Learning a Manifold of Fonts

Neill D.F. Campbell[1] Jan Kautz[1,2]
[1]University College London [2]NVIDIA Research

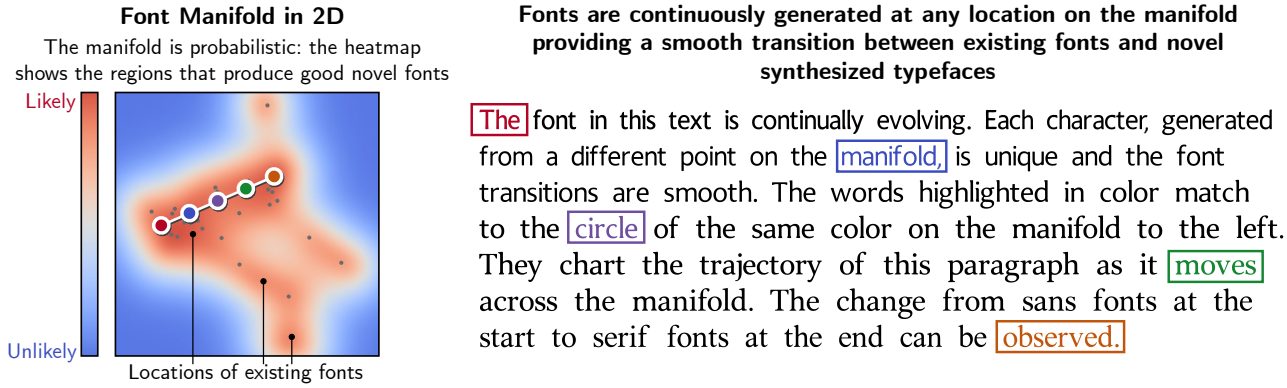


Figure 1: *The manifold of fonts. On the left, we show a 2D manifold learnt from 46 fonts. Every point in the manifold corresponds to a complete font; as you move across the manifold the corresponding font smoothly changes by interpolating and extrapolating the the original training fonts. We demonstrate this effect with the text on the right; each character is created from a different 2D location in the manifold that is obtained by moving along the straight line shown on the left. The colored dots match up with the colored words. The heatmap of the manifold is indicative of the likelihood of a location containing a good font. In addition to the results presented in this paper, we provide a standalone Javascript based viewer that allows users to explore both the joint manifold of fonts and manifolds for individual characters.*

Abstract

The design and manipulation of typefaces and fonts is an area requiring substantial expertise; it can take many years of study to become a proficient typographer. At the same time, the use of typefaces is ubiquitous; there are many users who, while not experts, would like to be more involved in tweaking or changing existing fonts without suffering the learning curve of professional typography packages.

Given the wealth of fonts that are available today, we would like to exploit the expertise used to produce these fonts, and to enable everyday users to create, explore, and edit fonts. To this end, we build a generative manifold of standard fonts. Every location on the manifold corresponds to a unique and novel typeface, and is obtained by learning a non-linear mapping that intelligently interpolates and extrapolates existing fonts. Using the manifold, we can smoothly interpolate and move between existing fonts. We can also use the manifold as a constraint that makes a variety of new applications possible. For instance, when editing a single character, we can update all the other glyphs in a font simultaneously to keep them compatible with our changes.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; I.2.6 [Artificial Intelligence]: Learning;

Keywords: digital typography, shape matching, modeling

Links: DL PDF

ACM Reference Format
Campbell, N., Kautz, J. 2014. Learning a Manifold of Fonts. ACM Trans. Graph. 33, 4, Article 91 (July 2014), 11 pages. DOI = 10.1145/2601097.2601212 http://doi.acm.org/10.1145/2601097.2601212.

0730-0301/14/07-ART91
DOI: http://dx.doi.org/10.1145/2601097.2601212

1 Introduction

Everyone has access to a large number of fonts; they are already installed with our operating systems, provided by software packages, or may be obtained from online repositories; for example, hundreds are available on the 'Google Web Fonts' project[1]. These fonts deliver a wide range of typefaces, each of which is the result of many hours of design by professional typographers, and each of whom has many years of training and experience, not to mention their artistry. Thus, a collection of fonts comprises a huge body of information about the appearance and style of type.

Many users of type are interested in its appearance and manipulation but do not have the training, or time, to make use of professional editing tools. The steep learning curves of font packages act as a barrier to more users exploring and manipulating fonts and creating new typefaces. We are motivated by a desire to give users who have no formal training in typography the ability to explore and manipulate fonts and to create new typefaces.

Starting from a different view, consider the set of all possible closed curves. Clearly, only a very small fraction of these curves are plausible glyphs that could represent a character. While it might take considerable expertise to start from a set of arbitrary curves and build a typeface, we already have access to many examples of valid glyphs. From a machine learning perspective, the set of typefaces lies in a low dimensional manifold within the space of all possible curves. Furthermore, we have access to a large number of training examples (fonts) that are known to lie within this manifold. Thus, identifying this manifold is an unsupervised learning task that, when completed, condenses the knowledge and expertise of many typographers into a single space. From within this space we can not only recover the original fonts, but also generate a continuum of novel fonts since every location in the space can generate a complete typeface.

In this paper we present a framework to perform this learning task and identify a manifold for fonts. We take, as input, a collection

[1] http://www.google.com/fonts

of existing font files and create a low dimensional space such that every location in this space generates a novel typeface through the interpolation and extrapolation of these fonts. To the best of our knowledge, this is the first work to learn a fully generative model for typefaces in a completely unsupervised manner; one that uses only existing fonts as input. This is advantageous since, as we have already discussed, there are a large number of fonts already available and our learning process is fully automatic; it requires neither user intervention nor input from a professional typographer.

We acknowledge that our learning based approach in no way replaces the role of a typographer. Since it is an unsupervised process, we do not follow a formal process of identifying individual characteristics or components (*e.g*. stroke axes and contrast). Instead we make use of this information by proxy, through the existing font designs, and summarize it into a form that can be exploited by untrained users; our learning to generate good fonts is dependent on well designed typefaces.

The manifold may also be of use to experienced type designers. It allows for exploration of existing fonts and can be used to identify regions that are missing typefaces. A generated font can always be used as a starting point for subsequent editing. Also, the smooth interpolation between fonts can be used as an artistic tool to create visual designs that would be time consuming to produce otherwise (*e.g*. creating a series of intermediate typefaces between two typefaces or even automatically producing different weights of a single typeface between two extrema).

Our Contributions Our paper makes contributions in terms of both its technical approach and its novel applications. On the technical side, we present a new energy model specifically designed to provide dense correspondences between character outlines across multiple fonts. In addition, we describe a representation scheme and coarse-to-fine approach that allows for high quality optimization results to be obtained for our model. We demonstrate that the results of our optimization can be used with a powerful non-linear dimensionality reduction technique to produce a low dimensional manifold that captures the variation within existing fonts; this allows it to be used as a generative model for new typefaces.

Once the manifold has been obtained, a number of interesting applications are made possible. We demonstrate the ability to smoothly interpolate between fonts at a character by character level as an exciting artistic tool. Non-expert users are provided with the ability to manipulate an entire font by just modifying a single character; their edits can be automatically propagated to maintain visual consistency across the font. The manifold may also be explored directly to discover new typefaces and we include an interactive tool (a standalone Javascript based browser) that allows readers to explore two dimensional versions of both joint manifolds (all characters) and those for individual characters.

2 Previous Work

In the following we discuss the most relevant previous work. Note that commonly 'typeface' refers to the design of a type, *e.g*. Helvetica, and a 'font' refers to a specific instantiation, *e.g*. Helvetica Semi-Bold Italic. Similarly, a 'glyph' refers to the specific design of a 'character'. However, both of these pairs of terms are often used interchangeably, and hence we do not differentiate strictly between them in this paper.

Parametric Font Design As we have discussed, designing new fonts from scratch requires professional skills and is a very time-consuming process. Not surprisingly, there exists a comprehensive body of research to simplify this process. Parameterizing fonts is one method of allowing a user to create novel fonts by just adjusting a few parameters. The well-known Metafont system by Knuth [1986] supports parametric fonts, and was used to create most of the Computer Modern typeface family through parameterization. Shamir and Rappoport [1998] proposed a system, where fonts are represented through high-level parametric features (*e.g*. serifs, arcs and joints), and constraints on those features. This enables the user to modify a glyph's feature, such as the width of a stem, and it be propagated to all other glyphs. However, the feature and constraint extraction is not fully automatic, and requires user assistance. The approach by Hu and Hersch [2001] extends this idea with parameterizable shape components, that can be assembled to form glyphs. Once these assemblies are defined, global parameters can create many variants of a font. However, creating these assemblies has to be done for a particular typeface and possibly even for a specific subset [Hassan et al. 2010]. Lau [2009] proposes to learn a parametric model with constraints from examples. However, the parametric model is simple and allows the user to only adjust parameters like its weight or width.

