Eurographics Conference on Visualization (EuroVis) 2018
J. Heer, H. Leitte, and T. Ropinski
(Guest Editors)

Volume 37 (2018), Number 3

Rendering and Extracting Extremal Features in 3D Fields

G. Kindlmann[1], C. Chiw[1], T. Huynh[1], A. Gyulassy[2], J. Reppy[1], P.-T. Bremer[2]

[1]Department of Computer Science, University of Chicago
[2]SCI Institute, University of Utah

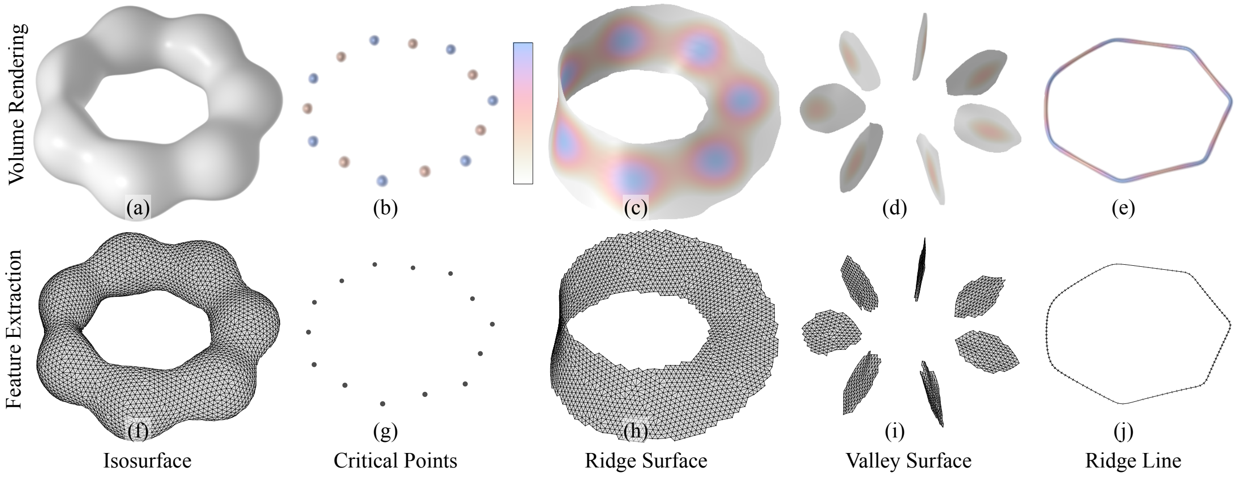


Figure 1: *Direct volume renderings (top row) and meshes (bottom row) show the structure of one synthetic dataset. One program computed all renderings, and another computed the mesh vertices. Between features (columns), the only differences in their source code were functions for computing a Newton step to the feature, and for measuring feature strength. These functions were shared between the two programs, achieving orthogonality between implementing visualization algorithms, and specifying the particular features of interest.*

Abstract

Visualizing and extracting three-dimensional features is important for many computational science applications, each with their own feature definitions and data types. While some are simple to state and implement (e.g. isosurfaces), others require more complicated mathematics (e.g. multiple derivatives, curvature, eigenvectors, etc.). Correctly implementing mathematical definitions is difficult, so experimenting with new features requires substantial investments. Furthermore, traditional interpolants rarely support the necessary derivatives, and approximations can reduce numerical stability. Our new approach directly translates mathematical notation into practical visualization and feature extraction, with minimal mental and implementation overhead. Using a mathematically expressive domain-specific language, Diderot, we compute direct volume renderings and particle-based feature samplings for a range of mathematical features. Non-expert users can experiment with feature definitions without any exposure to meshes, interpolants, derivative computation, etc. We demonstrate high-quality results on notoriously difficult features, such as ridges and vortex cores, using working code simple enough to be presented in its entirety.

