_P = T^{-1}\bar{V}_P$ to find V_P.

Figure 36: An algorithm for finding P-variables that make the font-wide distortion small

A noteworthy feature of the distortion minimization algorithm is that the time consuming steps involving QR factorization and Lovász basis reduction are all done before the scale factor σ is needed in steps 4 and 5. Section 5.1 shows how to take advantage of this by producing an output file that contains encoded instructions for performing steps 4 and 5 once σ is known. Evaluating these encoded instructions produces a vector V_P of low-distortion P-variables that can be substituted into the single-character distortion function of Section 3.

Section 5.2 then shows how to use the transformed P-variables in the single-character distortion function and create a similar set of encoded instructions for finding low-distortion adjusted character outlines once σ is known. Both this character encoding and the font-wide version are substantially more efficient when their input is modified to increase sparsity as explained in Section 5.3. The overall process of producing both kinds of encoding are then summarized in Section 5.4.

5.1. The Encoded Form for the Font-Wide Problem

Once the the scale factor σ is known, the last two steps of the algorithm in Figure 36 can be reduced to evaluating linear combinations of known quantities and rounding some of the results to integers. Thus encoded instructions for minimizing the font-wide distortion amount to a sequence of linear combinations that need to be represented as compactly as possible. They are expressed in terms of the transformed P-variables that are elements of $\bar{V}_P$. All the linear combinations define transformed P-variables in terms of other such variables. These are the expressions that get rounded in the second line of Figure 35.

[6] This method is more prone to numerical error than QR factorization, but no difficulties were encountered in a 64-bit floating-point implementation.

We want to do as much of the work as possible in advance so that we create an intermediate form that makes it easy to find the rest of V once the scale factor σ is given.

In the case of Section 4, V contains the integer-valued P-variables followed by the constant one and a scale factor σ. Since the object is to find good values for the P-variables, it is convenient to think of V as being partitioned into a vector V_P of P-variables and a vector V_1 that contains the two constant parameters.

It is a least-squares problem to choose V_P so as to minimize (23) for a fixed value of σ. Without the integer constraints on the P-variables, it would be easy to find a solution by finding the QR decomposition $A = QR$ where Q is orthogonal and R is upper triangular. Then $||AV||^2 = ||RV||^2$ and some elements of the vector RV are zero or determined by the fixed values of σ.

Specifically, if A is M by N_V and V has N_V elements, then the last $M - N_V$ rows of R are all zeros and elements $N_V - 1$ and N_V of RV are fixed. Thus the total distortion can be minimized by considering only the first $N_V - 2$ rows of R as shown in Figure 34. With the block structure shown in the figure, the problem reduces to finding an integer vector V_P so as to minimize

$$||R_{PP}V_P + R_{P1}V_1|| \tag{24}$$

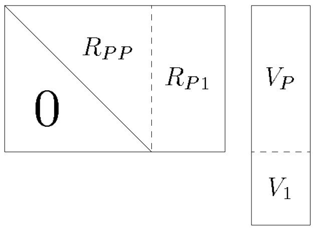


Figure 34: Block structure for the reduced problem

Van Emde Boas has shown that this problem is NP-complete [17], but good approximate solutions can often be found using the Lovász lattice basis reduction algorithm.[13] (See also Babai [3] for more details and an analysis of the approximation algorithm). When applied to R_{PP}, the algorithm finds a transformed matrix $R_{PP}T$ such that the matrices T and T^{-1} both have integer entries and the columns of $R_{PP}T$ are in some sense "more orthogonal" those of R_{PP}. (See [13] for details). The point is that a straight-forward rounding process is likely to do a better job of finding find an integer vector $\bar{V}_P$ that makes

$$(R_{PP}T)\bar{V}_P \approx -R_{P1}V_1$$

than the same process would do when choosing an integer vector V_P to make $R_{PP}V_P$ approximate the same right-hand side. Thus we can find $\bar{V}_P$ and then use the relationship $V_P = T^{-1}\bar{V}_P$ to find V_P.

This straight-forward rounding process involves choosing the entries of $\bar{V}_P$ one at a time using knowledge about previously chosen entries to compute the ideal values for the other entries. A good way to do this is first to use the QR factorization algorithm to find an orthogonal matrix $\bar{Q}$ and an upper triangular matrix $\bar{R}$ so that $R_{PP}T = \bar{Q}\bar{R}$. The remaining approximation problem is $\bar{R}\bar{V}_P \approx Z$, where

$$Z = -\bar{Q}^T R_{P1}V_1. \tag{25}$$

This be solved by via a slight variation on the standard back substitution algorithm where each element of $\bar{V}_P$ is rounded to an integer as soon as it is computed as shown in Figure 35.

Figure 36 gives the overall algorithm for finding low-distortion P-variable values given σ and the matrix A that defines the distortion measures. The running time for steps 1, 3, and 4 is dominated by the $O(MN_V^2)$ time to for the QR-factorization of A. This fine when M is not too

With this mapping, it is reasonable to subdivide (u, v) space into $\gamma_1/3$ by $\gamma_2/3$ rectangular buckets and allow any three-by-three block of such buckets to form a cluster. In other words, each integer offset interval is assigned a width and a direction angle and is placed into the appropriate bucket. The clustering process then looks at all possible three-by-three blocks of such buckets and computes the total weight of the integer offset intervals in each block. Figure 33 shows the regions in (θ, w) space assigned to typical buckets and blocks of buckets. Direction angles near $\theta = 0°$ and $\theta = 90°$ can safely be excluded because integer offsets intervals are intended only for diagonal strokes. This allows (θ, w) pairs with $90° < \theta < 180°$ to be mapped separately with $\theta_0 = 135°$.

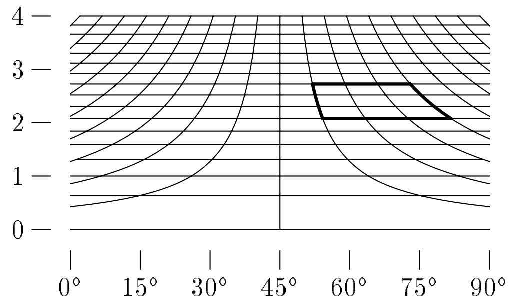


Figure 33: Square buckets of size $\frac{1}{3}$ in (u, v) space mapped into (θ, w) space by using $\gamma_1 = \gamma_2 = 1$ and $\theta_0 = 45°$ in (22). Potential width clusters for integer offset intervals are three-by-three blocks such as the one outlined in bold.

Besides creating the need for the mapping from (u, v) space, integer offsets require a few other changes to the clustering algorithm: Since integer offset intervals require two I-variables, clusters of them require two P-variables. Another change is that the termination condition needs to be modified to account for the large numbers of empty buckets and the lack of a linear ordering when buckets are laid out in a two dimensional space. This can be done by ordering the buckets according to the total weight in the blocks they belong to. This is implemented in the following integer offset clustering algorithm:

1. Scan the font to find all integer offset intervals, assigning each a weight equal to the arc length along its medial axis.

2. Find the width and direction angle for each integer offset interval and place it in the bucket corresponding to the (u, v) obtained from (22).

3. Each bucket belongs to up to nine three-by-three blocks of buckets. For each non-empty bucket B, let $\mathcal{B}(B)$ be the one of the nine blocks having the greatest total weight and call this weight $W(\mathcal{B}(B))$.

4. Find B_1 so as to maximize $W(\mathcal{B}(B_1))$. Then add up the weight ratios $W(B)/W(\mathcal{B}(B))$ for all B with $W(\mathcal{B}(B))$ at least 65% of $W(\mathcal{B}(B_1))$, and stop if the sum is at least ten.

5. Output $\mathcal{B}(B_1)$ by creating two new P-variables and using them in place of the I-variables for each integer offset interval in $\mathcal{B}(B_1)$. Then remove these integer offset intervals from their buckets and go back to step 3.

5. The Encoding

The distortion functions described in Sections 3 and 4 have been carefully chosen to be as easy as possible to minimize. Both can be described as a rectangular sparse matrix A times a column vector V that contains variables and constant parameters. The problem is to choose values for the variable entries in V so as to minimize the squared Euclidean norm

$$\|AV\|^2 . \tag{23}$$

3. Divide the points of S into two subsets S_1 and S_2 according to whether or not there is white space below them.

4. For each subset S_i and each h that is the height of a point in S_i, add the weights of the points in S_i whose height is in the interval $[h, h+\gamma_3]$.

5. Scan the height intervals for S_1 and S_2 in order of increasing height and count the number of non-overlapping intervals with weight at least γ_5 times the weight of the heaviest interval.

6. Stop if the count is more than γ_4. Otherwise output the heaviest height interval by creating a new P-variable and using it for all the integer adjustment points that went into that interval.

7. Remove from all height intervals the integer adjustment points just outputted and their partners. Then go back to step 5.

4.2. Width Clustering

Width clustering is very much like height clustering, except that it deals with features where there are integer constraints on the stroke width. Like height clustering, width clustering involves enforcing uniformity by introducing P-variables for each cluster. It differs from height clustering in that there are two types of integer-width features to consider.

The simplest type of width feature arises from integer adjustment intervals. These are created according to the rules given in Section 3.4 by finding pairs of integer adjustment points that lie on what can be called opposite sides of the same stroke. For example, Figure 30 shows four integer adjustment intervals numbered 1.1, 2.1, 4.1, and 4.2, each of which is associated with a pair of integer adjustment intervals as indicated by arrows. Intervals 1.1, 2.1, and 4.2 are associated with y coordinate integer adjustment points and have stroke width controlled by the vertical component of the separation between the associated integer adjustment points. In the case of interval 4.1 where the associated integer adjustment points control the $y-x$ component, the *width parameter* is the component of their separation in the direction $(-1,1)$.

Since horizontal and vertical stroke widths often need to be adjusted separately, it seems safe to group integer adjustment intervals according to the direction in which the width parameter is measured, then do separate clustering for each group as was done for the white-above versus white below distinction in Section 4.1. This allows the algorithm described in Section 4.1 to be used with the width parameter instead of the height. The only other change is that the weight is computed by estimating the arc length of the medial axis of the integer adjustment interval instead of finding the size of the integer influence zone. This produces an assignment of P-variables to integer adjustment intervals, at which point the P-variables are set equal to the difference between the I-variables for the appropriate pair of integer adjustment points as in (20).

Width clustering should also be done for integer offset intervals, but these are somewhat more difficult to work with because they must be grouped according to both width and direction. A further complication is the requirement that clusters must be based on width and direction ranges of non-uniform size in order to obey (10) as explained in Section 3.4.

One way to deal with these problems is to relax the constraints slightly and introduce a mapping $(\theta, w) \mapsto (u, v)$ such that a rectangle in (θ, w) space of the maximum size allowed by (10) roughly corresponds to a γ_1 by γ_2 rectangle in (u, v) space. In other words, u should change by about γ_1 when θ changes by γ_1/w for fixed w, and a change in w of $\gamma_2/\sqrt{w}$ should change v by approximately γ_2. This suggests $\frac{\partial u}{\partial \theta} = w$ and $\frac{\partial v}{\partial w} = \sqrt{\theta}$, leading to a mapping like

$$(u, v) = \left((\theta - \theta_0)w,\ \frac{2}{3}w^{3/2}\right). \tag{22}$$

It would be nice to eliminate the extraneous peaks between 16.1 and 16.7 after picking clusters for the integer adjustment points that contribute to the main peaks at 17.2 and 17.5. This amounts to looking for "partners" of the integer adjustment points that go into a new cluster, and removing the effect of the partners from the weight totals for other height intervals. In fact, it is not very hard to find these partners because the process of finding integer adjustment intervals as outlined in Section 3.4 involves pairing up such integer adjustment points. For example, each integer adjustment interval in Figure 30 is shown with arrows pointing to the two integer adjustment points that were used to create it.

After using this technique to remove the effects of partners of integer adjustment points that contribute to the six highest peaks in Figure 31, the remaining weight for each height interval is as shown in Figure 32. This makes the main peaks predominate so clearly that the rest of the data look like noise, indicating that there is probably no need to look for more clusters.

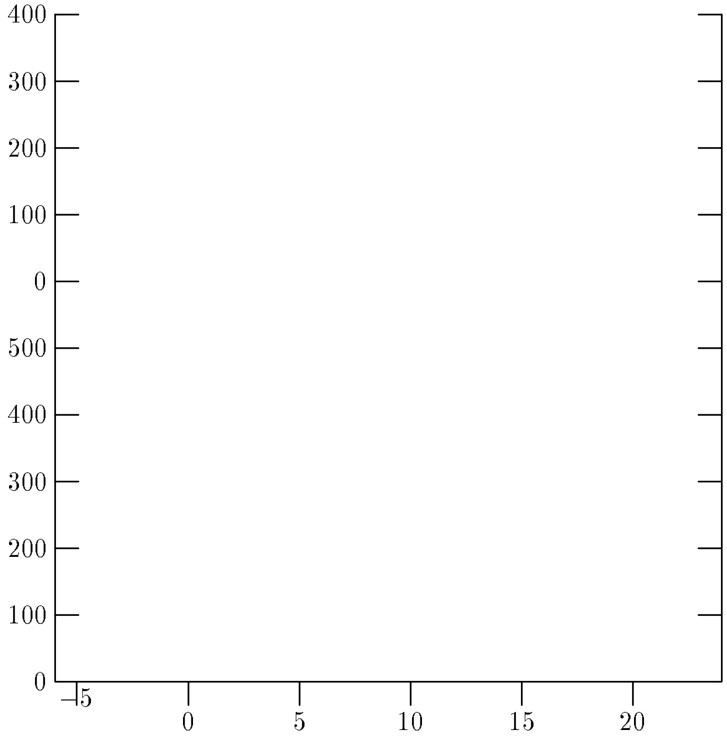


Figure 32: Two graphs of the total weight in a height interval versus the height at the center of the interval. The lower graph is based on integer adjustment points with white space below, and the upper graph uses only integer adjustment points with white space above. Each graph excludes the weight of integer adjustment points with partners that contribute to the six highest peaks.