There are industry standards for specifying fonts with more advanced interpolation between different glyph shapes, including 'OpenType GX' and 'Adobe Multiple Masters' fonts [Adobe Systems 1997]. In the case of the latter, these are designed such that each glyph has a set of master outlines where each bézier curve control point is in exact correspondence; these are the multiple masters for each glyph. A particular instance is then generated by specifying a weight vector (a linear convex combination) to apply to each control point to generate the final outline. Our work is similar in philosophy except that, instead of relying on a complex design process where every single font glyph has to be created with an identical parameterization (the correspondences between control points), we take a large number of existing fonts and automatically find the correspondences.

Data-Driven Font Synthesis and Hinting Suveeranont and Igarashi [2010] automatically derive a skeleton and an outline for each letter of a set of fonts. New fonts can then be created through a feature-preserving, weighted blend of skeletons and outlines. However, most results exhibit slight but quite noticeable distortions within glyphs.

Hersch and Betrisey [1991] and Zongker et al. [2000] aim to (semi)-automatically translate hints (*i.e*., the rules on how to render a character at low resolutions, such as on-screen) to a font without hints, which an artist can then tweak to perfection. The former technique requires a model font to be manually created, while the latter can use any font that has hints as a reference. While their goal is different from ours, they also find correspondences between the outlines of glyphs. More accurately, they match existing control points on the contour curves. This assumes that the existing knots describe similar locations, and hence only very similar fonts can be matched. Furthermore, the resulting matches are also very sparse, and would not be sufficient for interpolating glyphs.

Zitnick [2013] recently automated hand-writing beautification from stylus input, by essentially averaging multiple instances of the same strokes. To this end, input strokes need to be matched. As demonstrated, curvature-based sampling of the input strokes enables reliable matching. Unfortunately, this matching method relies on the temporal input strokes, which do not exist in our case.

Browsing and Structuring Fonts Almost all commonly used applications involving type, such as word processors, let a user access fonts by browsing their names, which is not very intuitive. Structuring fonts by similarity and creating an embedding in which the user can more intuitively navigate would be a useful tool, as proposed by Loviscach [2010]. Note that our manifold serves a different purpose. We do not use the manifold to let the user browse for fonts, rather it is a tool to enable the synthesis of new fonts.

Shape Interpolation The matching and interpolating of 2D and 3D shapes has been studied extensively [Alexa et al. 2000; Kazhdan et al. 2004; van Kaick et al. 2011]. Generally speaking, these techniques will have difficulties accurately matching and interpolating glyphs, unless a font-specific regularization term is used, due to the very different features that the same glyph from different fonts may exhibit (*e.g.* the letter 'i' with and without serifs). Furthermore, the pairwise matching of glyphs is not sufficient to build a generative manifold of fonts, since the creating of an appropriate embedding requires a single consistent representation.

Very recently, methods have appeared [Kalogerakis et al. 2012; Kim et al. 2013] that analyze large collections of 3D shapes, from which new shapes are synthesized by assembling plausible combinations of shape components either through templates or probabilistically. In some ways, we are addressing a 2D version of that problem. However, we do not decompose our shapes, *i.e.* the glyphs, into components. Instead we directly match and interpolate the outlines of the glyphs.

3 Character Matching

The font manifold is built in two stages. We begin by matching each individual character (glyph) across all fonts using an energy based optimization procedure with a coarse-to-fine approach; this is our primary contribution. We then use the dense correspondences for each character as a basis to perform the second stage of fitting a generative, non-linear manifold; this ties together the different characters into a single space.

In this section, we begin with a brief discussion about paramterization and then describe the matching algorithm. We provide further details on the manifold in the subsequent section.

3.1 Universal Parameterization

In order to create novel fonts using a generative manifold model, we must be able to express every font in a universal parameterization (UP). That is, we should be able to construct a high dimensional vector that both contains all the information necessary to draw each character, and, has its elements in correspondence between different fonts. This means that small changes in the generative space will result in smooth changes in the appearance of the font. The 'Adobe Multiple Masters' fonts [Adobe Systems 1997] makes use of such a parameterization since each of the masters for a single glyph is laboriously designed with each bézier curve control point in correspondence.

The ordinary fonts we use are not designed with such a parameterization. Instead, we must find a UP automatically by reparameterizing each of the characters across all the fonts; we use a polyline representation to achieve this. This is universal since every vertex in the polyline is consistent (in correspondence) across all the fonts for a given glyph; we assume consistent topology between glyphs. Furthermore, our matching algorithm can reparameterize glyph outlines for all fonts *simultaneously*. Simultaneous matching is a key benefit of our approach. If a pairwise matching technique were used, two additional challenges would have to be overcome. Firstly, there is a quadratic scaling in cost for matching all possible pairwise combinations. Secondly, there is the problem of consistency when converting the pairwise matches to a UP; namely that matching around a loop of fonts does not lead to the identity, *i.e.* $A \rightarrow B \rightarrow C \rightarrow A \neq I$.

Topology Using outlines limits our approach to glyphs with consistent topology across fonts (a one-to-one correspondence between the curves). This is not usually a problem for standard fonts except for cases with an alternate form exists: *e.g.* 'g' has an italic form 'g'. In such cases, we treat the two variants as independent glyphs; we note that an interim state between the two is never used and would not generate a reasonable character.

3.2 Outline Preparation

Each character is matched individually across all the different fonts. We describe the process for a single character. Since the matching is independent between characters, this matching process may be performed in parallel. We used a dataset of 46 standard fonts; we provide details of the specific fonts used in supplement § A.

Outline Extraction We extract the outlines of each character from a truetype file using a standard library.[2] Once we have obtained the raw Bézier curves for each outline, for example the the character 'o' has an inner and outer outline, we densely sample N points around each closed curve to produce a polyline representation.

At this stage, we also extract the metrics from the font file (for example, 'horizontal advance' and 'vertical linegap' distances) that will be used later.

Normalization Given the polylines, a normalization process is performed to ensure that the different glyphs from each font are at the same size so that the curvature values are comparable. We subtract the mean from the vertices and rescale them to have a unit variance; the offset and scale of this transformation is saved and is used later, in the manifold, to restore the glyphs true size and aspect ratio.

Initial Alignment Some characters have multiple outlines that are stored inconsistently in different font files; we must identify the correct permutation to, for example, match the outside of one font to the outside of another. We find the outline correspondence between two glyphs by evaluating the minimal squared distance between the sampled vertices for all permutations, and picking the one with the lowest score; this is inexpensive since there are very few outlines per glyph. We also use this method to align the 'starting points' on the polylines. Let $\{\mathbf{u}_i\}$ and $\{\mathbf{v}_j\}$, for $i, j \in [1..N]$, as two outlines. The distance matrix D is specified as

$$D_{i,j} = ||\,\mathbf{u}_i - \mathbf{v}_j||^2 \tag{1}$$

and suming along the diagonals gives

$$\Delta_n = \sum_{(i-j) \bmod N \equiv n} D_{i,j} \quad . \tag{2}$$

The minimal value $\Delta_n^* = \min_n \Delta_n$ gives the matching cost between the outlines and the value of n that achieves the minimum is the phase shift that aligns the starting points.

This alignment is performed pairwise but we need a global initialization across all the different fonts. We use the Δ_n^* score between pairs of fonts to build a minimum spanning tree and propagate the pairwise phase shifts; afterwards, the outlines are roughly aligned over all the fonts (a suitable initialization for the optimization).