CCS Concepts

•***Computing methodologies*** $\rightarrow$ ***Scientific visualization;*** •***Software and its engineering*** $\rightarrow$ ***Domain specific languages;*** •***Human-centered computing*** $\rightarrow$ *Visualization systems and tools;*

1. Introduction

Many different analysis and visualization techniques for spatio-temporal data share an overarching goal: to better understand some feature of interest. Features may be defined *locally*, such as isosurfaces as points with a certain value, or *globally*, as with streamlines and separatrices. In general, local feature definitions are more common and cover features ranging from isosurfaces and ridges in scalar fields [Ebe96], to vortex core lines in vector fields [PR99] and crease lines in tensor fields [TKW08]. Depending on the application, these definitions are either approximated by some (potentially fuzzy) indicator, such as a transfer function in volume rendering, or explicitly evaluated in an algorithm like Marching

Cubes [LC87]. However, feature definitions in visualization research have long evolved from the simple definitions to complex, often multi-variate expressions that involve higher order derivatives, curvatures, Jacobians and any number of mathematical expressions not easily built from scratch. Furthermore, the common bi- or trilinear interpolation schemes do not provide continuous derivatives of any order. Therefore, applying advanced feature definition typically requires various approximations and often leads to numerical instabilities and significant artifacts.

A typical example is ridge extraction, which is concisely defined in terms of a local indicator involving gradients and Hessian eigenvectors [Ebe96]. Computing the indicator with stencils for derivatives is not stable, often producing false positives and needing some pre-smoothing to be effective [SP07, PS08]. Finally, even with higher-order interpolation [NLKH12, NKH13], generalizing many of the traditional algorithms has proven challenging.

Our new approach allows users to easily express complex features directly in common mathematical notation. Given sampled data and a sufficiently smooth and accurate reconstruction, these expressions produce computationally tractable methods of visualization and extraction. Since all derivative and tensor evaluations are performed analytically, our feature definitions do not suffer from common discretization artifacts. The implementation complexity is low: experimenting with new definitions is more akin to copying mathematical definitions into code, rather than incurring per-feature and per-application implementation costs.

The two main contributions of this work are (1) conceptually bridging direct volume rendering of features with their explicit geometric extraction, and (2) implementing both in a way that is general with respect to feature type and application. We start with standard mathematical feature definitions (one or more equations satisfied at points *within* the feature, e.g. $\{\mathbf{x} | f(\mathbf{x}) = v_0\}$ for isosurfaces), but we employ a *feature step* function that describes one step *towards* the feature (e.g. the Newton-Raphson step $\frac{(v_0 - f(\mathbf{x}))\nabla f(\mathbf{x})}{\nabla f(\mathbf{x}) \cdot \nabla f(\mathbf{x})}$ for isosurfaces). The same feature step function can either be evaluated once to define a transfer function for direct volume rendering features, or evaluated repeatedly by particles uniformly sampling a feature. Code for rendering and extracting features is made specific to the feature, and to the application of interest, solely by choosing the step function for that feature, and by defining the field in which the feature is sought. Fig. 1 illustrates our approach with isosurfaces and some of the *extremal features* (e.g. ridges, valleys, critical points) that are the focus of this initial work. Our approach is enabled by Diderot, a domain-specific language for scientific visualization that combines mathematical expressivity with parallel execution [CKR*12, KCS*16, CKR18].

2. Background and Related Work

The terminology of “extremal features” has been used in different ways. For flow fields, Sahner et al. use “extremal features” to mean topological separatrices of strain (as a scalar field), defined globally rather locally [SWTH07]. In tensor fields, Zobel and Scheuermann locally define “extremal features” as points with a non-invertible tangent space map from the field domain to the 3D space of invariants [ZS17]. Following the earlier terminology [GM97, TM98, AK04], *we consider extremal features to be the extrema of some scalar field with respect to motion within a locally defined constraint subspace, as well as critical points of the scalar field*. Ridges and valleys are extremal relative to motion along one or more Hessian eigenvectors [Ebe96]. How we quantify the significance or strength of ridge features is informed by Haralick [Har83] and Schultz et al. [STS10].

