UNDER REVIEW

RESCIENCE C

Replication / Computer Graphics

[Re] Curve-Fitting with Piecewise Parametric Cubics

Daniel Perazzo[1], Davi Guimarães Nunes Sena Castro[2], Luiz Velho[1], and Luiz Henrique de Figueiredo[1]
[1]IMPA, Rio de Janeiro, Brazil – [2]Universidade Federal Fluminense, Niterói, Brazil

Edited by
(Editor)

Received
August 11, 2025
Revised
October 8, 2025

Published
–

DOI
–

Abstract We report on a replication of the paper "Curve-Fitting with Piecewise Parametric Cubics" by Michael Plass and Maureen Stone presented at SIGGRAPH 1983. Their paper proposes a method for fitting a cubic spline to a sequence of points. The method uses dynamic programming to fit a global spline composed of selected cubic Bézier curves that fit the data locally. Our code and data are publicly available.
A replication of Plass-Stone-1983.

1 Introduction

The early 1980's were exciting times. Donald Knuth finished TEX in 1982 and Metafont by 1984. Apple launched the original Macintosh in January 1984. Adobe was founded in December 1982 to develop the PostScript page description language, after John Warnock and Charles Geschke left Xerox PARC. In the context of computer graphics, all these endeavors had in common the computational description of shapes using piecewise parametric cubics based on Bézier curves. Some shapes are designed interactively; others are fit to scanned data. Of special interest at that time were typographic glyphs needed to support high-quality desktop publishing. The paper "Curve-Fitting with Piecewise Parametric Cubics" by Michael Plass and Maureen Stone [1], presented at SIGGRAPH 1983, is a landmark of that era.
This paper is a replication of the Plass–Stone paper following their methodology. Our code and data are publicly available [2].

2 Methodology

Plass and Stone seek "to reliably derive an efficient piecewise parametric cubic polynomial representation from a sequence of closely-spaced points that approximate the desired curve." They leave to the user to decide how good the approximation is based on the quality of the input sample points. Shapes can have sharp corners, straight edges, and other features that need to be preserved in the output representation. This is especially true for typographic glyphs. Although Plass and Stone briefly mention that they try to find corners automatically from scanned data, they also seem to imply that ideally it would best to leave to the user the segmenting of the shape outline into corners and curves. Between corners, they seek to fit the best piecewise cubic by selecting the best set of joints or knots that give a curve that is tangent continuous in the geometric sense: two cubics meeting at a knot share the same tangent direction at the knot. Here is an overview of their methodology. The details (and our take on them) are explained later.

Input The input is a sequence of closely-spaced *sample points* in the plane that approximate the boundary of a shape. If the shape has several boundary curves, treat each curve separately.

Correspondence should be addressed to Daniel Perazzo (daniel.perazzo@impa.br)
The authors have declared that no competing interests exist.
Code is available at https://github.com/danperazzo/plass_reproduction..

Output The output is a *cubic spline* that best approximates the sample points. A cubic spline is a sequence of concatenated cubic segments given as Bézier curves. The extreme points of the cubic segments are called *knots*. Except at *corners*, the spline is smooth in the G^1 sense: the cubic segments that meet at each knot have the same tangent *directions* (not *vectors*!) at the knot.

Knots The key ingredients of the method are the selection of the knots and corners, and the computation of the cubic Bézier curves that best fit the sample points between two consecutive knots. In principle, the knots can be arbitrary, but this is far too much freedom. Plass and Stone select knots from the sample points using dynamic programming to minimize the global approximation error of the spline.

Curve fitting The main problem in fitting a parametric curve γ to a sequence of sample points $p_1, \dots, p_n$ is that no parameter values are given, that is, we do not know the parameter values t_i such that $\gamma(t_i) = p_i$. Plass and Stone iteratively adjust the parameters t_i such that $\gamma(t_i) \approx p_i$. The approximation error of the curve is $\sum_{i=1}^{n} \|\gamma(t_i) - p_i\|^2$. Given the parameter values t_i, the best cubic curve is easily found by minimizing this error with respect to the coefficients of the curve. This is the method of least squares.