3.3 Energy Optimization

After the preparation stage we have a set of polylines (containing N samples) corresponding to the same outline over M different fonts. For characters with multiple outlines, each outline is processed independently so we will assume a single outline for clarity. The outline may be represented as a sampled, closed, parametric curve

$$\mathbf{u}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} \tag{3}$$

[2] We used the 'stb ttf' public domain library (`http://nothings.org`)

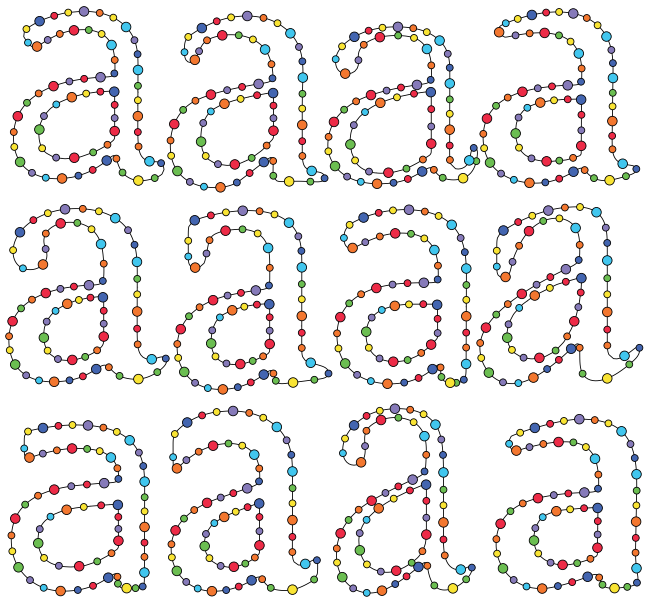


Figure 2: *Dense correspondence results for the character 'a'. We show a representative sample from across 46 fonts used. The colored dots (subsampled from the actual polyline) on the outline indicate corresponding points after simultaneous optimization. We can see that corners and concavities are well matched, despite the variation in the shape, and that the outlines are all evenly parameterized.*

where the parameter $t \in [0, 1]$. We denote the sampled set of outlines as $\{\mathbf{u}_{i,m}\}$ where

$$\mathbf{u}_{i,m} = \mathbf{u}_{(m)}(t_{i,m}) \tag{4}$$

is the outline for the m^{th} font evaluated at sample t_i, $i \in [1, N]$. Initially, all the outlines are uniformly sampled such that the difference $(t_{(i+1),m} - t_{i,m})$ is constant. Our goal is to update all of these parameters such that for a given i, the points $\mathbf{u}_{i,m}$ across the fonts $m = [1, M]$ are in shape correspondence. This is equivalent to reparameterizing each outline into a common universal parameterization. Figure 2 illustrates this parameterization by showing the corresponding samples as dots with the same color across outlines from different fonts.

We match the shapes of the outlines using both curvature and normal information. The curvature is defined as

$$\kappa(t) = \frac{x'y'' - y'x''}{(x')^2 + (y')^2} \tag{5}$$

where $x' = {}^{\mathrm{d}}/_{\mathrm{d}t}\,(x(t))$, $x'' = {}^{\mathrm{d}^2}/_{\mathrm{d}t^2}\,(x(t))$ and similarly for y' and y''. We also have the curve normal as

$$\eta(t) = \frac{1}{\left((x')^2 + (y')^2\right)^{\frac{1}{2}}} \begin{bmatrix} x' \\ y' \end{bmatrix} \quad . \tag{6}$$

To simplify notation we will use $\kappa_{i,m}$ to denote the curvature of curve $\mathbf{u}_m(t)$ evaluated at parameter $t_{i,m}$. Similarly, we use $\eta_{i,m}$ for the normal.

The Energy Model Our energy model is a weighted combination of terms involving the encouraging consistent curvature and normal information across fonts as well as an elastic regularization term with a reparameterization constraint. Formally, the energy $E(\mathbf{t})$, with with $\mathbf{t} = [\cdots\ t_{i,m}\ \cdots]^{\mathrm{T}}$, is given by

$$E(\mathbf{t}) = E_\kappa(\mathbf{t}) + \lambda_{\text{el}} E_{\text{el}}(\mathbf{t}) + \lambda_\eta E_\eta^{\text{up}}(\mathbf{t}) + \lambda_\eta E_\eta^{\text{down}}(\mathbf{t}) \tag{7}$$

along with the constraint $A\,\mathbf{t} - \mathbf{b} \geq \mathbf{0}$. We will now describe these terms in our energy function.

Curvature Variation When all the outlines are in correct correspondence, the variance in curvature across the different fonts should be small. That is, sharp concavities and convexities should occur at the same locations and regions of low curvature should coincide. The first term in our energy function seeks to minimize this curvature variance. However, at sharp corners, the curvature tends to infinity which will dominate the cost function; to avoid this, we remap the curvature through an exponential function. This resulting energy is

$$E_\kappa(\{t_{i,m}\}) = \sum_{i=1}^{N} \left[1 - \exp\left(-\frac{1}{2M}\gamma \sum_{m=1}^{M} \left(\kappa_{i,m} - \bar{\kappa_i}\right)^2\right)\right] \tag{8}$$

where

$$\bar{\kappa_i} = \frac{1}{M} \sum_{j=1}^{M} \kappa_{i,j} \tag{9}$$

is the mean curvature across all fonts at sample i and γ is a constant parameter.

Elasticity Regularisation If we minimize the curvature variation alone, all the samples will simply move to an area of low curvature and not accurately represent the outline. To prevent this from occurring, we include an elastic regularization term that encourages the samples to spread out around the whole of the outline while allowing them to stretch or contract when necessary to follow the curvature. The elasticity term is

$$E_{\text{el}}(\{t_{i,m}\}) = \sum_{i=1}^{N} \sum_{m=1}^{M} \left[\left(t_{(i+1) \bmod N,m} - t_{i,m}\right) - \frac{1}{N}\right]^2 \tag{10}$$

where the mod N accounts for the wrap around of the closed curve.

Monotonic Constraint To be a valid reparameterization the mapping must be monotonic, that is the samples across the outline of a given font must maintain the same ordering. We therefore need

$$\left(t_{(i+1) \bmod N,m} - t_{i,m}\right) \geq 0 \quad \forall\, i, m \quad . \tag{11}$$

We can enforce this as a linear constraint using a sparse matrix A and vector $\mathbf{b}$

$$A\,\mathbf{t} - \mathbf{b} \geq \mathbf{0} \quad . \tag{12}$$

We note that the same matrix and vector may be used to encode equation (10) as

$$E_{\text{el}}(\{t_{i,m}\}) = \left\| A\,\mathbf{t} - \mathbf{b} - \frac{1}{N}\mathbf{1} \right\|^2 \quad . \tag{13}$$

Normal Matching When substantial stretching or compression of the correspondences is required, the elasticity regularization prevents desirable matching. This occurs when matching the extreme cases of serif and sans serif fonts, as demonstrated in Figure 3. To cope with this situation, we add an energy term that minimizes the normal variance locally in the regions where serifs occur for vertical and horizontal normals. The normal energy

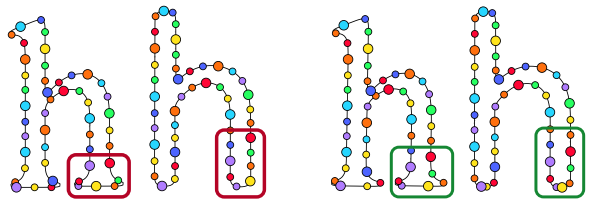
$$E_\eta(\{t_{i,m}\}) = \sum_{i=1}^{N} \sum_{m=1}^{M} \rho(\mathbf{u}_{i,m}) \left\| \Phi_{i,m} - \bar{\Phi}_i \right\|^2 \tag{14}$$

is a summation over two terms. The first term

$$\rho(\mathbf{u}_{i,m}) = \exp\left(-(\mathbf{u}_{i,m} - \mathbf{q})^2\right) \tag{15}$$

is non-zero where the outline is near $\mathbf{q}$; this is chosen to select either the top or the bottom of the character in the normalized outline. The second term

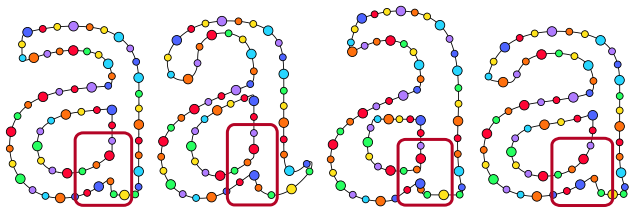
$$\Phi_{i,m} = \exp\left(-\beta\left(\eta_{i,m} \cdot \mathbf{r} - 1\right)\right) \tag{16}$$