Barakat et al. share our interest in rendering [BT10] and extracting [BAT11] ridges. Our approach is made coherent by a single feature step function, whereas Barakat et al. used different GPU-based methods to optimize each task. Their volume rendering [BT10] uses space-skipping, while ours is more brute-force. Their ridge extraction [BAT11] computes vertex locations and mesh geometry at the same time within a sweeping front, while we view meshing as an optional post-process of vertex computation. Unlike their work, our approach can easily and consistently handle a variety of features and feature dimensions (0D, 1D, and 2D). Our interest in visualizing extremal features is related to finding maxima in raycast volume rendering. Knoll et al. [KHW*09] find ray locations that maximize 1D opacity function, to ensure their contribution to the ray result. Kotava et al. [KKS*12] extend this to multidimensional transfer functions. Both their approaches draw on the same basic principles of computational math that we employ in Sec. 3, but their focus remains on fast rendering, while we seek to accomplish both rendering and explicit geometric extraction.

We implement our approach in Diderot [CKR*12, KCS*16], a mathematically expressive language with a tensor-based intermediate representation [Chi17, CKR18]. The syntax includes ∇ for differentiation, $\otimes$ for tensor product, and $\bullet$ for inner product, combined with the ability to apply operators to fields (to produce, for example, a scalar field of curvature). This allows the fields and features of Sec. 3 to be directly expressed in code, while parallel execution (via pthreads) facilitates work with real datasets. Relative to prior Diderot work, novel aspects of this work include: generalization of volume rendering from isosurfaces to ridge and other extremal surfaces, more sophisticated particle-based feature extraction (with meshing, in 3-D, instead of only 2-D), and an implementation design that isolates feature specificity to a few functions.

The advantages of Diderot for our work, relative to other languages and tools, rest in the mathematical abstractions supported by Diderot. MATLAB and Mathematica, for example, allow symbolic expression of functions, making it easy to write down expressions for ridges, vortices, etc. However, their ability to compute on real-world volume data is limited. First, their built-in reconstruction has limited order of continuity and accuracy, resulting in poor-quality second derivatives, and the inability to compute properties (like surface creases) that depend on higher derivatives. Second, volume rendering pipelines in these tools, while allowing custom transfer functions, are available with only built-in interpolants. To reproduce our results, a user would have to re-implement interpolation as well as ray casting, which would suffer immense performance penalty in these interpreted languages compared to the compiled binaries produced by Diderot.

There also exist a number of general analysis tools such as VisIt, Paraview, or Amira that can be used to extract a wide variety of features. However, these are fundamentally built around *discretized* domains and algorithms, while Diderot directly supports *continuous fields* and operations on them. While some other tools do support higher order interpolants, they do not allow expressing mathematical formulations in an abstract manner, nor do they provide the analytical evaluation of such expressions. Users must instead evaluate expressions by hand, typically using finite difference stencils, with the challenges that this entails. Finally, these tools do not provide a means to extract smooth features directly but rely on sampling the corresponding indicator functions on a grid and using techniques such as Marching Cubes to extract the geometry.

3. Mathematical Background

Our method is based on Taylor expansions. For isosurfaces, Sec. 3.1 reviews how a first-order Taylor expansion generates the formula for one iteration of Newton-Raphson root finding, which we term the *feature step* function $\mathbf{s}(\mathbf{x})$. We note a connection to Levoy's bivariate transfer function for isocontours [Lev88]. We then develop formulae for feature step functions for ridges and valleys as defined in [Ebe96]. Sec. 3.2 extends this to other extremal features of interest. Sec. 3.3 then formulates functions for *feature strength* to filter out insignificant features, and we note a connection to previous criteria for filtering ridge pixels [Har83].