3 Preparing the data

The input sequence of points needs to be preprocessed for two reasons: to simplify the sample, thus reducing noise and complexity, and to assign tangent directions to the points, a prerequisite for a G^1 smooth spline.

Selecting potential knots Searching for the best set of knots to fit a sequence of sample points can be extremely computationally expensive: if there are n input points, then there are 2^n possible knot subsequences. Plass and Stone avoid this exhaustive search of exponential cost by using dynamic programming, thus reducing the cost to $O(n^3)$. Although this is still costly, it is far more tractable. In 1983, such a cost could have been prohibitive (Plass and Stone did not report any timings). In 2025, thanks to the speed of modern computers, this computation is feasible, even if it remains demanding. Instead of using all input points as potential knots, Plass and Stone use a reduced set: the vertices of a polygonal approximation of the sample points. They gave no details about this step, stating that the choice of the polygonization method is not particularly relevant. We used the standard Ramer–Douglas–Peucker simplification algorithm [3, 4] to find a good polygonal approximation to the sample points.

Identifying corners While at regular knots the tangent direction is fixed to achieve a smooth fit (see below), at corners no tangent direction is enforced during the fitting. Plass and Stone define corners as knots whose angle in the polygonal approximation is less than $135°$. We follow their recommendation but we test the cosine instead of computing the angle.

Selecting tangents After selecting the potential knots and identifying the corners among them, Plass and Stone give a tangent direction to each regular knot by fitting a cubic curve locally around the knot and evaluating its derivative at the knot. Plass and Stone call this "setting tangents using fairly local information". We used a sliding window of sample points around the knot to fit a Bézier curve. The window is constrained to lie between two adjacent corners. In other words, the window does not straddle corners; otherwise, knots that are close to corners can have poor tangents, since by definition the shape curve changes direction abruptly at corners.

4 Fitting a cubic Bézier curve

To fit a single cubic Bézier curve to a (sub)sequence of points $p_1, \ldots, p_n$, Plass and Stone use an iterative process that alternates between estimating the curve by solving a least squares problem, possibly with boundary conditions when tangent directions are present at the endpoints, and updating the parameter values t_i associated with each point using the Newton–Raphson method. Repeating this iterative process multiple times improves the fitting accuracy, resulting in a good approximation of the points. This iterative procedure continues until the fitting error stabilizes or a maximum number of iterations is reached.

Initial parameters Plass and Stone initialize the parameters t_i as normalized cumulative chord lengths:

$$t_i = \frac{s_i}{s_n}, \quad \text{where} \quad s_i = \sum_{k=1}^{i-1} \|p_{k+1} - p_k\|, \quad i = 1, \ldots, n.$$

Note that $t_1 = 0$ and $t_n = 1$, as befits a Bézier curve.

Least squares The cubic Bézier curve is given by

$$\gamma(t) = (1-t)^3 B_0 + 3(1-t)^2 t B_1 + 3(1-t)t^2 B_2 + t^3 B_3,$$

with endpoints fixed at $B_0 = p_1$ and $B_3 = p_n$. The other two control points B_1 and B_2 are free and are found using least squares respecting the boundary conditions, if any. We consider three scenarios depending on whether the endpoints are knots with tangent constraints or corners without tangent constraints. The endpoints B_0 and B_3 are always fixed at p_1 and p_n, respectively. The tangent directions at these endpoints, if present, are denoted by T_0 and T_3. The coordinates of the free control points are $B_1 = (x_1, y_1)$ and $B_2 = (x_2, y_2)$. They are found by minimizing the approximation error $\sum_{i=1}^{n} \|\gamma(t_i) - p_i\|^2$ using least squares on the optimization variables described below:

No tangent directions. The optimization variables are x_1, y_1, x_2, y_2.