Without the normal term: the serif has incorrect matches on both of the legs

With the normal term: the elasticity is relaxed, allowing the legs to match

Figure 3: *The effect of the normal matching energy term. With some serifs there can be a dramatic stretching or compression of the corresponding samples that is too expensive under the elastic term of equation (10). This is the situation on the left matching the font 'Calibri' to 'Times New Roman'. The addition of the normal matching term, equation (14), allows the elastic term to be relaxed locally at the serif and the terminal is matched correctly.*



Without the coarse-to-fine approach: an incorrect local minimum is found where the concavities are out of correspondence

Figure 4: *Direct minimization of our cost function is prone to fall into local minima that results in poor correspondences. Here we show the correspondences between the fonts 'Tahoma', 'Times New Roman', 'Narkisim', and 'Miriam'. The highlighted regions show that the two sharp concavities are not in correspondence. Please compare to Figure 5 where we show results using our coarse-to-fine approach.*

is non-zero when the normal points in the direction of $\mathbf{r}$; this is set to point upwards at the top and downwards at the bottom. We then use a quadratic cost against the mean

$$\bar{\Phi}_i = \frac{1}{M} \sum_{m=1}^{M} \Phi_{i,m} \tag{17}$$

over the different fonts to encourage consistency; *e.g.* all the upwards normals at the top of the letter should occur at the same parameterizations. The constants β, $\mathbf{r}$ and $\mathbf{q}$ are chosen to localize the effect to vertical normals pointing upwards at the top of the outline and downwards at the bottom (two terms are used).

Optimization The gradients with respect to $\{t_{i,m}\}$ may be calculated for all the terms (and the constraints) and may be optimized using standard constrained non-linear optimization methods. We used the 'fmincon' solver in 'Matlab' using the interior-point algorithm and LBFGS hessian updates.

3.4 Coarse-to-Fine Approach

While our cost function encodes the desirable properties for good correspondences, the energy is prone to local minima, due, in particular, to the monotonic constraints. Figure 4 shows an example of this when matching four different fonts. We overcome this problem by using a Fourier representation of the curve and running the optimization in a coarse-to-fine approach. We observe that the low frequency (smoothed) representation of characters is very similar and use this to guide our optimizations to good minima.

Elliptical Fourier Representation Elliptic Fourier descriptors [Kuhl and Giardina 1982] express a closed curve as a summation of elliptic harmonics (in a similar manner to a standard Fourier series). Using the notation from equation (3), we can express the closed outline as a periodic function

$$x(t) = a_0 + \sum_{k=1}^{\infty} \left[a_k \cos\left(\frac{2k\pi t}{T}\right) + c_k \sin\left(\frac{2k\pi t}{T}\right) \right] \tag{18}$$

where T is the perimeter of the contour, and $t \in [0, 1]$ is our standard parameter; $y(t)$ is defined in a similar manner. Determining the coefficients $\{a_k, c_k\}$ is a straight-forward procedure and we refer to its treatment in [Kuhl and Giardina 1982] and [Prisacariu and Reid 2011].

Since it is impossible to hold an infinite number of coefficients, we must truncate our representation at some frequency and only store the first coefficients. If we store too few coefficients then we lose all the high frequency detail in the shape. While this is bad for shape recovery, it is useful to us since character outlines have very similar shapes at low frequency; this is true even if the full shapes (including the high frequency components) are quite different. We take advantage of this to guide our optimization to a good minima by first performing the minimization using only the low frequency components (we use a Gaussian filter). We then use the result of this optimization as the initial starting point for a subsequent optimization that incorporates more coefficients (higher frequencies). By gradually incorporating the high frequency components in a coarse-to-fine strategy we guide the optimization to better minima. Figure 5 demonstrates this procedure and we can see the benefit of the approach by comparing the results in the last column to Figure 4. Figure 2 shows some more results for a sample of the other fonts; these are all matched at once.

4 Finding the Manifold

As a result of the character matching in the previous section, we have found a consistent parameterization for each character across all fonts. We no longer need the elliptical Fourier components and instead represent each outline as a set of $i \in [1, N]$ samples $\{\mathbf{u}_{i,m}^{(h)}\}$ for each font $m \in [1, M]$ and now character $h \in [1, H]$. We also know, that each point sample i will be in correspondence across the different fonts.

While a collection of arbitrary polylines contains two degrees-of-freedom per vertex, the fact that our set of polylines (now matched to lie in correspondence) represent glyphs means that they should contain significant redundancy; the true degrees-of-freedom will be far fewer and depend on the typography rules used to design the fonts. To this end, we learn a low dimensional space (representing the few true degrees-of-freedom) and a mapping that generates the high dimensional set polylines that make up a font. We begin with a brief introduction to the generative model we use, the Gaussian Process Latent Variable Model (GP-LVM), before providing details about the font manifold. We provide a more extensive discussion, offering greater insight into the model of the GP-LVM, in supplement § B.

4.1 The GP-LVM

Gaussian Processes (GPs) [Rasmussen and Williams 2006] are non-parametric models that consist of distributions over functions. They can be used as a prior over functions to encourage smoothness. They have the property that for a function taken from a GP, any discrete set of draws under the function will be normally distributed. Thus if the function is evaluated at a series of values $X = [\cdots \mathbf{x}_j \cdots]^{\mathrm{T}}$ from a multidimensional input space $\mathbf{x}_j \in \mathcal{R}^{\mathcal{Q}}$, the corresponding

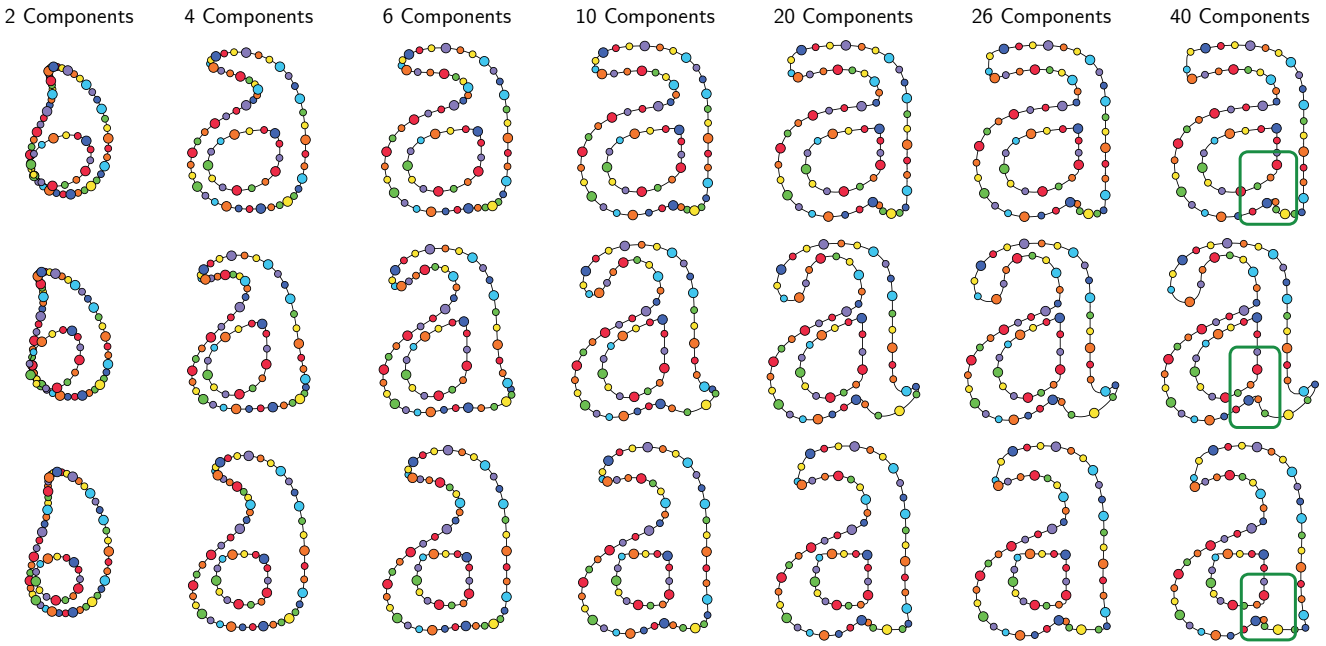


Figure 5: *The minimization results combining our cost function with an elliptical Fourier representation in a coarse-to-fine approach. We show correspondences between the fonts 'Tahoma', 'Times New Roman' and 'Narkisim'. The optimization proceeds from left to right; at each stage we use the result of the previous optimization to initialize the next while increasing the cut-off of the low pass filter over the Fourier components. Even though the final outlines are quite different, at low numbers of components we can see that the shapes are very similar and use this to guide the optimization to good minima. Please see the improvement over the results in Figure 4. The results are found to be in good correspondence with the other font examples given in Figure 2.*

output values $Y = [\cdots \mathbf{y}_j \cdots]^{\mathrm{T}}$ in a multidimensional output space $\mathbf{y}_j \in \mathcal{R}^D$ will be distributed as

$$P(Y|X) = \mathcal{N}(Y|M(X), C(X, X|\theta)) \tag{19}$$

where $M(X)$ is some mean function and $C(X, X|\theta)$ is a covariance function that creates a positive semi-definite covariance matrix. If we assume that our data has been normalized to be zero mean then we can neglect the mean function as $M(X) = \mathbf{0}$. We have used the notation $X = [\cdots \mathbf{x}_j \cdots]^{\mathrm{T}}$ and $Y = [\cdots \mathbf{y}_j \cdots]^{\mathrm{T}}$ to represent a collection of vectors