One way to recognize this situation where there are no more clusters worth considering is to remove the main peaks chosen so far, and estimate the number of disjoint height intervals where the weight is within some fixed percentage of the weight of the heaviest height interval. For instance, a successful prototype was designed to stop looking for height clusters when more than fifteen are found or when $\gamma_4 \approx 10$ height intervals each have total weight at least $\gamma_5 = 0.65$ times the weight of the heaviest interval.

Thus the overall height clustering algorithm is as follows:

1. Let S be the set of all y coordinate integer adjustment points for which the distortion measures are given by (6); i.e., omit integer adjustment points of the types shown in Figures 22 and 23.

2. Assign each point $P \in S$ a weight equal $\psi_4(P)$.

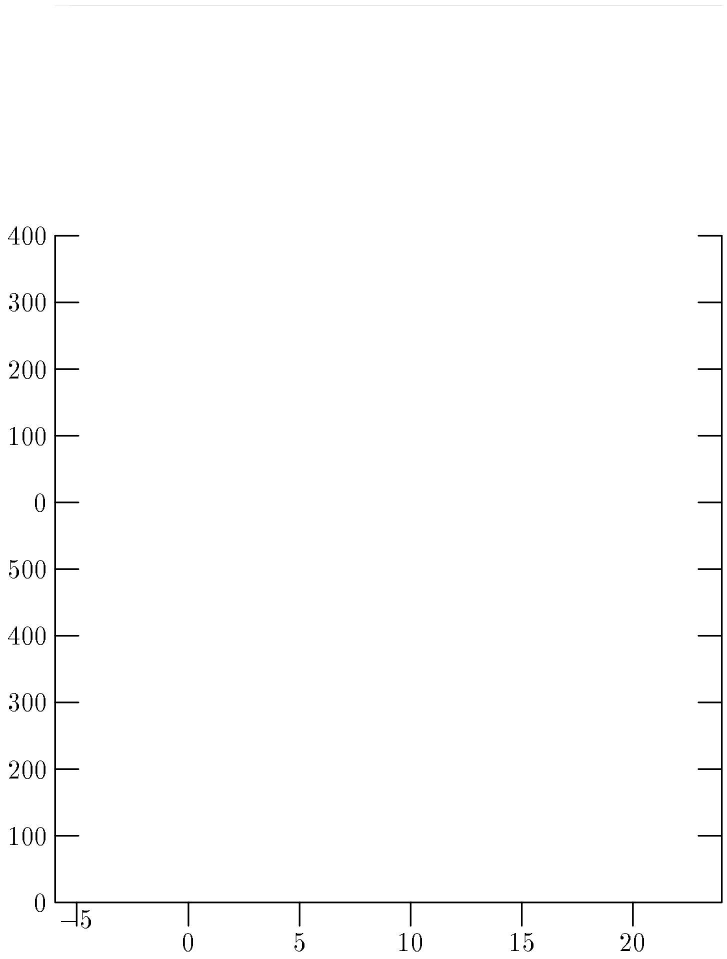


Figure 31: The lower graph is based on the weight of integer adjustment points with white space below, and the upper graph is based on points with white space above. Each shows the total weight in a height interval versus the height at the center of the interval. The data come from outlines for a Times Roman font with $\sigma_{\mathrm{hi}} = 4$.

where the weighting function ψ_3 depends on whether P_i and P_j are heights or widths. A possible weighting function for heights is

$$\psi_3(\pi_1, \pi_2) = \frac{570 A_{\text{tot}}}{H^3} \exp\left(-30\left(\frac{\pi_i - \pi_j}{H}\right)^2\right),$$

where A_{tot} is the total arc length of all medial axes in the entire font.[4] For widths, it may be better to use

$$\psi_3(\pi_1, \pi_2) = \frac{0.057 A_{\text{tot}}}{\frac{1}{4}(\pi_i + \pi_j)^2 H} \exp\left(-12\left(\frac{\pi_i - \pi_j}{\pi_i + \pi_j}\right)^2\right).$$

In spite of the complexity of the suggested weighting functions, the hard part of the process is not creating the distortion measures, but deciding what font-wide parameters are needed among all the integer-constrained heights and widths. The problems of selecting font-wide parameters and deciding which heights and widths correspond to which parameters are covered in Sections 4.1 and 4.2.

4.1. Height Clustering

As the heading implies, the selection of critical heights to control on a font-wide basis is essentially a clustering problem. If the critical heights shown in Figure 13 are to be recognized automatically without specific knowledge about letter shapes, it must be done by noting that for several character shapes, some integer adjustment points on the original outlines have almost the same y coordinates.

Suppose we scan a set of character outlines and find all the y coordinate integer adjustment points that occur at important horizontal regions. In other words, use the techniques of Section 3.3 to find all the y coordinate integer adjustment points except those of the type shown in Figures 22 and 23. Each integer adjustment point P is then assigned a weight $\psi_4(P)$ according to a heuristic function ψ_4. (A prototype implementation used the range of x coordinates covered by the integer influence zone for P plus a constant on the order of ten percent of the cap height).

The goal of the clustering process is to find small ranges of heights that maximize the total weight of all adjustment points in the height interval. This would be quite difficult if the size of height intervals were not known in advance, but the present application is compatible with a fixed value on the order of half a pixel unit. Some of the problems that can arise when the height interval size is not known in advance can be seen from the work of Pavlidis and Van Wyk. [15] The half-pixel limit for our application implies a height interval size of $\gamma_3 \approx 0.5/\sigma_{\text{hi}}$, where σ_{hi} is an upper limit on the factor σ by which the outlines are to be scaled.[5] With this setting of the interval size, the total weight as a function of height for a Times Roman font is as shown in Figure 31. The reason for showing separately the weight of integer adjustment points with white space above and white space below is to simplify the process of separating clusters.

Even with no further processing, Figure 31 shows a number of sharp peaks that could be used to pick height clusters. A closer examination reveals that many of the peaks correspond to heights marked in Figure 13 but other peaks appear to be extraneous. The apparently extraneous peaks on the upper graph between 0.5 and 1.6 are particularly noticeable. These correspond to the upper edges of strokes whose lower edge is at the baseline height or the overshoot height (corresponding to the large peaks at 0 and -0.3 on the lower graph). Similarly, the peaks between 16.1 and 16.7 in the lower graph are due to the lower edges of strokes whose upper edges are at yhe cap height or the ascender height (corresponding to the twin peaks at 17.2 and 17.5 in the upper graph).

[4]Typically $A_{\text{tot}} \approx 345H$ for fonts like Helvetica or Times Roman, but a complete set of Japanese Kanji has $A_{\text{tot}} \approx 68000H$

[5]This assumes that the original outlines are drafted carefully enough so that critical heights are within γ_3 of their intended values.

and solving them to obtain the substitutions

$$\begin{aligned} I_1 &= P_5 \\ I_4 &= I_2 + P_4 \\ I_7 &= P_1 - P_3 \\ I_{10} &= P_1 \\ I_{12} &= I_{14} + P_4. \end{aligned}$$

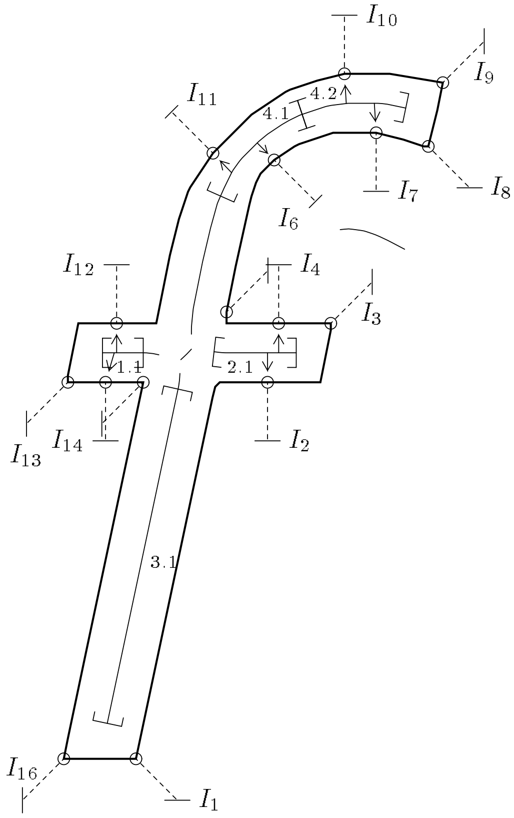


Figure 30: A character outline with medial axis lines and integer adjustment points marked as in Figure 27. Square brackets mark integer adjustment intervals and integer offset intervals, and dashed lines connect the integer adjustment points to symbols that indicate which coordinate is to be adjusted. These symbols are labeled with the corresponding I-variables.

In addition to enforcing font-wide uniformity, these substitutions can also allow the font-wide distortion function to be influenced by the ideas discussed in Section 3. In Figure 30 for example, it may be possible to get a lower bound on the single-character distortion that depends on $I_{10} - I_7 + I_2 - I_4$. Since $I_{10} - I_7 + I_2 - I_4 = P_3 - P_4$, this could lead to a font-wide distortion measure involving $P_3 - P_4$. In fact it will be shown in Section 5 that distortion measures involving only font-wide parameters are a natural consequence of reducing single-character distortion functions to a form that is easily minimized. The resulting distortion measures can only be used at the font-wide level.

The selection of other font-wide distortion measures reduces to controlling the relative values of similar height and width parameters using ideas adapted from Sections 3.6 and 3.7. In other words, if P_i and P_j represent a pair of heights or widths with ideal values π_i and π_j, there is a distortion measure

$$(P_i - P_j + \bar{\pi}_j - \bar{\pi}_i)\sqrt{\psi_3(\pi_i, \pi_j)}, \tag{21}$$

3.7. Vertical and Horizontal Alignment

It is relatively simple to avoid misalignment problems such as that shown in Figure 12 when using outlines that are adjusted to the pixel grid. All that is required is to use the information derived in Section 3.3 for determining which points on the character outline are critical for vertical and horizontal position. In the example of Figure 12, both A and B may be recognized as having important x coordinates since they are both integer adjustment points with integer-constrained x coordinates.

Although it is possible to use virtually any criteria to decide which pairs of integer adjustment points need to have their relative x or y coordinates constrained, the general idea is that two integer adjustment points need to have their relative x coordinates controlled if the following conditions hold: they have to be given integer x coordinates; and their x coordinates are only about one or two pixel units apart on the original contours. This is easily done by including a distortion measure

$$\sqrt{\psi_2(\xi_m - \xi_n)}\,(X_m - \bar{\xi}_m + \bar{\xi}_n - X_n) \tag{18}$$

for each pair of integer adjustment points (ξ_m, η_m) and (ξ_n, η_n) where x coordinates are given integer constraints. Similarly a distortion measure

$$\sqrt{\psi_2(\eta_m - \eta_n)}\,(Y_m - \bar{\eta}_m + \bar{\eta}_n - Y_n) \tag{19}$$

is used when y coordinates are given integer constraints. Note that $(\bar{\xi}_m, \bar{\eta}_m)$ and (X_m, Y_m) are the scaled and adjusted versions of (ξ_m, η_m) and ψ_2 is a weighting function something like

$$\psi_2(x) = \frac{100}{H^2} \exp\left(-\left(\frac{x}{0.03H}\right)^2\right).$$

4. The Font-Wide Distortion Function

Since some features such as baseline, x-height, and the width of main vertical stems should be uniform across the font as a whole, it is desirable to have a separate distortion function to allow the parameters to be chosen once and used for every character in the font.

An important property of the parameters that need font-wide control is that they all have integer values. For instance, the baseline, the x-height, and all the other heights marked in Figure 13 occur at integer adjustment points that are to be given integer y-values. Additionally, integer values are also needed to describe width features such as the width of vertical stems.

This suggests that the font-wide parameters can be described by integer-valued variables P_1, P_2, ..., that can be chosen by minimizing a distortion function of the form

$$T_1^2 + T_2^2 + T_3^2 + \cdots,$$

where each T_i is is a linear expression involving $P_1, P_2, P_3, \ldots$. For instance, if P_1 is the cap height and P_2 is the ascender height, there might be a font-wide distortion measure $T_{17} = P_1 - P_2 + \bar{\pi}_2 - \bar{\pi}_1$, where $\bar{\pi}_1$ and $\bar{\pi}_2$ represent the desired values of P_1 and P_2 given as affine functions of the scale factor.