One tangent direction. When the tangent direction is T_0, we have $B_1 = p_0 + \alpha_1 T_0$ and the optimization variables are α_1, x_2, y_2. When the tangent direction is T_3, we have $B_2 = p_n + \alpha_2 T_3$ and the optimization variables are x_1, y_1, α_2.

Two tangent directions. We have $B_1 = p_0 + \alpha_1 T_0$ and $B_2 = p_n + \alpha_2 T_3$ and the optimization variables are α_1, α_2.

Parameter update After estimating the curve, we update the parameter values t_i for each point p_i so that $\gamma(t_i)$ is the point of the curve closest to p_i. Thus, to minimize

$$g(t) = \|\gamma(t) - p_i\|^2 = (\gamma(t) - P_i) \cdot (\gamma(t) - P_i),$$

we find the root of (half) its derivative

$$f(t) = g'(t) = (\gamma(t) - P_i) \cdot \gamma'(t).$$

The Newton–Raphson iteration finding the root of f is

$$t \leftarrow t - \frac{f(t)}{f'(t)},$$

with

$$f'(t) = \gamma'(t) \cdot \gamma'(t) + (\gamma(t) - P_i) \cdot \gamma''(t).$$

After updating the t_i, we normalize them to remain within the interval $[0, 1]$ via the affine transformation

$$t \mapsto \frac{t - t_{min}}{t_{max} - t_{min}},$$

where t_{min} and t_{max} are the smallest and largest of the t_i, respectively.

5 Selecting the best knots

Plass and Stone adopt a dynamic programming approach to select the best set of knots, one that minimizes the total fitting error across the entire sample. Directly minimizing the total error can lead to overfitting, as we may end up selecting the maximum number of possible knots, which trivially reduces the error. To prevent this, Plass and Stone introduce a regularization parameter $\tau > 0$ that penalizes the introduction of additional knots, thus favoring splines with few cubic segments. More precisely, the local cost of approximating the points between two consecutive knots p_i and p_j by a single cubic Bézier curve γ is

$$e_{ij} = \sum_{k=i}^{j} \|\gamma(t_k) - p_k\|^2 + \tau.$$

Thus, the local cost is composed of the fitting error minimized in the previous section plus a penalty for introducing a curve segment in the spline.
Let $D[i]$ denote the minimum cumulative error for fitting the curve from point 1 to point i. Then, starting with $D[1] = 0$, the value of $D[i]$ is given recursively by

$$D[i] = \min_{1 \le k < i} \left(D[k] + e_{ki} \right).$$

Following Plass and Stone, we solve this recurrence using dynamic programming and recover the optimal set of knots via backtracking. This is a much simpler special case of dynamic programming since we are not using the complete graph. Recall that this procedure is applied only to the potential knots identified during the polygonization step and only between consecutive corners.

6 Implementation and Results

Our implementation [2] uses Python with Numpy [5].

6.1 Parameters

The method depends on several parameters, which are mostly never discussed by Plass and Stone. They only say "The closeness of the fit is specified by a user-defined tolerance; it is the user's job to consider the resolution and accuracy of the input when specifying the tolerance."

Tolerance for polygonal approximation We stop the Ramer–Douglas–Peucker simplification algorithm when all sample points between two points a and b are at a distance at most ε from the segment ab. We assume that ε is the "user-defined tolerance" mentioned above. The exact value of ε depends on the scale of the coordinates of the sample points and whether the sample has a lot of abrupt changes of direction or whether it is smoother. For noise-free samples having (near) integer coordinates, such as obtained from font glyphs or by tracing images, we mostly used $\varepsilon = 1$, but in some cases we used $\varepsilon = 0.5$ or $\varepsilon = 0.25$. For samples with significant noise, a slightly higher ε is needed to avoid complications in corner detection.

Identifying corners Plass and Stone define corners as knots whose angle in the polygonal approximation is less than $135°$. We have used this value as given; we have not experimented with other values.

Selecting tangents To give a tangent to a knot, we fitted local Bézier curves using a sliding window with 5 sample points before the knot and 5 points after it. For samples with significant noise, we got better results with sliding windows to 15 to 25 points before and after the knot. Recall that the sliding windows do not straddle corners.