GP-LVM The Gaussian Process Latent Variable Model (GP-LVM) [Lawrence 2005] is a powerful non-linear, dimensionality reduction technique that produces a probabilistic, generative model of a high dimensional dataset Y with a set of low dimensional 'latent' variables X. Note that the parameters are 'reversed' with respect to the GP; the GP-LVM in effect is a a function mapping from the low dimensional manifold (latent) space to a high dimensional output space, so we have $Q \ll D$.

The Covariance Function The covariance function defines the mapping between X and Y. The function must accept one or more samples from the input space as each argument and return a covariance value between each pairwise combination of samples. The function can depend on a small number of 'hyperparameters' that we shall denote with θ. The elements a particular covariance matrix, given by a set of inputs to the function, are indexed using the notation for representing a set of vectors described above. Thus

$$\left[C\left([\cdots \mathbf{x}_j \cdots]^{\mathrm{T}}, [\cdots \mathbf{x}_j \cdots]^{\mathrm{T}} \middle| \theta\right)\right]_{i,j} = c\left(\mathbf{x}_i, \mathbf{x}_j \middle| \theta\right) \tag{20}$$

where $[\cdot]_{i,j}$ denotes the element at the i^{th} row and j^{th} column of the matrix and $c(\mathbf{x}_i, \mathbf{x}_j|\theta)$ denotes the covariance between two input vectors.

We use the radial basis function (RBF)

$$c\left(\mathbf{x}_i, \mathbf{x}_j \middle| \theta\right) = \alpha \exp\left(-\frac{1}{2}\psi \left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2\right) \tag{21}$$

where α and ψ are the hyperparameters and therefore $\theta = [\alpha, \psi]$. We can see that this covariance function will produce a smooth mapping from the input space to the output space since nearby input vectors will have a high covariance and therefore the output vectors will be correlated. Similarly, if the input vectors are far apart (relative to the length-scale parameter ψ) then there will be a low covariance and the corresponding output vectors will be independent of one another.

Training The training process for the GP-LVM considers the likelihood of the high dimensional data Y as

$$P(Y|X, \theta) = \prod_{m=1}^{M} \mathcal{N}\left(\mathbf{y}_m \mid \mathbf{0}, C(X, X|\theta) + \sigma^2 I\right) \tag{22}$$

where there are M training examples (one for each font) and I is the identity matrix. We maximize this likelihood jointly over the latent vectors $X = [\cdots \mathbf{x}_j \cdots]^{\mathrm{T}}$ as well as the hyperparameters θ such that

$$X^*, \theta^* = \arg\max_{X,\theta} \log\left[P(Y|X, \theta)\right] \quad . \tag{23}$$

The σ value in (22) is a noise variance that accounts for any mismatch between the generated high dimensional vectors and the observed training data. In our experiments, the noise variation was found to be very small, suggesting that the font outlines do indeed lie on a low dimensional manifold.

The covariance function is a complex, non-linear function, both in terms of its relationship to X and to θ; this means that the optimization cannot be performed analytically. However, gradients may be found with respect to both so training may be performed using Conjugate Gradients [Lawrence 2005]. Such methods require initial values for both X and θ. As is standard, [Lawrence 2005], we

initialize the latent values with a linear PCA reduction of the high dimensional vectors Y and set the hyperparameters to conservative values with a wide uninformative prior ($log(\theta) \sim \mathcal{N}(\mathbf{0}, I)$, initialized with the mean).

4.2 Font Manifold

The GP-LVM model is well suited to learning a manifold of fonts since we are starting with a very high dimensional space. If we have $(2N)$ points for each outline (some characters have multiple outlines) and H characters we will have the high dimensional space as dimension $D \geq (2NH)$. In our experiments we used $N = 512$ and when using all the characters (lower- and upper-case) and the digits we have $H = 62$. Previous work has demonstrated that the non-linear embeddings produced by the GP-LVM require orders of magnitude fewer latent dimensions than linear models such as Principal Component Analysis (PCA); see [Prisacariu and Reid 2011] for an example.

Training the GP-LVM We concatenate all of our outline samples $\{\mathbf{u}_{i,m}^{(h)}\}$ to generate M high dimensional vectors $\mathbf{u}_m$, one for each font, as

$$\mathbf{u}_m = \begin{bmatrix} \begin{bmatrix} \mathbf{u}_{1,m}^{(1)} \\ \mathbf{u}_{2,m}^{(1)} \\ \vdots \end{bmatrix}^{\mathrm{T}} & \begin{bmatrix} \mathbf{u}_{1,m}^{(2)} \\ \mathbf{u}_{2,m}^{(2)} \\ \vdots \end{bmatrix}^{\mathrm{T}} & \dots \end{bmatrix}^{\mathrm{T}} \tag{24}$$

and we subtract off the mean $\bar{\mathbf{u}}$ to produce the $\mathbf{y}_m$ vectors as

$$\mathbf{y}_m = \mathbf{u}_m - \bar{\mathbf{u}}\,, \quad \bar{\mathbf{u}} = \mathbb{E}_m\left[\mathbf{u}_m\right] \tag{25}$$

for use with the GP-LVM. This allows us to use a zero mean function. The mean vector $\bar{\mathbf{u}}$ is actually the average font from the training data, shown in Figure 12. This mean will be added on to any new $\hat{\mathbf{y}}$ vector returned from the manifold to generate a new font outline vector $\hat{\mathbf{u}}$; this means that the average font will be returned in regions of the manifold that are far away from the embedded fonts.

The GP-LVM is trained using these outline vectors for Y, as described in § 4.1, to produce the optimal set of low dimensional vectors $X^* = [\cdots \mathbf{x}_m^* \cdots]^{\mathrm{T}}$ and hyperparameters θ^*.

Generating a Font Once we have learnt the latent variables X^* and hyperparameters θ^*, generating a new font from the manifold is straight-forward (described in supplement § B.2) and computationally inexpensive. Consider $\hat{\mathbf{x}}$ as our location on the manifold. We obtain the corresponding high dimensional vector $\hat{\mathbf{y}}$ as

$$\hat{\mathbf{y}} = C(\hat{\mathbf{x}}, X^* | \theta^*)[C(X^*, X^* | \theta^*)]^{-1} Y \tag{26}$$

where $[C(X^*, X^* | \theta^*)]^{-1}$ is precomputed and finding the $1 \times M$ row vector $C(\hat{\mathbf{x}}, X^* | \theta^*)$ is a simple application of equation (20). From $\hat{\mathbf{y}}$, we add on the mean vector $\bar{\mathbf{u}}$, from (25), to get $\hat{\mathbf{u}}$, which contains the outline polylines from which all the characters can then be drawn.

Example Manifold Figure 1 illustrates the result of this process generating a two dimensional manifold (*i.e.* $Q = 2$). Each of the grey dots on the heatmap represents a $\mathbf{x}_j$ vector that corresponds to a font. In regions surrounding the fonts, the manifold will generate novel $\mathbf{u}$ vectors, each of which corresponds to a new and unique font. Therefore by traversing the manifold we smoothly interpolate and extrapolate from the existing fonts.

To illustrate the fonts generated from this manifold, the text to the right of the figure is generated by taking a linear trajectory across the manifold (marked in the figure in white with colored dots) and using the location to generate each character individually. The colored words match up with the colored dots as the font changes from a sans region of the manifold, in red at the start, to a serif region, in orange at the end. Every character is made from a non-linear mixture of a number of the input fonts and is representative of an entire typeface. The smooth motion over the manifold means that we observe no sharp transitions between the fonts but rather a gradual and disguised one. Each location on the manifold is able to generate an entire set of characters as in Figure 6.