Since the *P-variables* $P_1, P_2, \ldots$ represent features that are controlled by the I-variables for the characters involved, it makes sense to introduce the P-variables into the single-character distortion function by substituting combinations of P-variables for I-variables. In the case of Figure 30, this could be done by writing down the relations

$$\begin{aligned} I_{10} - I_7 &= P_3 = \text{width of curved horizontal strokes} \\ I_4 - I_2 &= P_4 = \text{width of straight horizontal strokes} \\ I_{12} - I_{14} &= P_4 = \text{width of straight horizontal strokes} \\ I_{10} &= P_1 = \text{cap height} \\ I_1 &= P_5 = \text{baseline height} \end{aligned} \tag{20}$$

at (X_{a2}, Y_{a2}) and (X_{b2}, Y_{b2}). Using $d_{ab} = \frac{1}{2}\|D\|$, the argument used to derive (1) shows that the distortion measures

$$(\Delta_a + \Delta_b)\sqrt{\frac{\alpha_6 d_{ab}}{4}} \quad \text{and} \quad (\Delta_b - \Delta_a)\sqrt{\frac{\alpha_6 d_{ab}}{12}} \tag{16}$$

contribute to the total distortion a weighting factor $\alpha_6 \approx 5 \times 10^5/H^3$ times the integral with respect to arc length of the squared perpendicular displacement.

3.6. Width Matching

Figures 2 and 14 clearly demonstrated the need for controlling relative stroke widths in scan-converted bitmaps. As suggested previously, this control can be achieved by making sure that the outlines are adjusted to fit the pixel grid. Of course this depends on the widths being controlled in the adjusted outlines, so it is a good idea to give distortion measures to penalize for deviations in relative width.

This is relatively easy to do now because almost all of the work necessary to derive distortion measures was done in Section 3.2 with the presentation of distortion measure (5) for measuring the relative error in the adjusted stroke width. The idea is that it is a lot better for the adjusted stroke width to be 10% too small all the time than for some parts of some strokes to be 10% too large while others are 10% too small. Thus it suffices to use the techniques of Section 3.2 to get expressions for the relative error in adjusted stroke width at various points and then give distortion measures that try to force the difference between any two of them to be small.

Specifically, the distortion measure (5) that controls the relative error in the adjusted stroke width at a point P on the medial axis of a stroke is just $\sqrt{\alpha_4 s_P}$ times the relative error, where α_4 is a weighting factor and s_P is a measure of the length of the stroke near P. Thus the relative error in the adjusted stroke width at P is

$$E(P) = \frac{d_{AB} \cdot (\bar{\xi}_A - \bar{\xi}_B - X_A + X_B,\, \bar{\eta}_A - \bar{\eta}_B - Y_A + Y_B)}{w_P},$$

where the notation is as used in (5). That is, $(\bar{\xi}_A, \bar{\eta}_A)$ and (X_A, Y_A) are the scaled and adjusted coordinates of $Apoint(P)$ and similar expressions with B subscripts refer to $Bpoint(P)$. Additionally, w_P is twice the distance between P and $Apoint(P)$ and d_{AB} is a unit vector in the direction of $Apoint(P) - Bpoint(P)$.

All that remains is to write a distortion measure proportional to $E(P_i) - E(P_j)$ for each pair of points P_i and P_j at which the width is to be correlated. This may be done by assuming that the medial axis of each stroke is divided into intervals where the ith interval has arc length s_i and midpoint P_i. Since each P_i has its own value for the stroke width $w(P_i)$ and the medial axis direction $d(P_i)$, there can be a weighting function ψ_1 and a distortion measure

$$\big(E(P_i) - E(P_j)\big)\sqrt{s_i s_j \psi_1(w(P_i), w(P_j), d(P_i), d(P_j))} \tag{17}$$

for each i and each j. Presumably $\psi_1(w_1, w_2, d_1, d_2)$ depends on the width difference $w_1 - w_2$ and the angle between d_1 and d_2. This function is somewhat arbitrary, but one reasonable alternative is

$$\psi_1(w_1, w_2, d_1, d_2) = \frac{500}{H^2}\exp\left(-\left(\frac{w_1 - w_2}{0.7}\right)^2 - \left(\frac{\Delta\theta(d_1, d_2)}{9}\right)^2\right),$$

where $\Delta\theta(d_1, d_2)$ is the angle between d_1 and d_2 in radians.

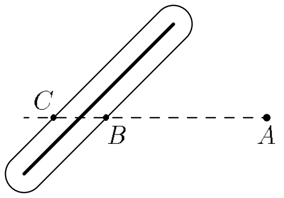


Figure 29: A segment shown as a heavy line with the outline of the neighborhood in which $\mathcal{M}(A)$ must lie in order for vertex A to match the segment under the symmetry mapping $\mathcal{M}$. Intersecting with the dashed line to give points B and C shows that the x coordinate of the symmetry axis must be between $\frac{1}{2}(A_x + C_x)$ and $\frac{1}{2}(A_x + B_x)$.

same symmetry in the adjusted outlines? Clearly, this involves comparing points on the adjusted version of $P_1 \mathinner{..} Q_1$ to matching points on the adjusted version of $P_2 \mathinner{..} Q_2$. The existing vertex-segment matches thus need to be refined by selecting for each vertex (ξ_i, η_i) on $P_1 \mathinner{..} Q_1$, the point on the matching segment closest to $\mathcal{M}(\xi_i, \eta_i)$. Naturally the same process is applied to each vertex (ξ_j, η_j) on $P_2 \mathinner{..} Q_2$, except that the point on the matching segment should be near $\mathcal{M}^{-1}(\xi_j, \eta_j)$.[3]

Once each vertex has a matching point, the problem is reduced to finding distortion measures for a segment

$$(\xi_{a1}, \eta_{a1}) \mathinner{..} (\xi_{b1}, \eta_{b1})$$

of $P_1 \mathinner{..} Q_1$ and a segment

$$(\xi_{a2}, \eta_{a2}) \mathinner{..} (\xi_{b2}, \eta_{b2})$$

of $P_2 \mathinner{..} Q_2$. It should be understood that a point such as (ξ_{a1}, η_{a1}) might lie on the interior of a segment and hence might represent a weighted average of two vertices (ξ_i, η_i) and (ξ_j, η_j). Then (X_{a1}, Y_{a1}) should be understood to refer to the same average of the corresponding adjusted vertices (X_i, Y_i) and (X_j, Y_j).

Since it is desirable to allow the adjusted contours to use a slightly different mapping parameter, the distortion measures are written in terms of a mapping $\bar{\mathcal{M}}$ that uses a newly introduced non-integer variable F_k in place of the mapping parameter. Thus

$$\bar{\mathcal{M}}(x, y) = (F_k - x,\; y) \tag{15}$$

if $\mathcal{M}$ is as given by (14). Note that because of the additive nature of the mapping parameter,

$$\bar{\mathcal{M}}(x, y) - \bar{\mathcal{M}}(0, 0) = \mathcal{M}(x, y) - \mathcal{M}(0, 0)$$

independent of F_k. The distortion measures given below refer to this function as $\mathcal{M}_v$ although for $\mathcal{M}$ as given by (14), $\mathcal{M}_v$ simply negates the x coordinate. Using $||\cdot||$ for the Euclidean norm and $A \cdot B$ for the scalar product of vectors A and B, the mean direction

$$D = \mathcal{M}_v(\xi_{b1} - \xi_{a1},\; \eta_{b1} - \eta_{a1}) + (\xi_{b2} - \xi_{a2},\; \eta_{b2} - \eta_{a2})$$

is used to define perpendicular displacements

$$\Delta_a = \frac{D \cdot \big((X_{a2}, Y_{a2}) - \bar{\mathcal{M}}(X_{a1}, Y_{a1})\big)}{||D||}$$

and

$$\Delta_b = \frac{D \cdot \big((X_{b2}, Y_{b2}) - \bar{\mathcal{M}}(X_{b1}, Y_{b1})\big)}{||D||}$$

[3]The inverse mapping $\mathcal{M}^{-1}$ is the same as $\mathcal{M}$ in the case of (14), but for shifting symmetries, the shift amount needs to be negated.

the distortion measure for perpendicularity of the integer offset vector at P is given as $1/w_P$ times the slope of the deviation from perpendicular times a weighting factor based on $\alpha_5 \approx 20/H$:

$$\sqrt{\alpha_5 s_P}\,\frac{d_{AB}\cdot(I_{k+1},-I_k)}{w_P^2}. \tag{13}$$

3.5. Enforcing Approximate Symmetry

Even when a character shape as whole is not symmetrical, it often has numerous parts that are almost symmetrical as shown in Figure 28. In the case of Figure 28a, the three legs labeled A, B, and C are each symmetrical about their own axis of symmetry and legs A and C together are almost symmetrical about symmetry axis B. In Figure 28b, there is a different kind symmetry in which the part of the outline labeled "B" is almost identical to part A except for a horizontal shift.

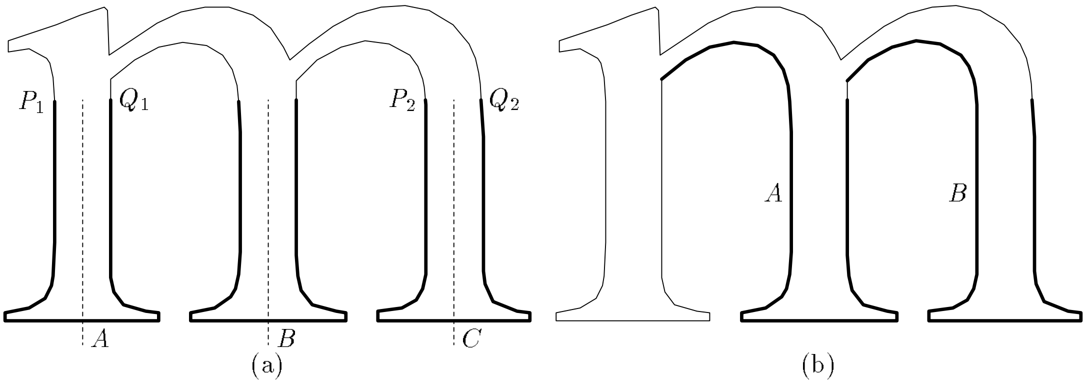


Figure 28: (a) The outline of an "m" with symmetrical parts shown in bold and axes of symmetry indicated by dashed lines. (b) The same outline with shift-symmetric parts shown in bold.

It is important that the bitmap version of the character retain the approximate symmetry, since it can be very noticeable if a character part such as a serif comes out one pixel wider on one side than the other. For this reason, it is a good idea to include distortion measures that force the adjusted character outlines to retain any approximate symmetries.

Finding approximate symmetries consists of finding pairs of intervals $P_1\,..\,Q_1$ and $P_2\,..\,Q_2$ on the original outlines such that for a suitable mapping $\mathcal{M}$, the polygonal lines $\mathcal{M}(P_1\,..\,Q_1)$ and $P_2\,..\,Q_2$ almost match. For instance when P_1, Q_1, P_2, and Q_2 are as shown in Figure 28a, it is appropriate to choose

$$\mathcal{M}(x,y) = (2x_B - x,\, y) \tag{14}$$

to reflect about the symmetry axis $x = x_B$. We should also look for symmetries about a horizontal axis, in which case $\mathcal{M}(x,y)$ is of the form $(x, 2y_B - y)$. For horizontal and vertical shifting symmetries, $\mathcal{M}(x,y)$ should be $(x_B + x, y)$ or $(x, y_B + y)$.

Since the mapping $\mathcal{M}$ is defined by a single parameter that gives the position of the symmetry axis (or the shift amount for shifting symmetries), it is possible to construct a sweepline algorithm based on this mapping parameter. Suppose for example, that the definition of when polygonal lines "almost match" is that each vertex of one line must be within a fixed distance ϵ of a segment of the other and that consecutive vertices P and Q can match different segments S and T only when the vertices between S and T all match the segment PQ. Then the sweepline algorithm only needs to keep track of which vertices match which segments for each value of the mapping parameter. This information is readily derived from the answers to a series of intersection problems like the one shown in Figure 29.

The remaining question is, given a symmetry map $\mathcal{M}$ and intervals of the character outline where $\mathcal{M}(P_1\,..\,Q_1)$ almost matches $P_2\,..\,Q_2$, what distortion measures are needed to enforce the

defined in Section 3.2 yield points in an integer influence zone. For example, any point P in the interval labeled "2.1" in Figure 27 has $Apoint(P)$ in integer influence zone A and $Bpoint(P)$ in integer influence zone B. In this case interval 2.1 is said to be the ***integer adjustment interval*** induced by integer influence zones A and B.

Of course integer adjustment intervals also apply to integer adjustment points where x or $x+y$ or $x-y$ is controlled, the main difference is that the integer influence zones are determined by limiting the change in x or $x+y$ or $x-y$. An integer adjustment interval is induced any time $Apoint(P)$ and $Bpoint(P)$ both belong to integer influence zones where P is a point on the medial axis where the direction is sufficiently close to $x=0$ or $x+y=0$ or $x-y=0$.

Scanning each medial axis line and identifying the integer adjustment intervals yields intervals 1.1, 2.1, 4.1, and 4.2 in the example shown in Figure 27. The remaining portions of each medial axis line can then be scanned for integer offset intervals by sampling the width and direction and scoring feasible integer offset intervals as explained previously. This produces the integer offset interval labeled 3.1 in the case shown in Figure 27.

The remaining task is actually generating the distortion measures that force an integer offset interval to have a width consistent with an integer offset vector. They can be similar to the distortion measures given in Section 3.2, except that the weight is higher and the adjusted stroke width is not compared to the unadjusted width, but rather to an expression involving the integer offset vector. This is done by introducing a pair of integer-valued variables I_k and I_{k+1} and comparing the actual width from the adjusted contours to the corresponding component of the vector (I_k, I_{k+1}).