Fitting a cubic Bézier curve We used at most 20 Newton–Raphson steps to adjust the parameter values t_i. We used at most 100 parameter adjustment steps to fit the curve. In both processes we stopped when successive approximation errors differ by less than 10^{-7}. We noted that the first time we adjust the parameters t_i, in the vast majority of cases it took around 14 Newton–Raphson steps; subsequent adjustments only took 1 step. This is consistent with Plass and Stone saying "Because Newton–Raphson iteration converges quickly, only a few steps are needed to find a close approximation to the root. In fact, the algorithm performs well when only one iteration step is used to make the adjustment." Most of the time the curve fitting stage took less than 4 iterations; the worst case was around 10 iterations. Plass and Stone say nothing about how many curve fitting iterations were used. (Figure 2 in their paper shows an example with 100 iterations.)

Selecting the best knots Choosing the regularization parameter τ is delicate. It must be roughly of the same scale as the approximation errors $\sum_{i=1}^{n} \|\gamma(t_i) - p_i\|^2$. These can be very large (in the order of 100) or very small (in the order of 0.01) depending on the order of magnitude of coordinates of the points.

6.2 Data

The original paper contains no usable data or code, as was the norm at the time. Therefore, we had to create our own data. In the spirit of the original paper, we used typographic glyphs for testing. We used three types of data: precise samples from glyph outlines, samples from precise image outlines, and samples from scanned images.
The C and S glyphs in Figures 1 and 4 are from the standard SFNSRounded font in macOS 15.5; that's a TrueType font that uses quadratic curves. The images and outline curves were computed and sampled with libschrift-show [6].
The G glyph in Figure 2 is from the standard TEX Latin Modern font, a PostScript font that uses cubic curves. That glyph is most probably identical to the G glyph used by Plass and Stone (even though the font at the time would have been Computer Modern). The outline was extracted using FreeType and sampled using uniform parametric samples with the number of samples proportional to the segment joining the curve extremes.
The image of the G glyph in Figure 3 was extracted from the original paper as follows. The PDF available at the ACM Digital Library contains scanned pages of size 2550 × 3300 pixels. We cropped the G glyph in page 231 and then reduced the image to 150 x 161 pixels, to follow the spirit of Figure 1a in the original paper. The outline was extracted from the image using potrace-outliner [7].

6.3 Results

The results are presented below. Figure 1 displays the outcomes for a C glyph, which is characterized by its absence of corners. As demonstrated below, the results we achieved are noteworthy, closely approximating the original data. As expected, this simple shape does not require a large number of Bézier curves for effective reconstruction.

The results for the original G, extracted from the original font, are presented in Figure 2. Despite the image having numerous corners, the method demonstrates impressive reconstruction quality. The influence of τ is noteworthy; increasing it results in fewer knots, thereby reducing the reconstruction quality. It is essential that the user balances this trade-off between result quality and the number of knots. Figure 3 illustrates the results for a G extracted from the original paper [1], which is considerably noisier. Nevertheless, the technique successfully achieves good results, reproducing the original curve.

Finally, there is the S glyph, which is characterized by its simplicity and absence of angular discontinuities. When directly extracted from the original dataset, this glyph exhibits a discontinuity in its midsection. However, as demonstrated in Figure 4, the method successfully reconstructs the shape. Notably, the reconstruction was achieved using a minimal number of data points, as illustrated in the examples provided. In Figure 5 we show the results from an outline extracted from an image of the S glyph. As can be seen, the method managed to perform the reconstruction in this setting too.

7 Conclusion

The technique described by Plass and Stone [1] provides an effective means for extracting a cubic spline curve from a sequence of points. Although we have successfully reproduced their results, several implementation details were absent from the original paper. We hope this report fills those gaps and answers any remaining questions.