Probabilistic Mapping The probabilistic nature of the mapping is displayed by the heatmap, in Figure 1, that encodes the model variance. Red regions are well mapped by the embedded fonts and are therefore likely to generate good fonts whereas the blue regions are far from existing data and are therefore more in the realm of extrapolation and hence a potential falloff in quality.

Manifold Dimensionality For the purpose of visualization, we have limited the manifold dimensionally in the figures to $Q = 2$. In reality, the true dimensionality of the manifold may be higher and we used a more advanced derivate of the GP-LVM, the Bayesian GP-LVM of Titsias and Lawrence [2010], to estimate it; we provide further details in supplement § B.3. For the joint manifold, shown in Figures 1 and 6, the best setting is obtained with $Q = 4$ manifold; this explains the separation into islands when constrained to $Q = 2$. Manifolds for individual characters may differ in intrinsic dimensionality, *e.g.* the character 'g' has $Q = 3$.

Additional Information As well as the outline information, we can also concatenate additional font specific information to the end of the high dimensional vectors. We add the outline offset and scale, which were taken off the font as part of the normalization process in § 3.2, as well as the font metric information (horizontal and vertical advance distances). These values will then be interpolated and extrapolated as appropriate along with the outlines to ensure the generated font has the correct scaling and metric information available.

Recovery of Bézier Curves Although the polyline representation is sufficient for browsing and displaying characters, to generate an actual font file (or for editing) we need to convert the polyline back into a bézier curve. We can do this using a standard least-squares fitting approach although there has been font specific research on curve fitting from optical scans, *e.g.* [Itoh and Ohno 1993], which could possibly confer improved performance.

Projection onto the Manifold Once we have trained a GP-LVM model, there are established techniques (*e.g.* [Navaratnam et al. 2007] or [Prisacariu and Reid 2011]) that may be used to project high-dimensional vectors, *i.e.* character outlines, onto the manifold. The standard approach is to define a cost function (or error measure) that compares a generated character outline $\mathbf{y}$ to some target $\mathbf{y}_{\text{target}}$; in the case of fonts, we use use the chamfer distance [Barrow et al. 1977] between the two outlines $E_{\text{chamfer}}(\mathbf{y}, \mathbf{y}_{\text{target}})$. An optimization is then performed; starting from an initial location on the manifold $\tilde{\mathbf{x}}$, the corresponding outline $\tilde{\mathbf{y}}(\tilde{\mathbf{x}})$ is found using (26). We then minimize

$$E_{\text{chamfer}}\left(\tilde{\mathbf{y}}(\tilde{\mathbf{x}}), \mathbf{y}_{\text{target}}\right) \tag{27}$$

iteratively by taking gradients with respect to $\tilde{\mathbf{x}}$; this is performed numerically. Since there may be local minima on the manifold we run the optimization from multiple starting locations, which span the manifold, in parallel; we can use the embedded font locations as starting points. This is not an expensive process since the dimensionality of the manifold Q is low and there are efficient ways to compute the chamfer distance.

Interactive Editing A practical example of manifold projection is font editing; we show this application in § 5. Our editing operation is for the user to select a point on the outline of a character and drag it to a new location; our desired outcome is for the rest of the

character outline to alter accordingly (to maintain an appropriate glyph) and for the edits to the single character to be propagated to others as necessary. Thus, if a user drags to increase the serif width on a character, we would like other characters with serifs to increase the width correspondingly; this is achieved by ensuring that the character under edit is always represented by a location on the manifold $\tilde{\mathbf{x}}$.

The font editing example is more straight-forward than general projection onto the manifold since we start from a known location (the initial font); this location could also be determined by manifold projection. During a drag operation we quantize each movement of the vertices into a series of small steps; for each step we consider the vertices on $\tilde{\mathbf{y}}$ nearest the cursor and move them to a new location (the target). By taking gradients, with respect to $\tilde{\mathbf{x}}$, of the distance between $\tilde{\mathbf{y}}(\tilde{\mathbf{x}})$ and the target (for the active vertices) we identify a new location, $\tilde{\mathbf{x}}'$, on the manifold to move to; this new location then gives a new outline $\tilde{\mathbf{y}}' = \tilde{\mathbf{y}}(\tilde{\mathbf{x}}')$ for all the characters. By performing a sequence of these steps we find a new manifold location and corresponding outline at the end of each editing operation. We can also provide the user with feedback during the drag: The variance (heat map) can be used to indicate when the user is getting close to the edge of the manifold; the outline will not be allowed to leave the manifold so will stop moving if the user drags too far.

5 Results and Applications

Interactive Manifold Exploration The best way to visualize the manifold results is via the interactive viewer. The browser is written in Javascript and runs standalone in a web browser. It is possible to explore 2D versions of manifolds for individual characters as well as several joint manifolds. The restriction to only two dimensions leads to some of the manifolds separating into islands and is thus not truly representative of the manifolds in their natural dimension. It is possible to observe the different fonts generated by dynamically changing the manifold location in real time.

Joint Font Manifold We demonstrate the variations in fonts captured by the learned manifold using the test word 'hamburgefon' in Figure 6. The light grey dots indicate the embedded locations of the training fonts. The heat map shows the predictive variance of the GP-LVM. Roughly speaking, the color indicates the likelihood of obtaining a useful interpolation of the data; points in the red region are representative of the characteristics of the training fonts and points in blue regions tend toward the average font. We observe that many font characteristics are captured by the manifold. This includes smooth transitions between serif and non-serif fonts; smooth changes in the strokes used in terms of thickness and the contrast ratio and angle; smooth variation in the aspect ratio (the fonts are normalized to have a fixed 'x' height). We provide details of the fonts used in supplement § A.

Single Character Manifolds We also learned manifolds for individual characters and show some samples of these in Figure 7. Again, we can see the variation captured and the efficacy of our matching technique. In addition to verification of the matching, the single manifolds can also be used to edit a character in isolation of the others; this could present more freedom for individual letters, *e.g.* for logo design. We note that sampling from lower likelihood regions can introduce artifacts, *e.g.* the second example for 'Q', however this is not necessarily the case, as in the first example for 'B'. We also show the manifold for the alternate 'a' glyph learned from fonts with the single storey character available.

Interpolation Comparison An alternative universal parameterization to our polyline would be to rasterize each font at a certain resolution yielding a fixed-length vector, either in a signed distance

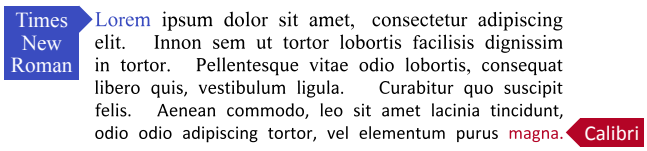


Figure 9: *Smoothly transforming between two typefaces using the character matching results. Each character is generated as a linear interpolation of the two input fonts and corresponds to a novel font.*

Figure 12: *The average font found from 46 fonts. We observe a font somewhere between a sans and a serif font since we had an even mix of the two in our training set. The asymmetries in some of the letters, for example the 'W', are due to asymmetries in some of the input fonts.*

domain or as a set of masses. We compare our interpolation results to those obtained by looking in the pixel space directly in Figure 8. In the top row we show the result of interpolating with the signed distance transform that has topology issues if the curves are not strictly inside one another. On the second row we use the Lagrangian mass transport method of Bonneel *et al*. [2011]. This has better performance but has no constraints to preserve the topology or provide smooth boundaries. The bottom row shows the interpolation using the correspondences found by our generative matching model. By minimizing the curvature variation we produce a smooth transition while maintaining the key characteristics of the character's visual appearance.

We can also perform interpolation between two fonts directly as a result of the character matching procedure. In Figure 9 we observe a linear interpolation between the character matches for the serif font 'Times New Roman' and the sans font 'Calibri'. The high quality of the matching leads to a smoothly varying output without any obvious jarring or discontinuities.

Font Editing An example of a novel application made possible by the joint font manifold is the propagation of user edits. Traditional font editing packages have a steep learning curve and require specialist training in typography. We can constrain the result of any basic editing operation to remain on the manifold and, therefore, ensure that at every stage in the proceedings we have a valid font. Furthermore, the manifold contains joint information between different characters, therefore edits to a single character may be propagated across all other characters using the manifold.