In the notation of Section 3.2 a point $P=(P_x,P_y)$ on the medial axis has $Apoint(P)=(\xi_A,\eta_A)$ and $Bpoint(P)=(\xi_B,\eta_B)$, where A is a weighted average of two vertices (ξ_i,η_i) and (ξ_j,η_j), and B is a weighted average of two other vertices. The adjusted version of A, written (X_A,Y_A) is the same weighted average of the adjust points (X_i,Y_i) and (X_j,Y_j), and the adjusted version of B is defined analogously. As in Section 3.2, the width is measured in the direction

$$d_{AB}=\frac{(\xi_A-\xi_B,\eta_A-\eta_B)}{\sqrt{(\xi_A-\xi_B)^2+(\eta_A-\eta_B)^2}}.$$

Using $\cdot$ for the vector dot product, the distortion measure for controlling the width at P is

$$\sqrt{\beta_4 s_P}\; d_{AB}\cdot(I_k+X_A-X_B,\ I_{k+1}+Y_A-Y_B), \tag{11}$$

where $\beta_4\approx 2\times 10^8/H^3$ is an adjustable weighting parameter and s_P is a measure of the arc length on the medial axis near P.

The large value of β_4 is needed to make sure the width limitation imposed by the integer offset vector is accurately obeyed. This is somewhat dangerous, however, since the component in direction d_{AB} is only an approximation to the width of the adjusted contour. To ensure the accuracy of the approximation, another distortion measure

$$\sqrt{\beta_5 s_P}\; d_{AB}\cdot(Y_B-Y_A,\ X_A-X_B) \tag{12}$$

with a somewhat smaller weight parameter $\beta_5\approx 2\times 10^6/H^3$ penalizes changes in the direction from (X_A,Y_A) to (X_B,Y_B).

With distortion measures (11) and (12) for various points P along the integer offset interval, there is a large contribution to the total distortion proportional to the mean squared deviation of the width from that given by (I_k,I_{k+1}). It is then the smaller distortion measures given by (5) in Section 3.2 that come into play if the width given by (I_k,I_{k+1}) deviates from the desired value. But since these smaller distortion measures control only the component of (I_k,I_{k+1}) perpendicular to the stroke direction, other distortion measures are needed to control the amount by which the integer offset vector deviates from perpendicular to the stroke.

To balance the width control measures (5) that are given relative to the stroke width

$$w_P=2\sqrt{(\xi_A-P_x)^2+(\eta_A-P_y)^2},$$

Whatever method is used to maximize the total score, the result for the example of Figure 24 is that an integer offset interval is found on medial axis number 3 and others would be found on medial axes 1, 2 and 4 if the scoring function were a little more liberal. This brings to light an important point not considered so far, namely why is medial axis number 2 considered at all when the techniques of Section 3.3 already force both the top and bottom edges of that stroke to have integer y coordinates? It is worthwhile to avoid such redundant integer constraints for two reasons: they make it harder to avoid conflicting systems of integer constraints; and there are time and space savings that might be significant.

Figure 27 shows the outline of an "f" with open circles marking all the integer adjustment points on the outline. The short line segments connected via dashed lines to the integer adjustment points indicate which components are being fixed relative to the pixel grid. For instance, a vertical segment indicates that the x coordinate is (heavily) penalized for being non-integer, and a horizontal segment indicates a similar penalty for the y component. Each time there is a restriction on the y component, integer restrictions are also effectively placed a on contiguous portion of the outline near the integer adjustment point. These *integer influence zones* are shown in the figure with thicker lines than the rest of the outline.

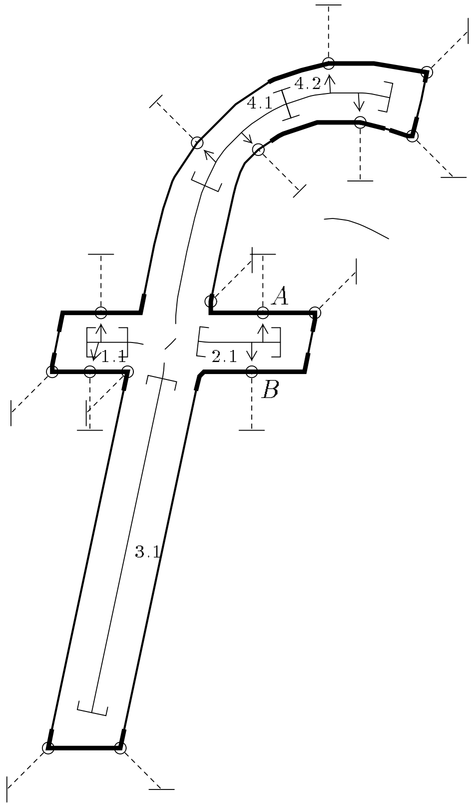


Figure 27: A character outline with the integer adjustment points marked. Integer influence zones for y coordinate integer adjustment points are shown in bold. They induce the bracketed integer adjustment intervals labeled 1.1, 2.1, and 4.2. Arrows show which integer adjustment points induce the integer adjustment intervals. The bracketed interval labeled 3.1 has no arrows because it is an integer offset interval.

When the y coordinate is controlled at an integer adjustment point A, the integer influence zone is bounded by points on the outline that differ from A in y coordinate by some fixed threshold near 0.5 pixel units as shown in Figure 27. The portions of a medial axis line where the corresponding stroke has integer-oriented width restrictions due to y coordinate integer adjustment points is simply the part where the direction is sufficiently close to horizontal and the *Apoint* and *Bpoint* functions

where $\gamma_1 \approx 0.25$ and $\gamma_2 \approx 0.25$ are adjustable parameters. An interval on a medial axis line that satisfies (10) is called a *feasible integer offset interval*

A good way to find such intervals is just to evaluate the width and direction angle at key points along each medial axis line, obtaining information such as that shown in Figure 25 for medial axis number 4 from Figure 24. The idea is to scan forward from each key point, keeping track of the ranges of width and direction angle and stopping when (10) fails. This finds all the feasible integer offset intervals of maximal length as shown in Figure 26.

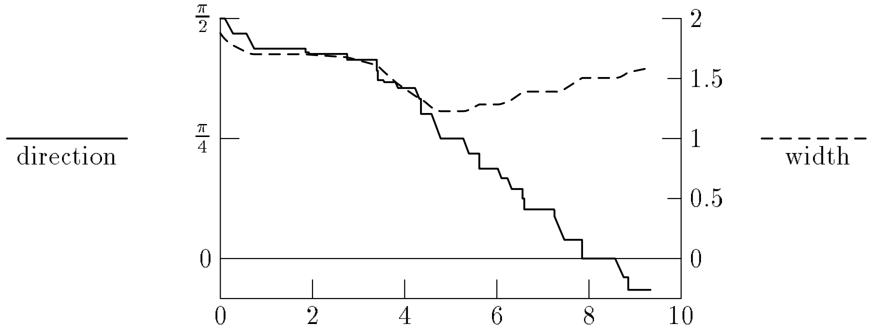


Figure 25: Width and direction angle as a function of arc length along medial axis number 4 from Figure 24.

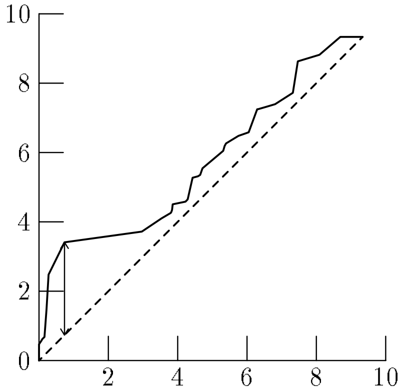


Figure 26: A graph that describes intervals of arc length along the medial axis where the width and direction angle functions shown in Figure 25 satisfy (10). The solid line shows the interval endpoint as a function of the starting point, and the dashed line shows the starting point versus itself for comparison. The longest interval occurs where the solid line attains maximal height above the dashed line as shown by the double arrow.

The graph in Figure 26 shows that the longest possible feasible integer offset interval starts 0.74 pixel units from the beginning of the medial axis line and has a length of 2.68 pixel units. Not surprisingly, this corresponds to the straight part at the lower end of stroke number 4 in Figure 24. It remains to be decided whether this is worth bothering with and whether there are any other intervals to consider.

One way to make these decisions is to choose a heuristic scoring function based on the arc length a and the range of widths $w_{\min} \ldots w_{\max}$. With a function such as

$$a - 0.75(w_{\min} + w_{\max}) - 1.0$$

that is positive when the interval is worth considering, it suffices to find the set of non-overlapping intervals that maximize the total score. This can be done easily using dynamic programming or by treating it as a problem on interval graphs and using Groetschel, Lovász and Shrijver's maximum weighted independent set algorithm.[6]

3.4. Using Integer Offset Vectors

Figures 8 and 9 in Section 2 showed that straight diagonal strokes of constant width look best when the outlines are adjusted to have integer offset vectors. Before choosing distortion measures to enforce integer offset vectors, it is necessary to decide just what constitutes a "straight diagonal stroke" and how such strokes should be found.

Fortunately the difficult problem of finding strokes was addressed Section 3.2. The techniques of that section locate the stroke-like features in terms of a set of medial axis lines and a function that gives the width of the stroke at each point on a medial axis line. For example, the "f" shown in Figure 24 has six medial axis lines including one very short one (number 5) and one that lies in the character's white space (number 6).

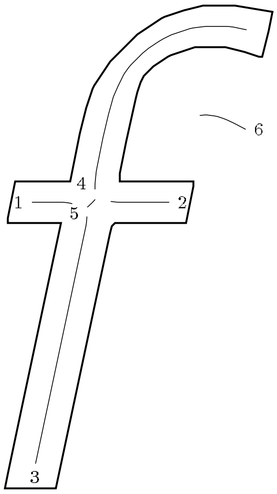


Figure 24: The outline of an "f" showing the medial axis lines that can be used to decide where integer offset vectors are needed.

What properties must the medial axis line and the width function have in order for an integer offset vector to be needed? Roughly speaking, the situation must be similar to that shown in Figure 8. That is, there must be some interval where the medial axis is fairly straight and the width is nearly constant. Another way to look at it is that the choice of the integer offset vector depends on the width and direction, and the variation in these should be small enough to ensure that a single integer offset vector can be used.

Suppose an interval along the medial axis has width ranging from $w_{\min}$ to $w_{\max}$ and direction angles between $\theta_{\min}$ and $\theta_{\max}$ when measured in radians. This means that the integer offset vector should have direction angle near $\theta_{\min}+\frac{\pi}{2}$ and $\theta_{\max}+\frac{\pi}{2}$ and length near $w_{\min}$ and $w_{\max}$. It is difficult to know precisely how big the ranges of width and direction can be until integer offset vectors are chosen, so we must err on the side of caution. Near a direction angle that corresponds to a simple rational slope, integer offset vectors of similar length can differ in direction by as little as little as $1/w_{\max}$ so it is not safe to allow $\theta_{\max}-\theta_{\min}$ to exceed this magnitude.

On the other hand, it is hard to decide in advance whether the direction angle is sufficiently close to a sufficiently simple rational slope. Without such special dirction angles, [8] shows that reasonable choices of integer offset vectors can easily deviate from perpendicular to the stroke direction by on the order of $1/\sqrt{w_{\max}}$ even when $w_{\max}=w_{\min}$. Allowing integer offset vector to be this far from perpendicular to the medial axis produces about $2\sqrt{w_{\max}}$ integer offset vectors per unit change in width. Thus a single integer offset vector can be safely used when

$$\theta_{\max}-\theta_{\min}\le\frac{\gamma_1}{w_{\max}}\quad\text{and}\quad w_{\max}-w_{\min}\le\frac{\gamma_2}{\sqrt{w_{\max}}},\tag{10}$$

best preserves the sharpness of the point when the y coordinate of the extreme point is in the middle of a row of pixels. This suggests a distortion measure

$$\sqrt{\frac{\beta_2(2.5\Delta x - \Delta y)}{2.5\Delta x}}\,(Y_m - I_k - 0.5), \tag{8}$$

where $\beta_2 \approx 5 \times 10^7/H^2$ is a weighting parameter, m is the index of the vertex where the extreme point occurs and I_k is a newly introduced integer-valued variable. Thus vertex m is also an integer adjustment point in this case.

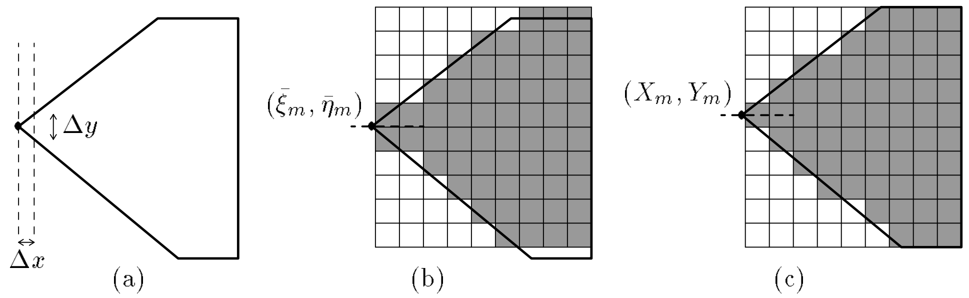


Figure 22: (a) An outline and an x extreme that should be treated as a sharp corner because of its small Δy value. (b) The outline and the scan-converted bitmap where the y coordinate indicated by the dashed line is poorly placed relative to the pixel grid. (c) The outline shifted prior to scan conversion so that the y coordinate of the dotted line is an integer plus $\frac{1}{2}$.