3.1. Basic Feature Step Functions, and Shading

The 1st-order Taylor expansion of scalar field f around $\mathbf{x}$ is

$$f(\mathbf{x}+\boldsymbol{\varepsilon}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \boldsymbol{\varepsilon} \tag{1}$$

If we assume $f(\mathbf{x}+\boldsymbol{\varepsilon}) = v_0$ for some isovalue v_0, then (1) describes a condition on the offsets $\boldsymbol{\varepsilon}$ from $\mathbf{x}$ to a point $\mathbf{x}+\boldsymbol{\varepsilon}$ on the v_0 isocontour. We can choose an offset $\boldsymbol{\varepsilon} = \ell \frac{\nabla f(\mathbf{x})}{|\nabla f(\mathbf{x})|}$ parallel to the gradient $\nabla f(\mathbf{x})$, and then solve for step length ℓ to arrive at a formula for the feature step function $\mathbf{s}_{iso}$ for isosurfaces:

$$v_0 - f(\mathbf{x}) \approx \nabla f(\mathbf{x}) \cdot \boldsymbol{\varepsilon} = \nabla f(\mathbf{x}) \cdot \ell \frac{\nabla f(\mathbf{x})}{|\nabla f(\mathbf{x})|} = \ell |\nabla f(\mathbf{x})| \tag{2}$$

$$\Rightarrow \ell = \frac{v_0 - f(\mathbf{x})}{|\nabla f(\mathbf{x})|} \;\Rightarrow\; \mathbf{s}_{iso}(\mathbf{x}) = \frac{(v_0 - f(\mathbf{x}))\nabla f(\mathbf{x})}{\nabla f(\mathbf{x}) \cdot \nabla f(\mathbf{x})}. \tag{3}$$

For locations $\mathbf{x}$ near isocontour $f(\mathbf{x}) = v_0$, $\mathbf{s}_{iso}(\mathbf{x})$ (3) indicates the distance and direction towards the feature.

To volume render, we assign opacity with a tent function parameterized by distance d to an isocontour of interest

$$\alpha_{tent}(d) = \alpha_0 \max\left(0, 1 - \frac{|d|}{w}\right), \tag{4}$$

where w is the full-width half-max of $\alpha_{tent}(d)$, the apparent thickness of the rendered isocontour. Plugging (3) into (4):

$$\alpha(\mathbf{x}) = \alpha_{tent}(|\mathbf{s}_{iso}(\mathbf{x})|) = \alpha_0 \max\left(0, 1 - \frac{1}{w}\frac{|v_0 - f(\mathbf{x})|}{|\nabla f(\mathbf{x})|}\right). \tag{5}$$

Eq. (5) is exactly Levoy's bivariate opacity function for rendering approximately constant-thickness isocontours, which he justifies with assumptions equivalent to (1); c.f. (3) in [Lev88].

Feature step functions also facilitate building geometric models of features. Locations of model vertices may be found by iterated application of the feature step function

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{s}(\mathbf{x}_i), \tag{6}$$

combined with some strategy (such as mutual repulsion) for uniformly distributing points across the feature. Many have used this Newton-based approach to sample implicit surfaces [CA97, MGW05] (among others).

The above treatment of isocontours exemplifies our approach. Taylor expansion around the feature of interest leads to a feature step function, which can then be plugged into either an opacity function for volume rendering the feature, or into the constraint satisfaction of an interacting particle system to explicitly sample the feature. *Feature visualization or extraction programs become specific to the feature via the feature step function.* We now derive feature step functions $\mathbf{s}(\mathbf{x})$ for various extremal features.

Newton-Raphson *optimization*, to find critical points $\mathbf{x}$ where $\nabla f(\mathbf{x}) = \mathbf{0}$, is also based on Taylor expansion. Differentiating (1) with respect to $\mathbf{x}$ produces

$$\nabla f(\mathbf{x}+\boldsymbol{\varepsilon}) \approx \nabla f(\mathbf{x}) + \mathbf{H}f(\mathbf{x})\boldsymbol{\varepsilon}, \tag{7}$$

where $\mathbf{H}f(\mathbf{x})$ is the Hessian (second derivative) of f at $\mathbf{x}$. Assuming $\mathbf{x}+\boldsymbol{\varepsilon}$ is a critical point (that is, $\nabla f(\mathbf{x}+\boldsymbol{\varepsilon}) = 0$), then

$$-\nabla f(\mathbf{x}) \approx \mathbf{H}f(\mathbf{x})\boldsymbol{\varepsilon} \;\Rightarrow\; \mathbf{s}_{cp}(\mathbf{x}) = -(\mathbf{H}f(\mathbf{x}))^{-1}\nabla f(\mathbf{x}). \tag{8}$$

The signs of Hessian eigenvalues determine the type of the critical point (minima, maxima, or saddle point).