Acknowledgments

This work is the product of a graduate course on reproducing results in computer graphics. Daniel is a graduate student at IMPA. Davi is an undergraduate student at UFF. Luiz and Luiz Henrique are researchers at IMPA and led the course. This work was done in the Visgraf Computer Graphics laboratory at IMPA in Rio de Janeiro, Brazil. Visgraf is supported by the funding agencies FINEP, CNPq, and FAPERJ, and also by gifts from IBM Brasil, Microsoft, and NVIDIA.

References

1. M. Plass and M. Stone. "Curve-fitting with piecewise parametric cubics." In: **Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques**. SIGGRAPH '83. Detroit, Michigan, USA: Association for Computing Machinery, 1983, pp. 229–239. DOI: 10.1145/800059.801153. URL: https://doi.org/10.1145/800059.801153.
2. D. Perazzo and D. G. N. S. Castro. **Bézier Curve Fitting - Piecewise Parametric Cubics Reproduction**. Code at github.com/danperazzo/plass_reproduction. 2025.
3. U. Ramer. "An iterative procedure for the polygonal approximation of plane curves." In: **Computer Graphics and Image Processing** 1.3 (1972), pp. 244–256. DOI: https://doi.org/10.1016/S0146-664X(72)80017-0. URL: https://www.sciencedirect.com/science/article/pii/S0146664X72800170.
4. D. H. Douglas and T. K. Peucker. "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature." In: **Cartographica** 10.2 (1973), pp. 112–122. DOI: 10.3138/FM57-6770-U75U-7727. URL: https://doi.org/10.3138/FM57-6770-U75U-7727.
5. C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. "Array programming with NumPy." In: **Nature** 585.7825 (2020), pp. 357–362.
6. L. H. de Figueiredo. **libschrift-show – A program to test the features of libschrift**. Code at github.com/lhf/libschrift-show. 2025.
7. L. H. de Figueiredo. **potrace-outliner – A tool for extracting polygonal outlines from bitmaps**. Code at github.com/lhf/potrace-outliner. 2025.