A somewhat more interesting case occurs in Figure 23 where the outline achieves local extrema in x and y simultaneously. In that case the scan-converted bitmap seems to preserve the sharp point best when the outline contains a pixel center near its extreme point as shown in Figure 23b. One way to derive distortion measures that ensure this is to construct a vector (a, b) that expresses the desired displacement of the extreme point from a pixel center and express the pixel center in question as $(I_k + \frac{1}{2}, I_{k+1} + \frac{1}{2})$, where I_k and I_{k+1} are two newly introduced integer-valued variables. To place an integer adjustment point at the vertex m where the extreme point occurs, include distortion measures

$$\sqrt{\beta_3}(I_k + 0.5 + a - X_m) \quad \text{and} \quad \sqrt{\beta_3}(I_{k+1} + 0.5 + b - Y_m), \tag{9}$$

where $\beta_3 \approx 10^8/H^2$ is an adjustable weighting factor. The vector (a, b) should be in the direction of the medial axis pointed outward toward vertex m. A reasonable way to select the magnitude of this vector is to make $\max(|a|, |b|) = \frac{1}{2}$ so that $a = 0.5$ and $b = 0.35$ in the case of Figure 23b.

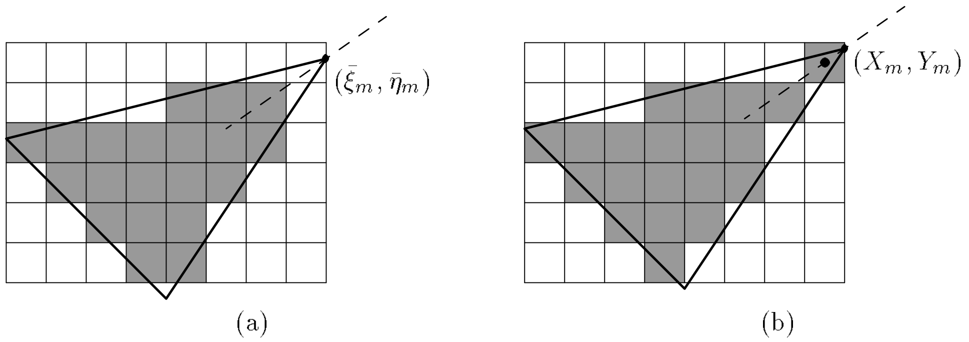


Figure 23: (a) A scan-converted outline with a local extreme point in both x and y poorly positioned on the pixel grid as indicated by the dashed line through the medial axis. (b) The same outline shifted so that the medial axis line passes through a pixel center $\frac{1}{2}$ pixel unit away.

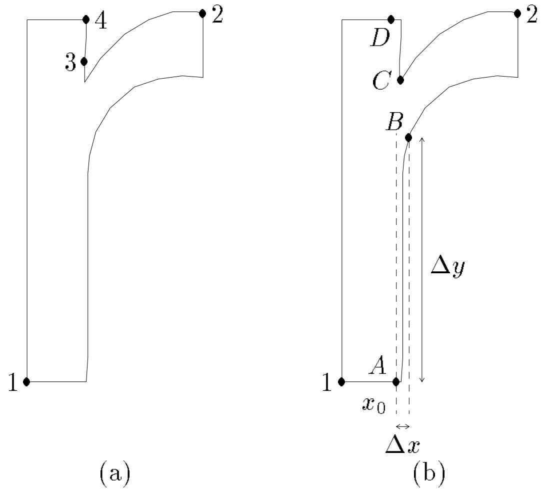


Figure 21: (a) A character outline and its x-extrema. (b) The construction for important verticals between x-extrema. The verticals occur at the intervals AB and CD and at extrema 1 and 2.

and control the position of the adjusted vertex (X_m, Y_m). This should have the desired effect on the adjusted version of M if the segment it is on does not have too much shape distortion. The resulting distortion measure is

$$\sqrt{\beta_1}\,(X_m + \bar{\xi}_M - \bar{\xi}_m - I_k), \tag{6}$$

where $\beta_1 \approx 10^8/H^2$ is an adjustable weighting parameter, I_k is a newly introduced integer-valued variable that will be chosen to minimize the absolute value of (6), and $\bar{\xi}_M$ and $\bar{\xi}_m$ are the scaled versions x coordinates for M and (ξ_m, η_m). Since this effectively imposes an integer restriction on the coordinates of the adjusted version of M, point M is called an *integer adjustment point.*

To force the rest of the region AB to be adjusted in a manner similar to M, it is a good idea to add additional distortion measures whose sum of squares is β_1 times the mean squared difference between the shift amount $X_m - \bar{\xi}_m$ and the shift amount for other points in AB. Making use of the trick used to derive (1) in Section 3.1, it suffices to add distortion measures

$$\begin{aligned} &\sqrt{\frac{\beta_1 d_{ij}}{4 d_{AB}}}\,\big(X_i + X_j - \bar{\xi}_i - \bar{\xi}_j + 2\bar{\xi}_m - 2X_m\big), \\ &\sqrt{\frac{\beta_1 d_{ij}}{12 d_{AB}}}\,\big(X_i - X_j - \bar{\xi}_i + \bar{\xi}_j\big) \end{aligned} \tag{7}$$

where i and j range over all pairs of adjacent vertex numbers between A and B, inclusive. Here d_{AB} is the arc length between A and B, and d_{ij} is the distance from (ξ_i, η_i) to (ξ_j, η_j).

Naturally, the same type of distortion measures can be generated for important horizontal regions by just using Y and $\bar{\eta}$ in place of X and $\bar{\xi}$. In fact it is possible to control the positions of 45° lines by using $X + Y$ and $\bar{\xi} + \bar{\eta}$, although in this case the line is fitted to the pixel grid if it contains a point (X_M, Y_M) with $X_M + Y_M \approx I_k + \frac{1}{2}$ for some integer I_k. (The curved portions of the letter "m" in Figure 14 have been fitted to pixel grid in this fashion).

Another case where distortion measures need not be exactly in the form of (6) or (7) is when the character outline has a local extreme in x without having much of a vertical edge. This situation can be identified by using the same test used to locate important vertical extrema: i.e., the part of the outline near the extreme point with x coordinates falling within the constant Δx of the extreme has a range of y coordinates Δy not more than about 2.5 times Δx. Performing this test with $\Delta x = 0.6$ yields 1.02 pixel units for Δy in the case shown in Figure 22a where the outline has a sharp point at the x extreme. As can be seen from Figures 22b and 22c, the scan-converted bitmap

definitions for (ξ_B, η_B) etc. With this notation, the most useful definition for the width at $P = (P_x, P_y)$ is[2]

$$w_P = 2\sqrt{(\xi_A - P_x)^2 + (\eta_A - P_y)^2}. \tag{4}$$

Since the distortion measure for width errors must be simple enough to allow for subsequent processing, we use an approximation based on a unit vector in the direction from A to B:

$$d_{AB} = \frac{(\xi_A - \xi_B, \eta_A - \eta_B)}{\sqrt{(\xi_A - \xi_B)^2 + (\eta_A - \eta_B)^2}}.$$

The distortion measure for width error as a fraction of w_P is

$$\sqrt{\alpha_4 s_P}\ \frac{d_{AB} \cdot (\bar{\xi}_A - \bar{\xi}_B - X_A + X_B,\ \bar{\eta}_A - \bar{\eta}_B - Y_A + Y_B)}{w_P}, \tag{5}$$

where $\alpha_4 \approx 50/H$ is a weighting factor, $\cdot$ refers to the vector dot product and s_P is a measure of the arc length on the medial axis near P.

The meaning of the arc length s_P is that there are a number of distortion measures (5) for various points P chosen by dividing the medial axis lines into short intervals and choosing a point midway along each interval. In this way the contribution to the total distortion is made roughly proportional to the mean squared relative width error.

3.3. Fitting Verticals and Horizontals to the Pixel Grid

In order for a set of outlines to "fit" the pixel grid, vertical and horizontal parts of the outline must fall on pixel boundaries. This also holds for curves that pass through vertical or horizontal. As shown in Figure 10 of Section 2, such curves produce better pixel outlines when the horizontal or vertical tangent line falls on pixel boundaries.

Figure 21 illustrates the process of finding vertical and horizontal parts of a character outline. To find important vertical parts, first scan the outline and mark the points where the x-coordinate achieves local minima or maxima as shown in Figure 21a. Next throw away pairs of consecutive extrema whose x coordinates differ by no more than 0.5 pixel units or some other fixed threshold. This causes extrema 3 and 4 to be omitted in Figure 21b. The remaining extrema either lie on important vertical edges or need to have their positions fixed relative to the raster for other reasons as explained below.

To find any remaining important verticals, consider each pair of consecutive extrema and look for verticals between them by finding the portion of the outline that falls in a small range of x-coordinates $x_0 \le x \le x_0 + \Delta x$. If this portion of the outline is an important vertical region, it will cover a range of y-coordinates of some large size Δy. Thus for each pair of consecutive extrema, the idea is to treat Δy as a function of x_0 and find all local maxima where Δy is greater than some threshold (at least a few pixel units) and separated from any higher maximum by at least some other threshold (typically one or two pixel units). This produces an important vertical region AB between extrema 1 and 2 and a similar region CD between extrema 2 and 1 in Figure 21b.

After using an analogous process to find important horizontal regions, the next step is to find distortion measures that force the vertical and horizontal regions to fit the pixel grid. If the grid lines that divide pixel squares have integer x and y coordinates, there should be distortion measures that get large when the x and y coordinates of important vertical and horizontal regions get far from an integer.

Consider the vertical region AB and let M be a point on AB whose y coordinate is midway between A and B. The point corresponding to M on the adjusted outlines is a weighted average of the endpoints of the segment it lies on, but it is safer to choose one of these endpoints (ξ_m, η_m)

[2] We could just take the distance between points A and B but that is not always a continuous function of P.

After throwing away parts of the Voronoi Diagram where the opposition angle is less than the primary threshold, the resulting set of medial axis lines can be pruned further by looking at the maximum value of the opposition angle on each axis line. For instance, the opposition angle ranges from 146° to 180° along the length of axis 6 while it is constant at 151° along axis 7. This causes axis 7 to be thrown away when the secondary threshold is set to require a maximum opposition angle of at least 157° as shown in Figure 19b. Setting the threshold at 166° causes three other extraneous axis lines to be dropped (Figure 19c).

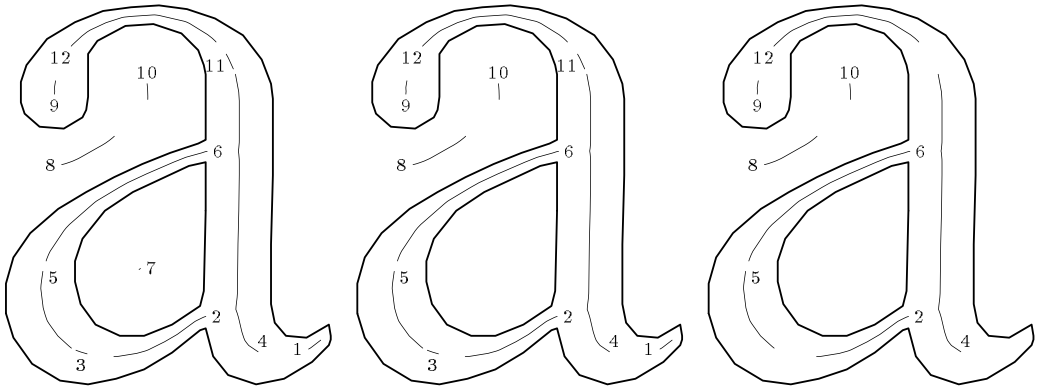


Figure 19: Polygonal outlines for an "a" with parts of its Voronoi Diagram selected by various pruning methods. (a) shows single-stage pruning at 146°, and (b) and (c) show two-stage pruning with the secondary threshold at 157° and 166° respectively.

Recall that the purpose of the dual threshold process is to produce the information necessary to create distortion measures for controlling the width of stroke-like features. In addition to the medial axis lines shown in Figure 19c, this requires knowledge of the points A and B used to compute the opposition angle for each point P on the medial axis lines. (See Figure 17a). Fortunately, this information is a byproduct of many Voronoi Diagram construction algorithms. The Voronoi Diagram is constructed as a set of line segments and parabolic arcs, each of which is "midway between" two segments, two vertices, or a vertex and a segment. In each case, the points A and B are easily constructed as shown in Figure 20 once the two segments or vertices are known. Thus given a point P on a line segment or parabolic arc of the Voronoi Diagram, functions $Apoint(P)$ and $Bpoint(P)$ are defined to lie on the appropriate vertex or segment from the character outline with the understanding that the point to select on a segment is the one closest to P.

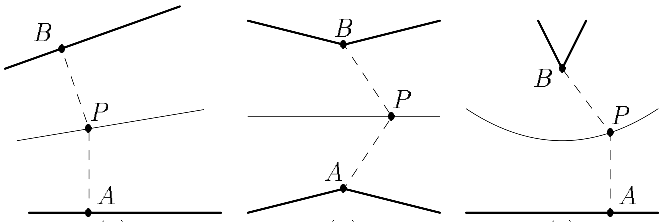


Figure 20: (a) How to find A and B when P is on the line midway between two segments (b) The same computation when P is on the line midway between two vertices. (c) The same for P on the parabolic arc generated by a point and a segment.