We use Eberly's definitions of ridge (and valley) surfaces and lines [Ebe96] to define their feature step formulae. The Hessian $\mathbf{H}f(\mathbf{x})$ at $\mathbf{x}$ has sorted real eigenvalues $\lambda_0 \geq \lambda_1 \geq \lambda_2$ and corresponding unit-length and mutually orthogonal eigenvectors $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2$. Point $\mathbf{x}$ is on a ridge surface if f is maximal at $\mathbf{x}$ with respect to motion along eigenvector $\mathbf{e}_2$ (corresponding with the most negative eigenvalue λ_2), i.e. $\nabla f(\mathbf{x}) \cdot \mathbf{e}_2 = 0$ and $\lambda_2 < 0$. Ridge line membership is defined by $\nabla f(\mathbf{x}) \cdot \mathbf{e}_1 = \nabla f(\mathbf{x}) \cdot \mathbf{e}_2 = 0$ and $\lambda_1 < 0$. Ridge and valley features can be uniformly treated by defining a projection $\mathbf{P}$ onto the span of one or two eigenvectors, with

$$\begin{aligned} \mathbf{P}_{rs} &= \mathbf{e}_2 \otimes \mathbf{e}_2, & \mathbf{P}_{rl} &= \mathbf{e}_1 \otimes \mathbf{e}_1 + \mathbf{e}_2 \otimes \mathbf{e}_2, \\ \mathbf{P}_{vs} &= \mathbf{e}_0 \otimes \mathbf{e}_0, & \mathbf{P}_{vl} &= \mathbf{e}_0 \otimes \mathbf{e}_0 + \mathbf{e}_1 \otimes \mathbf{e}_1, \end{aligned} \tag{9}$$

for ridge surfaces (*rs*) and lines (*rl*), and valley surfaces (*vs*) and lines (*vl*). Left-multiplying both sides of (7) by $\mathbf{P}$ gives

$$\mathbf{P}\nabla f(\mathbf{x}+\boldsymbol{\varepsilon}) \approx \mathbf{P}\nabla f(\mathbf{x}) + \mathbf{P}\mathbf{H}f(\mathbf{x})\boldsymbol{\varepsilon}. \tag{10}$$

Ridge/valley feature membership of $\mathbf{x}+\boldsymbol{\varepsilon}$ implies $\mathbf{P}\nabla f(\mathbf{x}+\boldsymbol{\varepsilon}) = \mathbf{0}$, i.e. $f(\mathbf{x}+\boldsymbol{\varepsilon})$ is at an extremum with respect to motion within the range of $\mathbf{P}$. $\mathbf{P}$ and $\mathbf{H}f(\mathbf{x})$ commute ($\mathbf{P}\mathbf{H}f(\mathbf{x}) = \mathbf{H}f(\mathbf{x})\mathbf{P}$) since in the eigenvector basis they are both diagonal matrices. Analogous to how in (1) we chose $\boldsymbol{\varepsilon}$ parallel to ∇f to derive $\mathbf{s}_{iso}$ (3), in (10) we chose to align $\boldsymbol{\varepsilon}$ with relevant Hessian eigenvectors: $\mathbf{P}\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}$. All combined, these simplifications to (10) give

$$-\mathbf{P}\nabla f(\mathbf{x}) = \mathbf{H}f(\mathbf{x})\boldsymbol{\varepsilon} \tag{11}$$