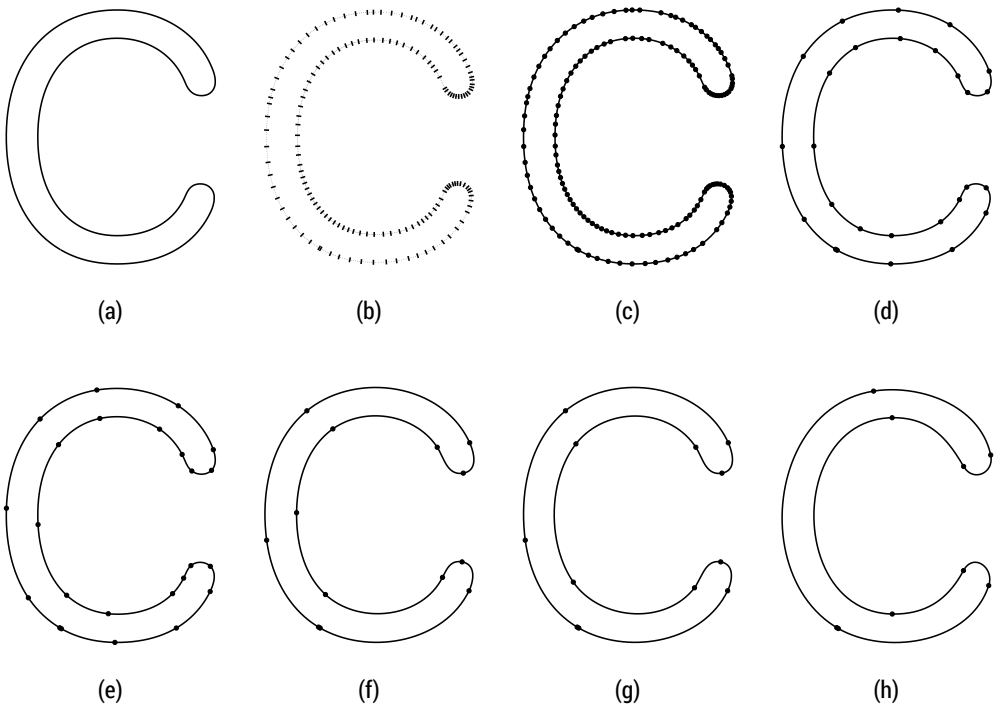

Figure 1. Results using sliding window for tangent computation (5 points before and after each center point, $\varepsilon = 1$). (a) Original data. (b) Potential knots and corners. (c)–(h) Processed curves: (c) $\tau = 0$, (d) $\tau = 4$, (e) $\tau = 20$, (f) $\tau = 320$, (g) $\tau = 3200$, (h) $\tau = 8200$.

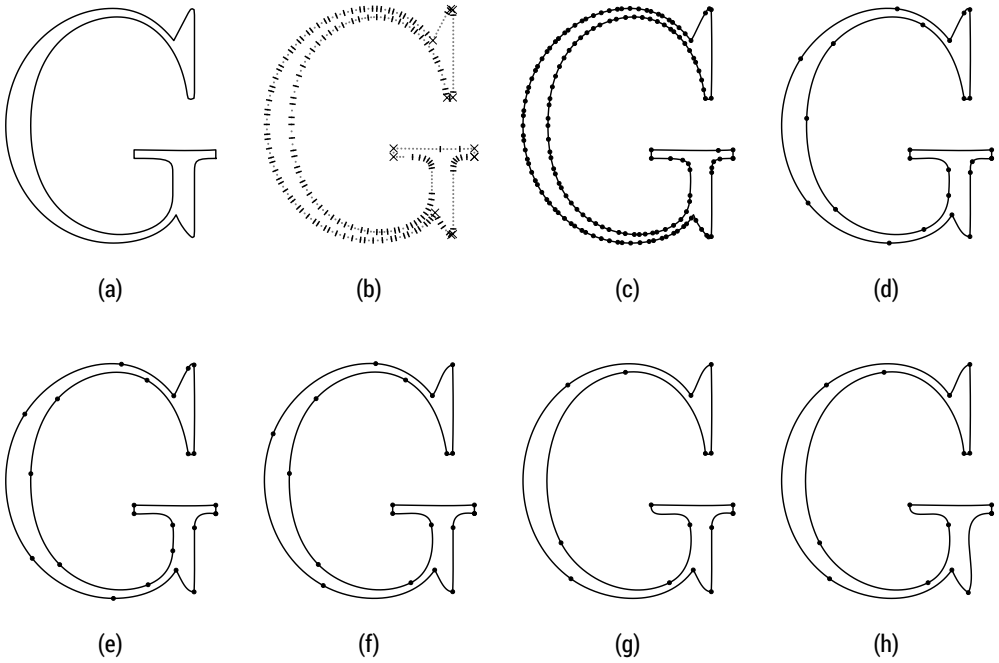

Figure 2. Results using sliding window for tangent computation (5 points before and after each center point, $\varepsilon = 0.25$). (a) Original data. (b) Potential knots and corners. (c)–(h) Processed curves: (c) $\tau = 0$, (d) $\tau = 6$, (e) $\tau = 13$, (f) $\tau = 20$, (g) $\tau = 100$, (h) $\tau = 130$.