If A is t_A of the way from (ξ_i, η_i) to (ξ_j, η_j), it is convenient to let

$$(\xi_A, \eta_A) = \big(\xi_i + t_A(\xi_j - \xi_i),\ \eta_i + t_A(\eta_j - \eta_i)\big)$$

and define $(\bar{\xi}_A, \bar{\eta}_A)$ and (X_A, Y_A) analogously to be the scaled and adjusted versions of point A. Similarly, point B will lie some fraction t_B of the way along some other edge, and this leads to

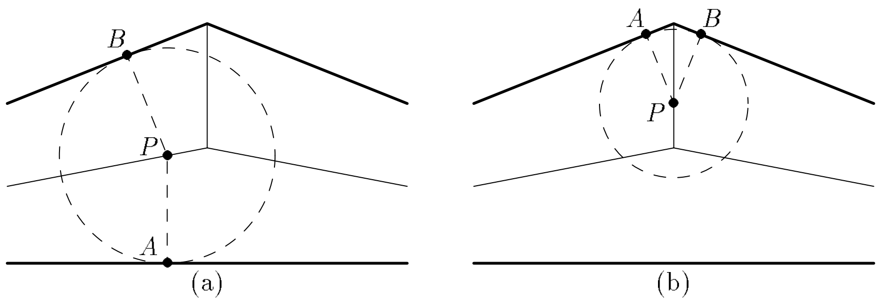


Figure 17: Thick lines give portions of polygonal outlines, thin lines give corresponding parts of the medial axis tree, and the dashed circles are as defined by the radius function at the points P. The opposition angle APB is closer to $180°$ in (a) than in (b).

If P is to fit the intuitive notion of a stroke, A and B should be on opposite sides of the stroke and the opposition angle APB should be close to $180°$. This suggests scanning the medial axis structures and throwing away parts where the opposition angle is below some threshold. Figure 18 shows the effect of this pruning process for two different thresholds. Figure 18a also introduces an extended version of the medial axis structures that cover the exterior as well as the interior. These extended structures, commonly known as Voronoi Diagrams allow the thresholding process to find stroke-like features in the white space. (Voronoi edges that arise from a segment and one of its endpoints are considered to have opposition angle $0°$ and are therefore removed by the thresholding process).

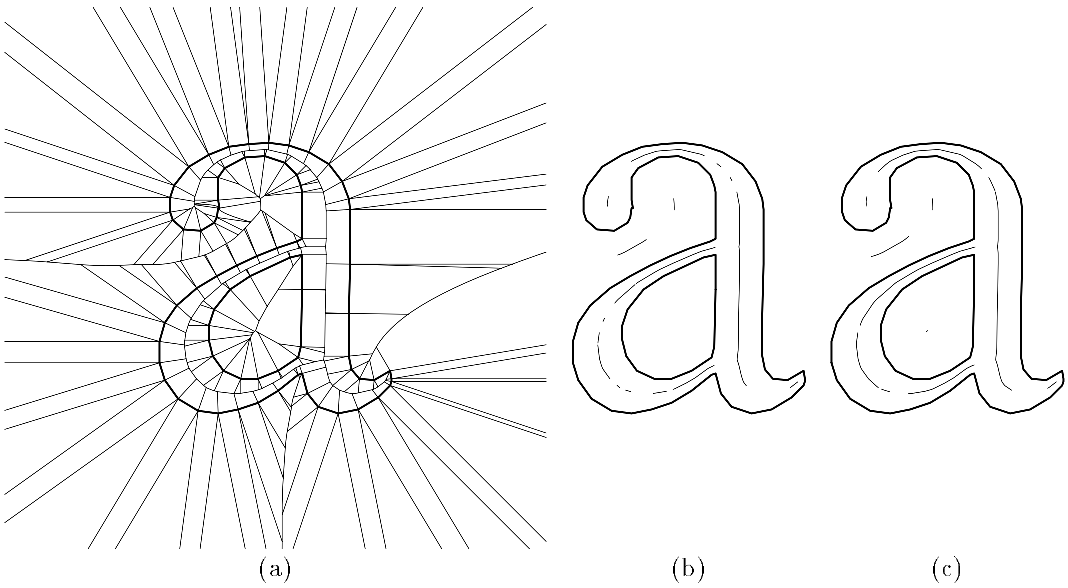


Figure 18: (a) The polygonal outline of an "a" shown in bold with its Voronoi Diagram drawn with thinner lines. (b) The result of throwing away parts of the Voronoi Diagram where the opposition angle is less than $157°$. (c) The same diagram with the threshold set at $146°$.

Figure 19a illustrates more clearly some of the interesting stroke-like features found in the white space by applying an opposition angle threshold to the Voronoi Diagram. The medial axis lines labeled "8" and "10" fall into this category, but the short line labeled "7" is not so interesting. Artifacts such as axis 7 tend to arise when the threshold angle is set high enough to prevent fragmentation of the medial axis lines in cases like that shown in Figure 18b. Montanari does not deal with this problem, but it can be avoided by extending his thresholding process to include a secondary threshold as explained below.

and the distortion measure for bending is

$$\sqrt{\alpha_3 d_{ij}}\ \frac{(\eta_i-\eta_j)(X_j-X_i+\bar{\xi}_i-\bar{\xi}_j)+(\xi_j-\xi_i)(Y_j-Y_i+\bar{\eta}_i-\bar{\eta}_j)}{(\xi_j-\xi_i)^2+(\eta_j-\eta_i)^2}. \tag{3}$$

Note that (2) and (3) contain heuristic weighting factors $\alpha_2 \approx 25/H$ and $\alpha_3 \approx 25/H$. Their denominators contain only one factor of H because the percentages of stretching and bending distortion should not depend on H.

3.2. Width Control

The preceding section shows how to construct distortion measures in the relatively easy case of position and shape control. In the more difficult case of width control, we can draw on experience and concentrate on feature recognition. Specifically, we need to take a character shape given as a polygonal outline, and find features that can be said to have width. This may be done by the well-known technique of *medial axis decomposition* introduced by Blum.[4] Most of the work necessary to compute the decomposition was done by Montanari[14] and later improved by Lee.[12] The prototype implementation used Fortune's algorithm for line segment sites.[5]

The resulting medial axis decomposition of a set of polygons is a *radius function* and a set of tree-like structures such as those shown in Figure 16a. If the radius function is used to define a set of circles, one centered at each point on the medial axis structures, then the union of all the circles is the original polygon. Another way to look at it is that the medial axis structures identify all interior points equidistant from disjoint parts of the polygonal outlines and the radius function gives the distance.

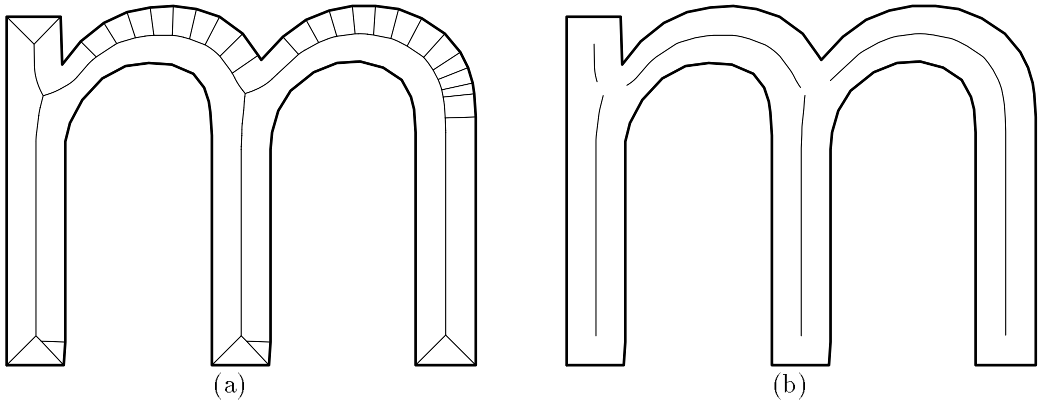


Figure 16: (a) A polygonal outline and its medial axis tree. (b) The same outlines with the medial axis pruned by using a threshold angle of 146°.

The only problem with the medial axis structures shown in Figure 16a is that they include many extraneous branches that do not appear to be the centers of strokes. What is needed is a way to cut out the extraneous branches and leave only the truly necessary ones as shown in Figure 16b. Montanari does this by using a thresholding process based on what may be called the *opposition angle*. The opposition angle for a point P on a medial axis of a set of polygonal outlines O is found by looking at the points where O intersects the P-centered circle defined by the radius function as shown in Figure 17. The medial axis construction guarantees that there must be at least two distinct intersection points A and B, hence we can find an angle APB measured so as to be between zero and 180°. Normally, there are two intersection points and they give the opposition angle APB. If there are more than two intersection points, the opposition angle is the maximum angle APB over all pairs of intersection points A and B.

Since the perpendicular displacement varies linearly along the edge, the mean squared displacement is

$$\int_0^1 \left(\Delta_i + t(\tilde{\Delta}_j - \Delta_i)\right)^2 dt = \int_0^1 \Delta_i^2 + 2t\Delta_i(\tilde{\Delta}_j - \Delta_i) + t^2(\tilde{\Delta}_j - \Delta_i)^2\, dt$$
$$= \Delta_i^2 + \Delta_i(\tilde{\Delta}_j - \Delta_i) + \frac{(\tilde{\Delta}_j - \Delta_i)^2}{3}$$
$$= \frac{\Delta_i^2 + \Delta_i\tilde{\Delta}_j + \tilde{\Delta}_j^2}{3}$$
$$= \frac{(\Delta_i + \tilde{\Delta}_j)^2}{4} + \frac{(\tilde{\Delta}_j - \Delta_i)^2}{12}.$$

Thus adding to the distortion function a weighting factor $\alpha_1 \approx 1000/H^3$ times the integral with respect to arc length of the squared perpendicular displacement is equivalent adding distortion measures

$$(\Delta_i + \tilde{\Delta}_j)\sqrt{\frac{\alpha_1 d_{ij}}{4}} \quad \text{and} \quad (\tilde{\Delta}_j - \Delta_i)\sqrt{\frac{\alpha_1 d_{ij}}{12}} \tag{1}$$

where

$$d_{ij} = \sqrt{(\xi_j - \xi_i)^2 + (\eta_j - \eta_i)^2}$$

and i and j range over all pairs of consecutive vertices i and j. In practice either $j = i + 1$, or i is the highest numbered vertex on a polygonal outline and j is the lowest numbered vertex.

Why does α_1 need to have H^3 in the denominator in order to make (1) independent of H? One factor of H comes in because the total arc length of the (unscaled) outlines is proportional to H. The other factor of H^2 allows Δ_i and $\tilde{\Delta}_j$ to be fractions of H.

Using the two distortion measures given by (1) for each edge in the polygonal outlines gives a good measure of the overall magnitude of the adjustment to the outlines. One way to get a similar measure for the local distortion of the shape of the outlines is to consider changes in the length and direction of one segment of the polygonal outlines. Figures 15b and 15c show two ways the adjusted outlines can distort the edge between vertices i and j shown in Figure 15a. Since Figure 15b shows a 50% relative error in the length component of the edge in the adjusted outline, there is 50% *stretching distortion* in this case. In Figure 15c on the other hand, the edge has acquired a perpendicular component equal to 31% of the desired length. This represents a 31% *bending distortion.*

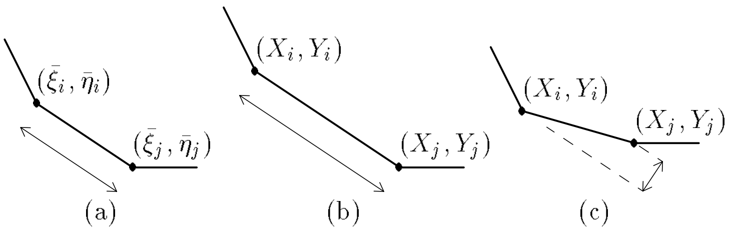


Figure 15: (a) A typical edge of the scaled outlines. (b) A corresponding edge from the adjusted outline showing 50% stretching distortion. (c) An alternative version of the adjusted edge showing 31% bending distortion.

Since the stretching and bending distortions are properties of the edge as a whole, it is not necessary to integrate with respect to arc length as we did for the perpendicular displacement. All we need do is give distortion measures for each edge including a square root of arc length factor so that the sum of squared distortion measures is weighted by arc length. For each pair of adjacent vertices i, j, the distortion measure for stretching is

$$\sqrt{\alpha_2 d_{ij}}\; \frac{(\xi_j - \xi_i)(X_j - X_i + \bar{\xi}_i - \bar{\xi}_j) + (\eta_j - \eta_i)(Y_j - Y_i + \bar{\eta}_i - \bar{\eta}_j)}{(\xi_j - \xi_i)^2 + (\eta_j - \eta_i)^2} \tag{2}$$

that I_1, I_2, and I_3 are to be chosen so as to minimize the distortion function. This requires the distortion function to be chosen carefully so that the optimal I_1, I_2, and I_3 will be what is intended, but this is easy to verify for the simple cases given below.

It turns out to be fairly simple to give distortion measures that are linear in the vertex coordinates and the auxiliary variables I_1, I_2, and I_3. Naturally, the distortion measures are heuristic in nature and contain constants whose settings are based on aesthetic criteria. Suggested values such as $\alpha_1 \approx 1000/H^3$ are given when such constants are introduced, but these can be changed to alter the relative importance of the various types of distortion. The constants are given in terms of a parameter H that describes the approximate maximum vertical extent of the original character outlines. Thus we can let $H = 41.5$ if the original outlines describe a ten point font scaled so that there are 4.15 pixel units per point. This allows the heuristic weighting factors to be chosen so as to make the distortion measures independent of the scale of the original outlines.