$$\Rightarrow \mathbf{s}_{ft}(\mathbf{x}) = -(\mathbf{H}f(\mathbf{x}))^{-1}\mathbf{P}_{ft}\nabla f(\mathbf{x}), \tag{12}$$

where feature type *ft* can be *rs*, *rl*, *vs*, or *vl* (c.f. 9). The product (12) can be simplified by recognizing again that both $\mathbf{P}$ and the Hessian inverse are diagonal matrices in the Hessian eigenvector basis. With ridge surfaces, for example,

$$\mathbf{s}_{rs}(\mathbf{x}) = -(\mathbf{H}f(\mathbf{x}))^{-1}\mathbf{P}_{rs}\nabla f(\mathbf{x}) = -\frac{\mathbf{e}_2(\mathbf{e}_2 \cdot \nabla f(\mathbf{x}))}{\lambda_2}. \quad (13)$$

Obermaier et al. used (13) for ridge surface sampling [OMD*12]. When iterating (12) converges, tests on the Hessian eigenvalue signs can verify membership in the desired ridge/valley feature.

Besides assigning opacity, direct volume rendering uses shading to convey surface orientation and shape. The codimension-one and codimension-two features considered above have analytically defined normals and tangents, respectively, that can be used for shading. In the current work, we explore using the step function $\mathbf{s}(\mathbf{x})$ itself for shading, based on an *approximate* surface normal $-\mathbf{s}(\mathbf{x})/|\mathbf{s}(\mathbf{x})|$. This gives correct shading for isocontours, but not for ridge and valley features; their normals and tangents depend on third derivatives [Ebe96]. In addition, since ridges may be non-orientable [STS10], we use two-sided lighting as a simple approach that works adequately for all features.

3.2. Other Extremal Features

To define the feature step function for extremal *surfaces*, let $\mathbf{c}$ be the unit-length direction along which the extremum of scalar f is sought (to make $\nabla f \cdot \mathbf{c} = 0$). The desired step along $\mathbf{c}$ at $\mathbf{x}$ is then a straight-forward 1D Newton optimization problem in terms of the first f' and second f'' directional derivatives:

$$f' = \nabla f \cdot \mathbf{c}; \quad f'' = \mathbf{c} \cdot (\nabla \otimes \nabla f)\mathbf{c} \quad (14)$$

$$\mathbf{s}_{-s}(\mathbf{x}) = \mathbf{s}_{+s}(\mathbf{x}) = -\frac{f'}{f''}\mathbf{c} \quad (15)$$

with $-s$ or $+s$ indicating minimal and maximal surfaces, respectively; the same Newton step applies to both. The sign of f'' determines the extremum type.

For extremal *lines*, let $\mathbf{c}_0$, $\mathbf{c}_1$ span the plane within which the extrema of f is sought (to make $\nabla f \cdot \mathbf{c}_0 = \nabla f \cdot \mathbf{c}_1 = 0$), and let g and H be the projection of the gradient ∇f and Hessian $\mathbf{H} = \nabla \otimes \nabla f$, respectively, onto that plane:

$$g = \begin{bmatrix} \mathbf{c}_0 \cdot \nabla f \\ \mathbf{c}_1 \cdot \nabla f \end{bmatrix}; \quad H = \begin{bmatrix} \mathbf{c}_0 \cdot \mathbf{H}\mathbf{c}_0 & \mathbf{c}_0 \cdot \mathbf{H}\mathbf{c}_1 \\ \mathbf{c}_1 \cdot \mathbf{H}\mathbf{c}_0 & \mathbf{c}_1 \cdot \mathbf{H}\mathbf{c}_1 \end{bmatrix}. \quad (16)$$

Whether seeking the local maxima or minima of f restricted to the span of $\{\mathbf{c}_0, \mathbf{c}_1\}$, (to find maximal line or minimal line features, respectively), we expand the 2D Newton step in the $\{\mathbf{c}_0, \mathbf{c}_1\}$ basis:

$$s = \begin{bmatrix} s_0 \\ s_1 \end{bmatrix} = -H^{-1}(\mathbf{x})g(\mathbf{x}) \quad (17)$$

$$\mathbf{s}_{-l}(\mathbf{x}) = \mathbf{s}_{+l}(\mathbf{x}) = s_0\mathbf{c}_0 + s_1\mathbf{c}_1 \quad (18)$$

with $-l$ or $+l$ for minimal or maximal lines, respectively.