In Figures 10 and 11 we show examples of a user editing a font by simply grabbing a location on the outline of a character and dragging it. Propagated results are shown across similar characters to demonstrate that they move in sympathy with the edits. In Figure 10, the user makes the bottom stroke of the '2' thinner and we see the other digits copying this to produce thinner strokes while maintaining the stroke contrast. This is an operation that would be very labor intensive with existing tools, even for an expert typographer.

Similarly, in Figure 11, the user wishes to increase the serif width of the 'm' character. By grabbing the tip of the existing serif and dragging it to the right, not only are the serifs within the 'm' character changed to stay symmetric, but the serifs across all the other characters are widened to stay consistent.

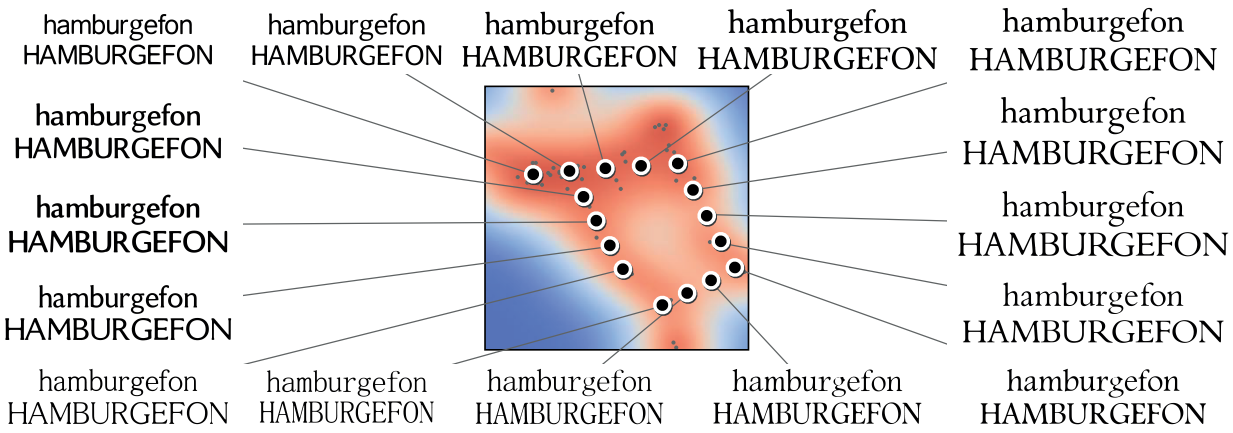


Figure 6: *Typography samples of the test word 'hamburgefon' over the learnt joint manifold of fonts. We can observe many changes over the manifold including variations in the presence of serifs; the stroke contrast, thickness and angle; and the aspect ratio. The heat map of the manifold roughly indicates the probability of finding good fonts as in Figure 1; see the text for further details.*

Figure 7: *Explorations of a sample of the single manifolds demonstrating our matching results. We find each manifold on the left using only the outlines for a single character. The points and path drawn on the manifold indicates where the samples on the right are taken from. The color coding and path indicate the points on the manifold used to generate each glyph. See Figure 13 for matching limitations.*

The Average Font The Avería font[3] is a project to create an 'average font' by a user with no formal typography training; instead of matching, their process uses a simple superposition of rasterizations of many fonts. For curiosity and comparison, we present the average font from our matching in Figure 12; this is $\tilde{\mathbf{u}}$ from (25). As expected, we see a font between sans and serif due to the equal mix of the two in our training fonts.

[3] http://iotic.com/averia/

Parameters and Timing We used $M = 46$ fonts to generate our manifolds (details are provided in supplement § A). For the matching procedure we used $N = 512$ samples for each outline and the energy parameters: $\gamma = 50$, $\beta = 5$, $\lambda_\eta = 1$ and $\lambda_{\text{el}} = (MN^2/6000)$. The optimization stage took an average of 20 minutes for each outline. This is a one off procedure and each outline can be matched in parallel independently. The manifold learning took under 2 minutes (again a one off procedure). Once all the processing has been performed the generation of a font from the manifold is in the order of milliseconds, as is demonstrated by the Javascript based manifold browser.

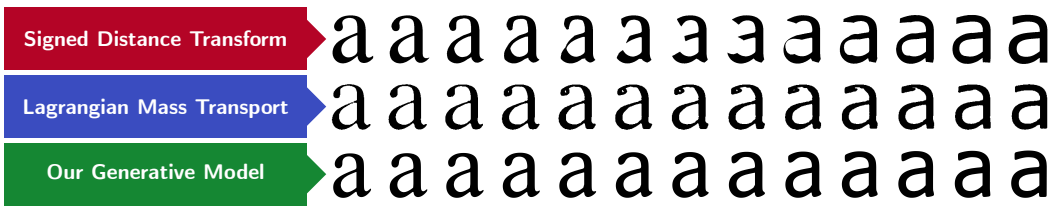


Figure 8: *Comparison of our matching approach to interpolation methods. Moving from left to right interpolates from the font 'Times New Roman' to 'Tahoma'. The signed distance transform has topological issues. While the Lagrangian mass transport method improves, it has no constraints to preserve topology or provide smooth boundaries. Our curvature optimization result produces a smooth transition while maintaining the visual characteristics of the character for all interpolants.*

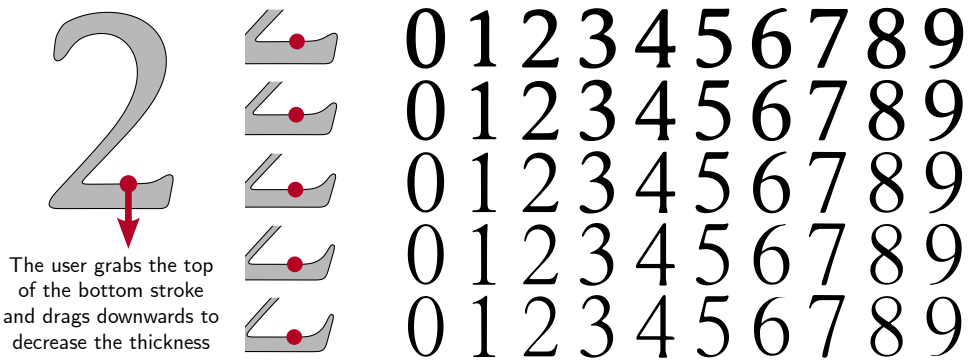


Figure 10: *An example of interactive font editing. The manifold can be used to propagate edits to a single character across the entire font. The user edits the '2' digit by selecting at the red dot and dragging the outline down to reduce the thickness of the base. As the user drags the outline to a new location, we infer the change in location on the manifold. This means that not only will the whole outline of the '2' change to maintain a reasonable font, but the changes can be propagated to all the other characters; we show all the other digits as they change.*

6 Conclusion

We have presented a framework that takes a set of existing fonts and successfully learns a generative manifold of fonts. This unsupervised learning process requires no input from either an end user or a professional typographer and yet is capable of generating new, high quality fonts. The results presented in the previous section, and the interactive manifold browser, demonstrate the efficacy of the combination of our novel energy model and optimization scheme for dense character matching, and the non-linear manifold embedding.

Armed with this font manifold, we have shown that a number of new applications are now made possible. By constraining edits to lie on the manifold, non-expert users can manipulate fonts without any formal training in typography and guarantee that the results of their edits will produce recognizable fonts. In addition, the time requirements for editing are greatly reduced since edits can be propagated across entire fonts, maintaining consistency of appearance, rather than having to edit each glyph independently. Such automation is also time saving for expert type designers. Generation of fonts on the manifold is computationally efficient so all of these processes may be performed at interactive speeds.

In addition to providing smooth interpolations between fonts, the manifold may also be used directly as a means to explore the space of fonts, whether browsing for a new font or trying to identify possible new fonts to fill a gap in those currently available.

6.1 Limitations and Future Work

Character Matching Since our matching method is based around matching outlines, we cannot match two characters with differing topology as discussed in § 3.1. For 'a' and 'g' we treat the two glyphs separately; we show a manifold for the alternate 'a' in Figure 7. If a font is missing the appropriate character version, it can still be embedded in the manifold using a GP-LVM with missing data [Navaratnam et al. 2007]; this will also produce the best estimate of the missing character given the the other fonts.