Actually, the major problem in finding distortion measures is not the scaling or the heuristics, but rather the recognition of features whose distortion is to be controlled. Thus the rest of this section is devoted to the categories of features for which distortion measures are needed. An attempt has been made to give reasonable distortion measures for important features, but there is plenty of room for additions and improvements.

Perhaps the easiest class of features to recognize are those that deal with the overall shape of the character. Thus we begin in Section 3.1 by giving distortion measures for limiting the variation in shape and position between the original and adjusted outlines. This experience makes it easier to find stroke-like features and create distortion measures for controlling their width. Section 3.2 shows how to accomplish this task with the aid of Voronoi Diagrams. Next Section 3.3 covers the problem of finding places where an outline is horizontal or vertical and needs to be adjusted so that it fits the pixel grid. Section 3.4 examines when to use integer offset vectors and how distortion measures can be used to enforce them. Section 3.5 then gives methods for finding approximate symmetry and using distortion measures to ensure that the adjusted outlines preserve the symmetry. Finally, Section 3.6 gives a method for controlling the relative values of stroke widths and Section 3.7 discusses the relative positioning of places where the character outlines align vertically or horizontally.

3.1. Controlling Position and Shape Distortion

The reason for controlling the overall shape and positioning of the adjusted outlines is to ensure that they match the original outlines as closely as possible. In Figures 14 for instance, no part of the outline is shifted by more than about 0.6 pixel units. (See also Figure 2).

For shape control, we create distortion measures that reflect the amount by which the adjusted outlines are shifted, using constants ξ_i and η_i for the original position of the vertex whose adjusted position is given by the variables X_i and Y_i. When outlines are scan converted, $\bar{\xi}_i$ and $\bar{\eta}_i$ will be used to indicate linear expressions involving an adjustable scale parameter. Thus there are the original outlines with vertices given by constants of the form (ξ_i, η_i), scaled outlines with vertices $(\bar{\xi}_i, \bar{\eta}_i)$, and adjusted outlines with vertices (X_i, Y_i). Since the nominal value for the scale factor is assumed to be one, the distinction between (ξ_i, η_i) and $(\bar{\xi}_i, \bar{\eta}_i)$ is unimportant for now except that Section 5 requires distortion measures to be linear in $\bar{\xi}_i$ and $\bar{\eta}_i$. (Most distortion measures turn out to be proportional to the hidden scale factor).

With this notation, consider perpendicular displacements relative to the edge from $(\bar{\xi}_i, \bar{\eta}_i)$ to $(\bar{\xi}_j, \bar{\eta}_j)$. The dispacement of (X_i, Y_i) is

$$\Delta_i = \frac{(\eta_i - \eta_j)(X_i - \bar{\xi}_i) + (\xi_j - \xi_i)(Y_i - \bar{\eta}_i)}{\sqrt{(\xi_j - \xi_i)^2 + (\eta_j - \eta_i)^2}}$$

and the displacement of (X_j, Y_j) is

$$\tilde{\Delta}_j = \frac{(\eta_i - \eta_j)(X_j - \bar{\xi}_j) + (\xi_j - \xi_i)(Y_j - \bar{\eta}_j)}{\sqrt{(\xi_j - \xi_i)^2 + (\eta_j - \eta_i)^2}}.$$

scan-conversion process once the outlines have been adjusted to fit the pixel grid, but it should not be viewed as a possible replacement for the present work since Plass and Hochschild do not allow the types of shape changes that result from adjusting the outlines.

To build a function that measures distortion of the features given in Section 2, it is necessary to allow small changes in the shape of the character so as to make key parts of the character fit the pixel grid better. For instance, the pimples and flat spots in Figure 10 arise when the boundary curves from the original outlines have horizontal tangent points near the middle of a pixel square rather than near the top or bottom edge.

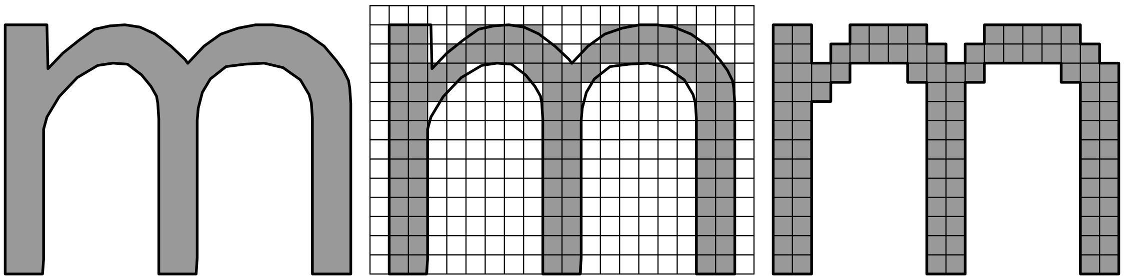
To accommodate this notion of small changes in shape to fit the pixel grid, it is natural to choose a fixed scan-conversion algorithm and then adjust the outlines prior to scan conversion so as to minimize the distortion that will result. For example, Figure 14 shows how the outlines from Figure 2 can be adjusted to improve the scan-converted bitmaps. With these adjusted outlines as its argument, the distortion function can evaluate what would result from applying the chosen scan-conversion algorithm. This allows the distortion function to see how the adjusted outline fits the pixel grid and impose a penalty based on the magnitude of the adjustments to the original outlines.

Note that the distortion depends on both the original and the adjusted outlines, but the original outlines remain fixed while adjusted outlines need to be optimized. Thus it is convenient to build the original outlines into the distortion function so that when finding tuned bitmaps for several outline characters, each has its own version of the distortion function.

For simplicity, we shall assume that the outlines are polygonal so that the adjusted outlines can be identified by the coordinates of the vertices. For instance, the outline of the "m" in Figure 14a is a polygon with 58 vertices. An adjustment to the outline can be given as a vector

$$(X_1,\ Y_1, X_2, Y_2, \ldots,\ X_{58},\ Y_{58})$$

which can then be an argument to the distortion function.



(a) (b) (c)

Figure 14: (a) An adjusted character shape and its polygonal outline. (b) The same outlines with the pixel grid and shaded squares for the scan-converted bitmap. (c) The resulting bitmapped character.

A simple way to construct the distortion function would be to write it as a sum of squares

$$T_1^2 + T_2^2 + T_3^2 + \cdots,$$

where each T_i is an expression in the vertex coordinates that represents a particular kind of distortion. These *distortion measures* T_1, T_2, ... should be as simple as possible, but it does not suffice to use linear combinations of vertex coordinates since positioning relative to the pixel grid depends on the fractional parts of vertex coordinates. One way to get around this problem is to introduce new *I-variables*

$$I_1,\ I_2,\ I_3,\ \ldots$$

that can only take on integer values. If the distortion function needs to refer to the fractional parts of X_5, Y_{17} and X_{23}, they can then be written $X_5 - I_1$, $Y_{17} - I_2$, and $X_{23} - I_3$ with the understanding

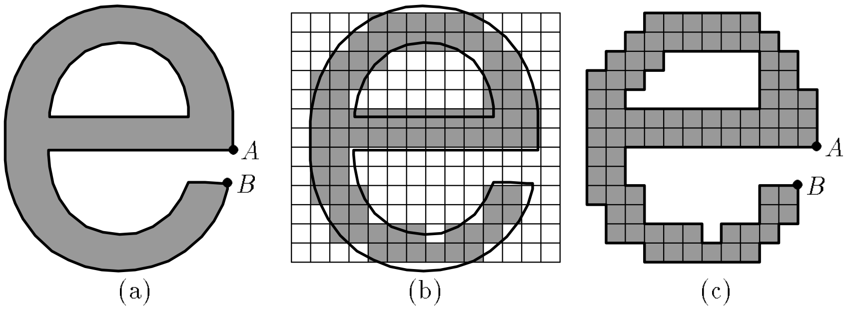


Figure 12: (a) Outlines for an "e" with points A and B almost vertically aligned. (b) The same outlines with the pixel grid and shaded squares for the scan-converted bitmap. (c) The resulting bitmap showing loss of vertical alignment.

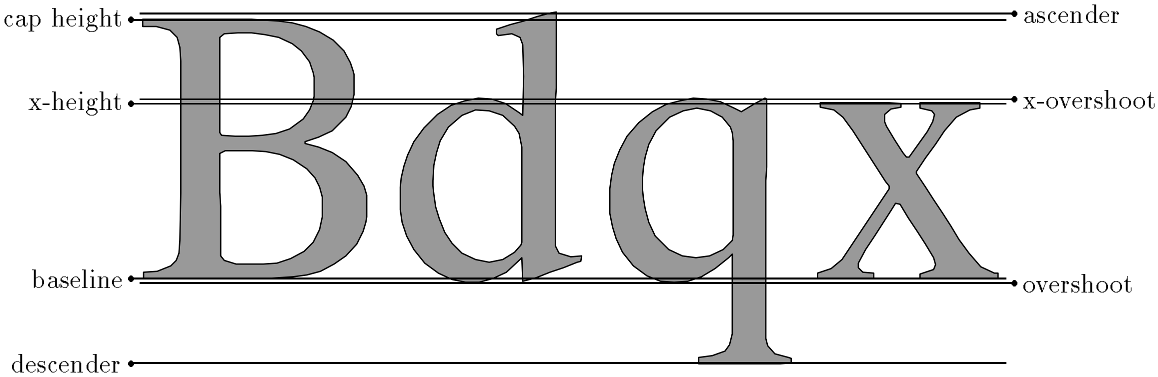


Figure 13: Samples of Times Roman showing heights at where parts of many characters line up.

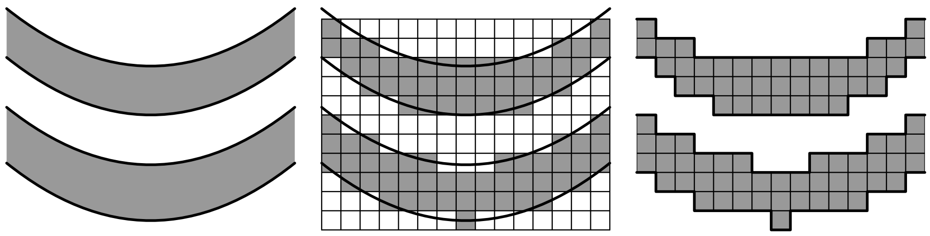


Figure 10: (a) Portions of two strokes with their outlines shown as bold lines. (b) The same outlines with the pixel grid and shaded squares for the scan-converted bitmap. (c) The bitmaps for the three strokes with boundary curves showing various degrees of distortion.

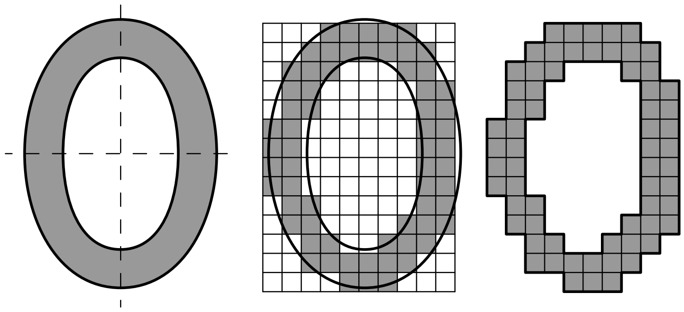


Figure 11: (a) Outlines for an "0" with dashed lines for symmetry axes. (b) The same outlines with the pixel grid and shaded squares for the scan-converted bitmap. (c) The resulting bitmap showing loss of symmetry.

Other important features deal with overall shape and positioning. Most of these are conceptually simple and are readily measured by the distortion function as we see in the next section. One of the less obvious features of this type is the alignment of different parts of a character shape that fall at similar x or y coordinates. Figure 12 illustrates what can happen when this type of alignment is poorly controlled. In the outline version of the character in Figure 12a, points A and B have nearly identical x coordinates, while the corresponding points on the scan-converted characters in Figure 12c have x coordinates that differ by one pixel unit. Thus the difference in x coordinates between points A and B is a feature that needs to be controlled in order to get a good bitmap version of Figure 12a.

This same type of alignment occurs on a font-wide basis in that as characters are lined up side by side, important parts of different characters can line up at the same y-coordinate. This is so important that font designers have names for the various heights at which such alignments occur as shown in Figure 13. This alignment must be preserved when producing bitmap fonts since even a one pixel variation in baseline tends to be very noticeable on a device such as a laser printer. It is especially important to consider the relative values of heights that are close together. For example, the baseline and the overshoot height are so close together in Figure 13 that it would be a bad idea to let them differ by one pixel unit on a device where the pixel size is relatively large.

3. The Distortion Function

Since the distortion function is intended to measure how the scan-converted bitmaps distort the features of the original outlines, one could build such a function by modeling what happens when the bitmaps are printed, and compare this to the desired outlines. Plass and Hochschild have used this technique with some success although it takes a lot of computing to find bitmaps that minimize their version of the distortion function.[16]. Their algorithm could be used in place of the standard

pixels per unit length but it is more practical with the present application to use some ideas from [8] and [9] to construct special outlines whose width matches the bitmap.

For example, Figure 8a shows a pair of outlines that have the *integer offset* property defined in [8] and [9]. When superimposed on the pixel grid in Figure 8b, the separation between the two halves of the outline for the upper stroke is such that the vector $(1, 2)$ measured in pixel units just spans the gap. Similarly, for the outline of the lower stroke, the integer offset vector $(1, 1)$ just spans the gap. If we scan convert each stroke by turning on the pixels whose centers lie inside the outline, a theorem from [9] guarantees that the number of pixels per unit length matches the original outlines. Thus the lower stroke has 19 pixels turned on along a line about 15.52 pixel units long and the ratio 19/15.52 exactly matches the 1.224 pixel-unit width of the lower stroke in Figure 8a.