We can also consider intersections of features. *Surface creases*, for example, are the intersection of isosurfaces and extremal surfaces of curvature. The surface normal map is $\nabla \otimes \mathbf{n}$, i.e. $\mathbf{n}(\mathbf{x}+\boldsymbol{\varepsilon}) \approx \mathbf{n}(\mathbf{x}) + (\nabla \otimes \mathbf{n})\boldsymbol{\varepsilon}$ describes local change in surface normal $\mathbf{n}$ around $\mathbf{x}$ for offsets $\boldsymbol{\varepsilon}$ tangent to the surface ($\boldsymbol{\varepsilon} \cdot \mathbf{n} = 0$). Curvature along unit-length tangent direction $\mathbf{d}$ is then [Car76]

$$\kappa(\mathbf{d}) = \mathbf{d} \cdot (\nabla \otimes \mathbf{n})\mathbf{d}. \quad (19)$$

Curvature is extremal along the *curvature directions* $\mathbf{d}_1$ and $\mathbf{d}_2$, with *principal curvatures* $\kappa_1 = \kappa(\mathbf{d}_1)$ and $\kappa_2 = \kappa(\mathbf{d}_2)$: $\kappa_1 \geq \kappa_2$ and curvature $\kappa(\mathbf{d})$ for any other direction $\mathbf{d}$ is within $[\kappa_2, \kappa_1]$. Considering the local variation $\nabla\kappa_1$, $\nabla\kappa_2$ of the principal curvatures, surface creases are points $\mathbf{x}$ where $\nabla\kappa_1 \cdot \mathbf{d}_1 = 0, \kappa_1 > 0$ (outward creases) and $\nabla\kappa_2 \cdot \mathbf{d}_2 = 0, \kappa_2 < 0$ (inward creases), i.e., a principal curvature is extremal with respect to motion along the corresponding curvature direction [Koe90]. In the context of volumetric image analysis, these have been called *crest lines* [MLD94, MB95].

3.3. Feature Strength and Mask Functions

Features are worth visualizing or extracting insofar as they indicate the presence and shape of some underlying structure of interest, rather than artifacts or background noise. To complement the feature step functions that point towards features, we use *feature strength* functions to render or extract only those portions of features exceeding a user-defined strength threshold. Defining feature strength functions and thresholds is unfortunately less clear-cut than deriving step functions. For isosurfaces, a non-zero gradient magnitude may suffice, since that ensures the feature step is finite, even without indicating feature strength per se.

Following Haralick [Har83], Schultz et al. [STS10] judge ridge surface significance by testing whether $|\nabla f|/\lambda_2 < 0$ (in our notation) is above a negative threshold. Comparing to the feature step function for ridge surfaces (13), we note that when ∇f is parallel to eigenvector $\mathbf{e}_2$, this is equivalent to testing for small feature step length $|\mathbf{s}_{rs}|$, but it more generally penalizes a large gradient component orthogonal to the step direction. We formulate a positive strength function that is large for strong ridges by considering the negated reciprocal of the expression used by Haralick and Schultz et al: $-\lambda_2/|\nabla f|$. This is large when λ_2 is very negative (height is clearly concave-down, cutting across the ridge) and $|\nabla f|$ is small (the ridge does not run along the gradient of a steep hill). To gracefully handle $|\nabla f| \approx 0$, and to provide additional control over what downward concavity is significant, we propose strength measures for ridge and valley surfaces and lines:

$$\mathbf{r}_{rs} = \frac{-\lambda_2}{r_0 + |\nabla f|}; \mathbf{r}_{rl} = \frac{-\lambda_1}{r_0 + |\nabla f|}; \mathbf{r}_{vs} = \frac{\lambda_0}{r_0 + |\nabla f|}; \mathbf{r}_{vl} = \frac{\lambda_1}{r_0 + |\nabla f|} \quad (20)$$

where *rs*, *rl*, *vs*, and *vl* mean the same as in (9), and $r_0 > 0$ is the *feature strength bias*. For a given (positive) feature strength threshold, increasing bias r_0 creates a more stringent test on the negativity (ridges) or positivity (valleys) of the relevant eigenvalue.