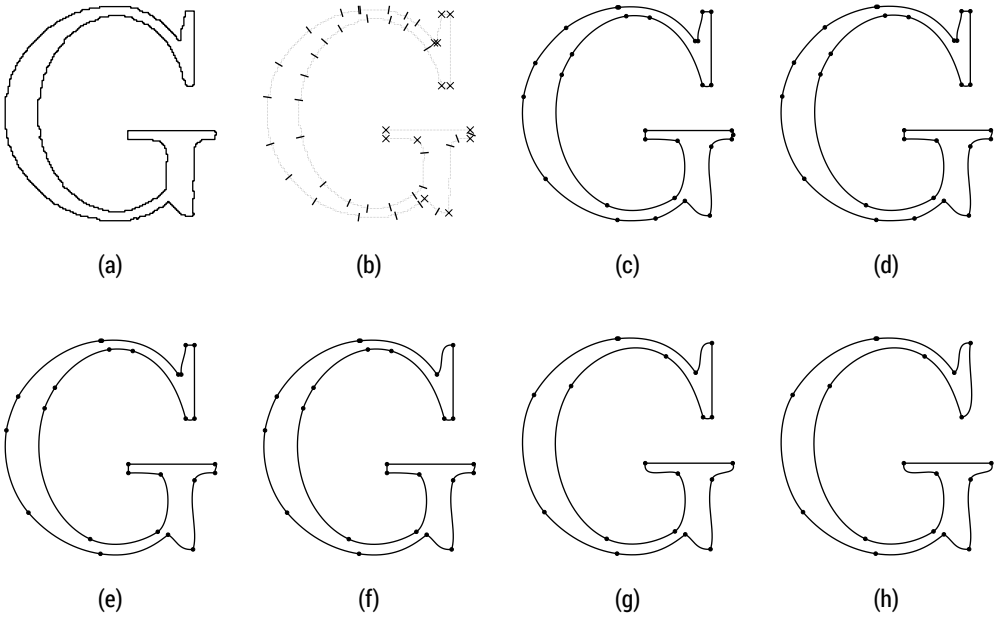

Figure 3. Results using sliding window for tangent computation (25 points before and after each center point, $\varepsilon = 1.75$). (a) Original data. (b) Potential knots and corners. (c)–(h) Processed curves: (c) $\tau = 0$, (d) $\tau = 0.5$, (e) $\tau = 5$, (f) $\tau = 7$, (g) $\tau = 76$, (h) $\tau = 100$.