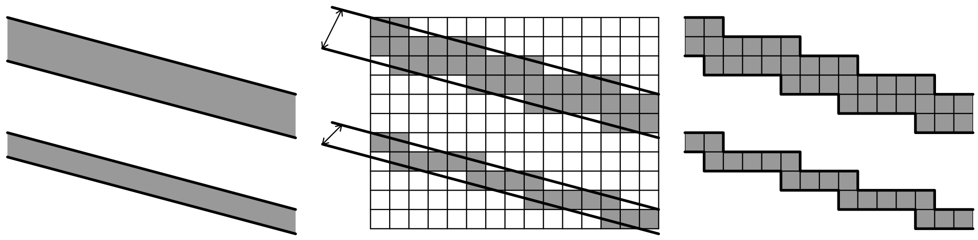


Figure 8: (a) Portions of two strokes with their outlines shown as bold lines. (b) The same outlines with their integer offset vectors and the pixel grid with shaded squares for the scan-converted bitmap. (c) The bitmaps for the two strokes

There will be more discussion of integer offset vectors in the next section when we give the distortion function. Figures 9a–c show what can happen when scan converting outlines that do not have the integer offset property. All three strokes in Figure 9a have the same width (about 1.48 units), but the scan-converted bitmaps in Figure 9c have two strokes of very different width and one stroke whose width is very nonuniform.

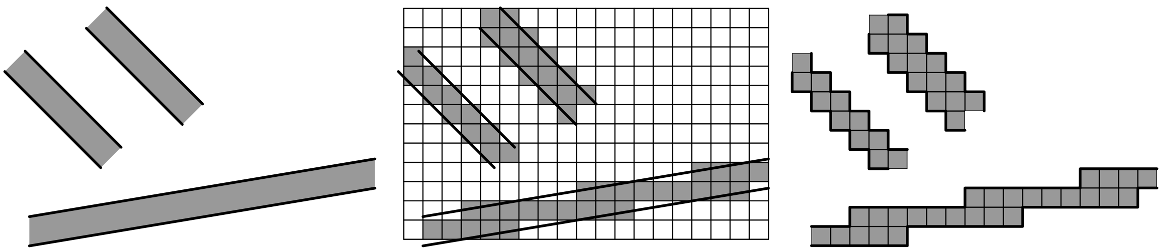


Figure 9: (a) Portions of three strokes with their outlines shown as bold lines. (b) The same outlines with the pixel grid and shaded squares for the scan-converted bitmap. (c) The bitmaps for the three strokes

Another type of feature that needs to be controlled is the shape of curves near "critical directions." Figures 10a–c show how scan conversion affects the appearance of such curves. Each of the two strokes in Figure 10a is bounded by two identical curves, one above the other. In the scan-converted versions of these curves in Figure 10c, the upper boundary of the upper stroke has a long "flat spot," while the lower boundary of the lower stroke has a "pimple." Font designers know that it is desirable to avoid both pimples and flat spots when scan converting curved outlines. Further details appear in D. E. Knuth's *The METAFONTbook*.[11]

Another important feature of character shapes that can easily get lost in the scan-conversion process is symmetry. The "0" in Figure 11a has symmetry about vertical and horizontal axes, but when scan converted by turning on the pixels centered inside the outlines, the symmetry is lost as shown in Figure 11c.

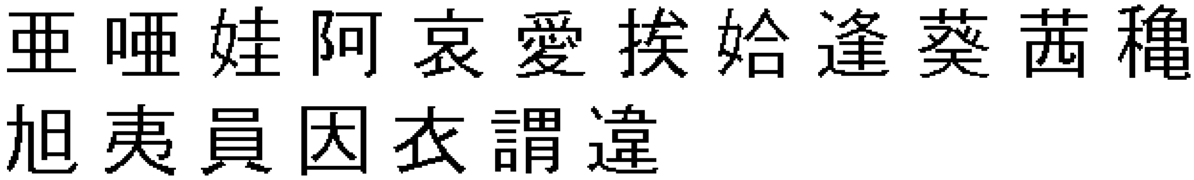


Figure 6: Automatically tuned bitmaps for 30 point Japanese Kanji at 100 pixels per inch generated from the outlines used to generate Figure 5.

of the distortion function and the form its argument should take. These ideas are covered in Section 3 where we introduce the distortion function and give techniques for computing the matrix that defines it. A font-wide version of the distortion function covered in Section 4 uses the same principles to measure nonuniformity in the font as a whole.

Well known ideas based on work by Lovász [13] give good approximate minima for both versions of the distortion function. By implementing this as a two step process, we retain the idea that a single set of character outlines can be used to generate bitmaps at any desired size. Section 5 describes a way to generate a fairly compact set of instructions that can be used to find low distortion bitmaps once a scale factor is chosen. Finally, Section 6 explains how to adapt the distortion function to the case where the character outlines are curved rather than polygonal.

2. Features to Recognize

In order to produce high quality bitmap fonts from outlines, it is necessary to preserve important features as well as possible. But what features are important and how are they best preserved? The answer to this question is necessarily somewhat open-ended, but we can give some important features and suggest ways of dealing with them. The methods presented in this paper can accommodate additional features if desired.

Perhaps the most important features of a character shape are the widths of the strokes that compose it. For instance, the Helvetica "m" has three vertical strokes connected by two curved strokes. The widths of these strokes can be measured at various points as shown in Figure 7.

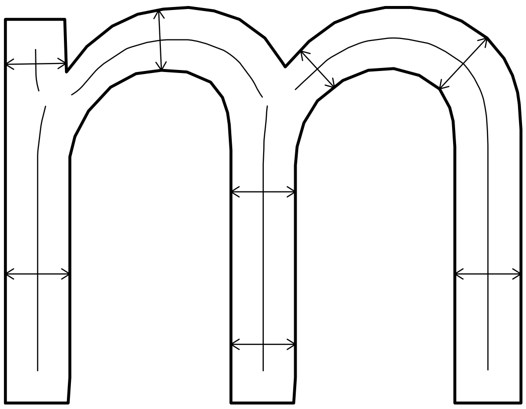


Figure 7: The outline of a Helvetica "m" with thin lines marking the centers of strokes.

It is somewhat more difficult to measure the stroke widths in a bitmapped character produced by scan conversion, but it is clear that the middle stem in Figure 2c is three pixels wide and the other two vertical stems are two pixels wide. (Since all stems are 2.2 pixel units wide in Figure 2a, the width of three pixels for the middle stem is a significant distortion).

Truly measuring width distortion for strokes that are neither vertical nor horizontal requires a definition of "width" that applies to bitmaps. As explained in [8], this can be done by counting

problems in a sample of 18 point Helvetica at 100 pixels per inch.[1] Scan converting outlines at this low resolution produces a "ransom note" appearance due to asymmetries and excessive variations in stroke widths. See for example the appearance of "m" in the first word on line 4. Much better bitmapped characters can be obtained by applying the methods described in this paper to the same Helvetica outlines. Figure 4 gives a sample of the resulting font.

teaching the blessings of liberty regulated by law, and
inculcating love and reverence for the great principles
of government as derived from the inalienable rights of
man to life, liberty, and the pursuit of happiness.

Figure 3: 18 point Helvetica scan converted from outlines at 100 pixels per inch.

teaching the blessings of liberty regulated by law, and
inculcating love and reverence for the great principles
of government as derived from the inalienable rights of
man to life, liberty, and the pursuit of happiness.

Figure 4: Automatically tuned bitmaps for 18 point Helvetica at 100 pixels per inch, generated from the outlines used to generate Figure 3.

One way to achieve results similar to figure 4 is to come up with a set of instructions for adjusting the outlines to control the phase relative to the pixel grid. For instance, R. D. Hersch presents a technique for using "grid constraints" to adjust the outlines prior to scan conversion. [7] Other approaches to the problem are discussed by Apley [2]. Unfortunately, whatever form the grid constraints take they usually have to be generated laboriously by hand.

In order to achieve the results in Figure 4 via a fully automatic process, we need to recognize features such as stem widths and make sure the scan-conversion process does not distort them more than necessary. Thus the two main subproblems are feature recognition and minimization of the distortion of important features during scan conversion. The idea of this work is to use the result of feature recognition to build a function that measures the distortion after scan conversion. By trying to minimize this function, we can find "optimal" or nearly optimal scan-converted bitmaps. These nearly optimal bitmaps are based on such general concepts that they are appropriate for almost any kind of characters, including the Japanese Kanji shown in Figures 5 and 6. Compare for example the widths of the horizontal and vertical strokes in the very first character.

亜 唖 娃 阿 哀 愛 挨 姶 逢 葵 茜 穐
旭 夷 員 因 衣 謂 違

Figure 5: 30 point Japanese Kanji scan converted from outlines at 100 pixels per inch.

The generality of the approach is achieved by being careful about the kinds of features to recognize. After these features are introduced in Section 2, the next task is to determine the nature

[1] The same effects appear in 6 point Helvetica at 300 pixels per inch, but the three-fold magnification makes the illustration clearer and less susceptible to artifacts of the printing process.

Generating Automatically-Tuned Bitmaps from Outlines

John D. Hobby

1. Introduction

Hardware and software for electronic typesetting has long made use of outlines to describe letter shapes, but the actual printing engine or display device usually requires bitmap input. The outline representation is used because it is compact and it is easily scaled to produce different type sizes. It is also much easier to design character shapes in outline form, since this avoids the discreteness of the raster grid imposed by the bitmap representation.

The Screen fonts for the Apple Macintosh exemplify the problems of producing multiple type sizes by scaling bitmap fonts. The font samples shown in Figure 1 appear in Apple's *Technical Introduction to the Macintosh*.[10] One particularly unfortunate aspect of the bitmap scaling process illustrated in the figure is how the jagged edges in the original bitmap get magnified when scaling up. For this reason, it is generally considered preferable to start with outlines instead of bitmaps.

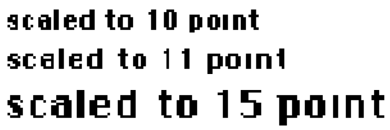


Figure 1: These screen fonts for the Apple Macintosh are generated by scaling 12 point bitmap fonts. The resolution is about 75 pixels per inch.

The difficulty in generating scaled bitmaps from outlines is that well-known scan-conversion algorithms usually do not produce good looking bitmap fonts. For instance, Figure 2a shows the outline of an "m" which the scan-conversion process diagramed in Figure 2b renders as shown in Figure 2c. Even though all three stems are essentially the same width in the original outline, the middle stem comes out half again as wide as the others in bitmapped version. This difference in width is a side effect of how the relevant parts of the original outline happened to fall on the pixel grid.

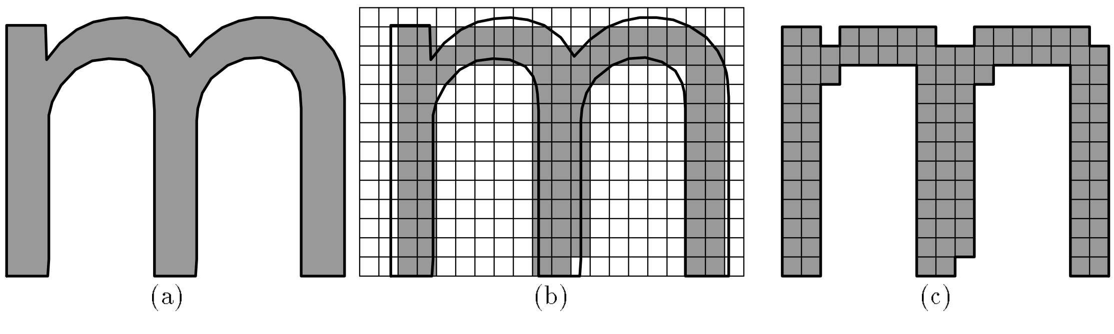


Figure 2: (a) A character shape and its polygonal outline. (b) The polygonal outline and the pixel grid with the scan-converted bitmap indicated by shaded squares. (c) The resulting bitmapped version of the character.

When the outlines for an entire font are scan converted in this manner, features such as stem widths become quite nonuniform due to the effects of the pixel grid. Figure 3 illustrates these

Generating Automatically-Tuned Bitmaps from Outlines

John D. Hobby

Abstract

Consider the problem of generating bitmaps from character shapes given as outlines. The obvious scan-conversion process does not produce acceptable results unless important features such as stem widths are carefully controlled during the scan-conversion process. This paper describes a method for automatically extracting the necessary feature information and generating high quality bitmaps without resorting to hand editing. Almost all of the work is done in a preprocessing step, the result of which is an intermediate form that can be quickly converted into bitmaps once the font size and device resolution are known.

A heuristically defined system of linear equations describes how the ideal outlines should be distorted in order to produce the best possible results when scan converted in a straight-forward manner. The Lovász basis reduction algorithm then reduces the system of equations to a form that makes it easy to find an approximate solution subject to the constraint that some variables must be integers.

The heuristic information is of such a general nature that it applies equally well to Roman fonts and Japanese Kanji.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation—*digitizing and scanning*; I.5.4 [**Pattern Recognition**]: Applications—*text processing*

General Terms: Algorithms

Additional Key Words and Phrases: Scan-conversion; Fonts; Feature recognition; Lovász basis reduction