While the results of the character matching are very successful there is an occasional matching failure when a font has an outlying character that is far from the outlines in the other fonts (the shape is under-represented in the training fonts). Figure 13 shows an example for the character 'J'. Here the representative examples, both sans and serif fonts, are matched correctly whereas the long horizontal stroke of the outline on the left is too far from any other fonts and is incorrectly matched to the vertical stroke. This can introduce a local artifact on the manifold but usually outlying fonts are on the edges of the manifold and therefore distortion is minimized. Another such example is seen in the second 'Q' sample in Figure 7; the crossing tail in the first sample is rare and therefore the matching finds it difficult to blend in to the purely descending tails. Outliers may be reduced by increasing the number and variety of fonts.

Hinting and Kerning In our present system, the generated fonts do not have any kerning or hinting information. The addition of kerning is a straight-forward extension by concatenating the additional kerning tables to the end of the high dimensional vector. The transfer of hinting information is more challenging and is an area for future work.

Typographic Supervision As we have discussed previously, our learning approach is fully unsupervised and requires no input from a user or an expert typographer. As such, we do not break down characters into individual components under standard typographic rules; this may limit the extrapolation to new fonts by placing constraints on the fonts that can be generated from our manifold. In future work we would like to make use of recent advances in 'deep learning' to perform more advanced unsupervised learning to identify automatically these constituent parts.

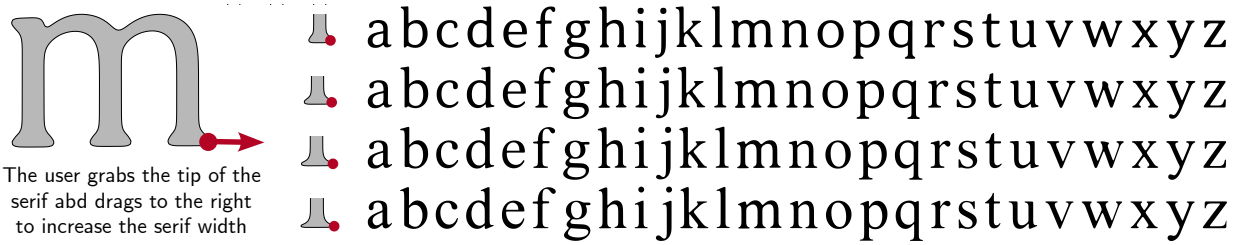


Figure 11: *Edit to widen the serif. As the user drags the edge of the 'm' serif to the right we observe the serifs in the other letters widening in sympathy to remain consistent and on the manifold. Thus the user can change the serif width for the entire font by editing a single character.*

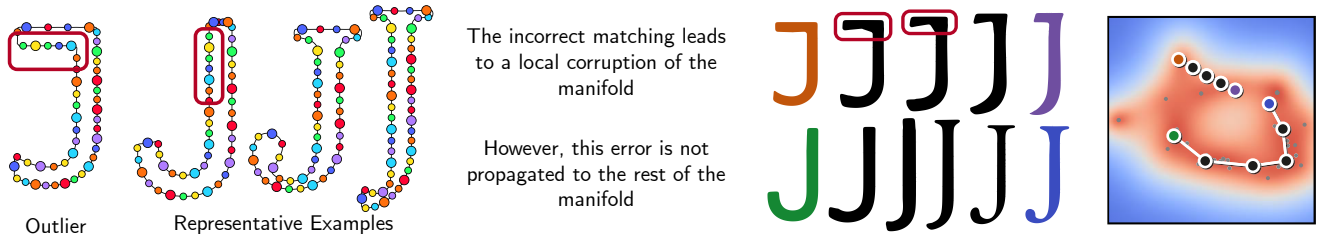


Figure 13: *Occasionally the matching can fail for specific examples when there is a font with an outlying character, for example the 'J' shown here. The representative examples are all correctly matched however the long extended top stroke on the example on the right is an outlier. By matching to the ascenders of the typical examples, artifacts can be introduced locally for a small part of the manifold however this effect does not disrupt the entire manifold.*

Acknowledgements

The authors would like to thank Andrew Fitzgibbon for insightful discussions about optimization philosophy and techniques. This work was funded by EPSRC grant EP/I031170/1.

References

ADOBE SYSTEMS. 1997. Designing multiple master typefaces. Technical Note #5087.

ALEXA, M., COHEN-OR, D., AND LEVIN, D. 2000. As-rigid-as-possible shape interpolation. In *Proc. of SIGGRAPH '00*, 157–164.

BARROW, H., TENENBAUM, J., BOLLES, R., AND WOLF, H. 1977. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *Proc. of the International Joint Conference of Artificial Intelligence*, 659–663.

BONNEEL, N., VAN DE PANNE, M., PARIS, S., AND HEIDRICH, W. 2011. Displacement interpolation using lagrangian mass transport. *ACM Trans. Graphics 30*, 6, 158:1–158:12.

HASSAN, T., HU, C., AND HERSCH, R. D. 2010. Next generation typeface representations: Revisiting parametric fonts. In *Proc. 10^{th} ACM Symposium on Document Engineering*, 181–184.

HERSCH, R. D., AND BETRISEY, C. 1991. Model-based matching and hinting of fonts. *Proc. of SIGGRAPH '91*, 71–80.

HU, C., AND HERSCH, R. D. 2001. Parameterizable fonts based on shape components. *IEEE Comput. Graphics Appl. 21*, 3, 70–85.

ITOH, K., AND OHNO, Y. 1993. A curve fitting algorithm for character fonts. *Electronic Publishing 6*, 3, 195–205.

KALOGERAKIS, E., CHAUDHURI, S., KOLLER, D., AND KOLTUN, V. 2012. A probabilistic model for component-based shape synthesis. *ACM Trans. Graphics 31*, 4, 55:1–55:11.

KAZHDAN, M., FUNKHOUSER, T., AND RUSINKIEWICZ, S. 2004. Shape matching and anisotropy. *ACM Trans. Graphics 23*, 3, 623–629.

KIM, V. G., LI, W., MITRA, N. J., CHAUDHURI, S., DIVERDI, S., AND FUNKHOUSER, T. 2013. Learning part-based templates from large collections of 3d shapes. *ACM Trans. Graphics 32*, 4, 70:1–70:12.

KNUTH, D. E. 1986. *The Metafont Book*. Addison-Wesley.

KUHL, F., AND GIARDINA, C. 1982. Elliptic fourier features of a closed contour. *Computer Graphics and Image Proc. 18*, 3, 236–258.

LAU, V. M. K. 2009. Learning by example for parametric font design. In *SIGGRAPH ASIA '09 Posters*, 5:1–5:1.

LAWRENCE, N. 2005. Probabilistic non-linear principal component analysis with gaussian process latent variable models. *The Journal of Machine Learning Research 6*, 1783–1816.

LOVISCACH, J. 2010. The universe of fonts, charted by machine. In *SIGGRAPH '10 Talks*, 27:1–27:1.

NAVARATNAM, R., FITZGIBBON, A., AND CIPOLLA, R. 2007. The joint manifold model for semi-supervised multi-valued regression. In *Proc. of IEEE 11^{th} International Conf. on Computer Vision*.

PRISACARIU, V., AND REID, I. 2011. Nonlinear shape manifolds as shape priors in level set segmentation and tracking. In *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*.

RASMUSSEN, C. E., AND WILLIAMS, C. 2006. *Gaussian Processes for Machine Learning*. MIT Press.

SHAMIR, A., AND RAPPOPORT, A. 1998. Feature-based design of fonts using constraints. In *Proc. of Raster Imag. and Dig. Typography '98*.

SUVEERANONT, R., AND IGARASHI, T. 2010. Example-based automatic font generation. In *Proc. of 10^{th} Int. Symp. on Smart Graphics*.

TITSIAS, M., AND LAWRENCE, N. 2010. Bayesian gaussian process latent variable model. In *Proc. of 13^{th} Int. Workshop on AI and Stats.*

VAN KAICK, O., ZHANG, H., HAMARNEH, G., AND COHEN-OR, D. 2011. A survey on shape correspondence. *Computer Graphics Forum 30*, 6, 1681–1707.

ZITNICK, C. L. 2013. Handwriting beautification using token means. *ACM Trans. Graphics 32*, 4, 53:1–53:8.

ZONGKER, D. E., WADE, G., AND SALESIN, D. H. 2000. Example-based hinting of true type fonts. In *Proc. of SIGGRAPH '00*, 411–416.