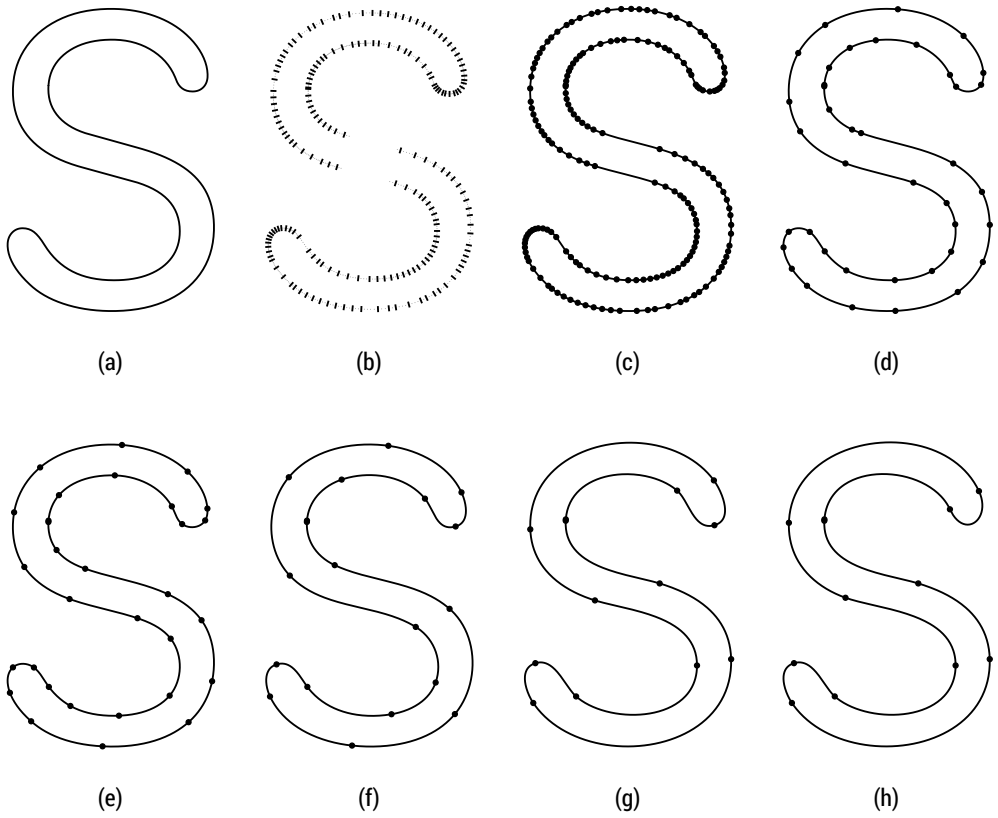

Figure 4. Results using sliding window for tangent computation (5 points before and after each center point, $\varepsilon = 1$). (a) Original data. (b) Potential knots and corners. (c)–(h) Processed curves: (c) $\tau = 0$, (d) $\tau = 1$, (e) $\tau = 10$, (f) $\tau = 160$, (g) $\tau = 2800$, (h) $\tau = 8500$.

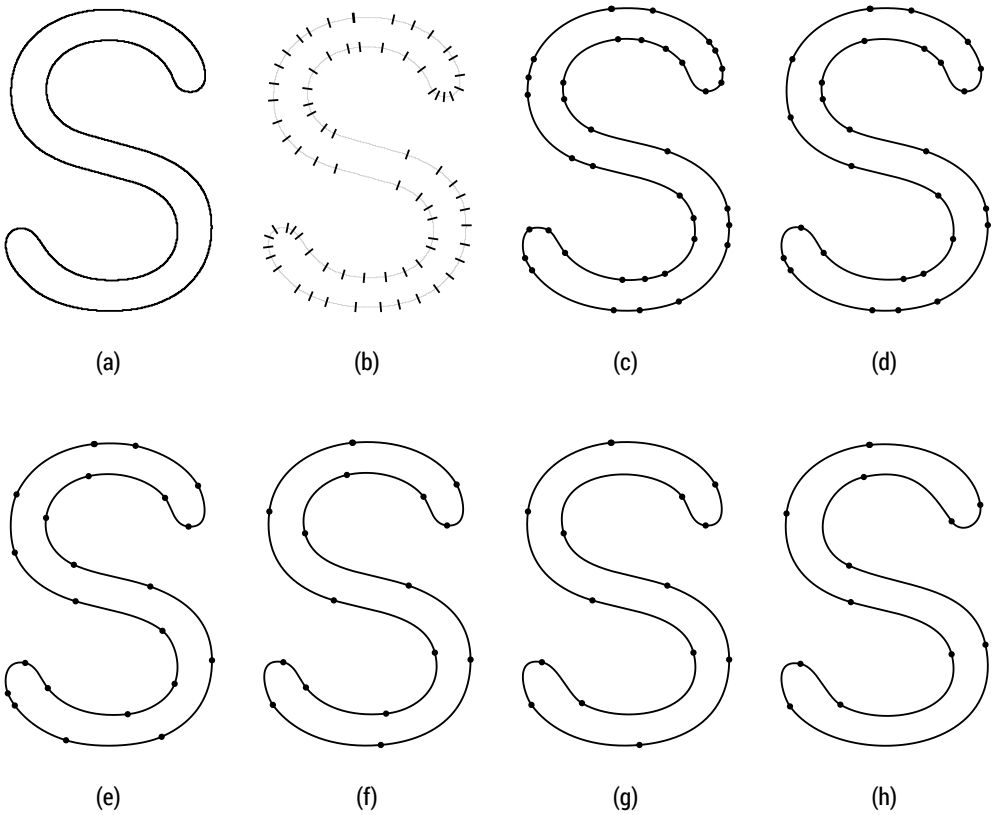

Figure 5. Results using sliding window for tangent computation (15 points before and after each center point, $\varepsilon = 1.5$). (a) Original data. (b) Potential knots and corners. (c)–(h) Processed curves: (c) $\tau = 0$, (d) $\tau = 10$, (e) $\tau = 50$, (f) $\tau = 210$, (g) $\tau = 450$, (h) $\tau = 2500$.