We adapt the above measures for other extremal surfaces, with

$$\mathbf{r}_{-s} = \frac{f''}{r_0 + |\nabla f|}, \quad \mathbf{r}_{+s} = \frac{-f''}{r_0 + |\nabla f|}, \quad (21)$$

where $\mathbf{c}$ is the direction (as above) along which the extremum is sought, and $f'' = \mathbf{c} \cdot (\nabla \otimes \nabla f)\mathbf{c}$ is the second directional derivative of f along $\mathbf{c}$. For extremal lines, again consider H (16), the Hessian of f projected to the plane within which the extremum is sought. We propose strength measures for minimal line ("$-l$") and maximal line ("$+l$") features:

$$\mathbf{r}_{-l} = \frac{\rho_0}{r_0 + |\nabla f|}, \quad \mathbf{r}_{+l} = \frac{-\rho_1}{r_0 + |\nabla f|}, \quad (22)$$

where $\rho_0 \geq \rho_1$ are the eigenvalues of H.

```
 1 input vec3 camEye ("Camera look-from point"); // look-at = [0,0,0]
 2 input real camDepth ("Distance between near,far clip planes");
 3 input real camFOV ("Vertical angle subtended by image");
 4 input int imgRes ("Resolution on edge of square output image");
 5 input real rayStep ("Sampling distance on central ray");
 6 input real thick ("Apparent thickness of isosurface");
 7 input real v0 ("which isosurface to render");
 8 input image(3)[] vol ("data to render");
 9 field#2(3)[] F = bspln3 ⊛ vol; // convolve image w/ recon kernel
10 // Only these feature functions are specific to isosurfaces
11 function vec3 fStep(vec3 x) = (v0 - F(x))*∇F(x)/(∇F(x)•∇F(x));
12 function real fStrength(vec3 x) = |∇F(x)|;
13 // Computing ray parameters and view-space basis
14 vec3 camN = normalize(-camEye);            // N: away from eye
15 vec3 camU = normalize(camN × [0,0,1]); // U: right
16 vec3 camV = camN × camU;                   // V: down
17 real camNear = |camEye| - camDepth/2;  // near clip, view space
18 real camFar  = |camEye| + camDepth/2;  // far clip, view space
19 // Core opacity function is a capped tent function
20 function real atent(real d) = clamp(0, 1, 1.5*(1 - |d|/thick));
21 // Renders ray through (rayU,rayV) on view plane through origin
22 strand ray(int ui, int vi) {
23   real UVmax = tan(camFOV*π/360)*|camEye|;
24   real rayU = lerp(-UVmax, UVmax, -0.5, ui, imgRes-0.5);
25   real rayV = lerp(-UVmax, UVmax, -0.5, vi, imgRes-0.5);
26   vec3 rayVec = camN + (rayU*camU + rayV*camV)/|camEye|;
27   real rayN = camNear - rayStep;   // init ray position
28   output vec4 rgba = [0,0,0,0];    // output ray color
29   real gray = 0;                   // ray grayscale
30   real tt = 1;                     // ray tranparency
31   update {
32     rayN += rayStep;       // increment ray position
33     if (rayN > camFar) {  // done if ray passed far plane
34       real q = 1-tt if tt < 1 else 1; // un-pre-multiply
35       rgba = [gray/q, gray/q, gray/q, 1-tt];
36       stabilize;
37     }
38     vec3 pos = camEye + rayN*rayVec;  // ray sample position
39     if (!inside(pos,F) || fStrength(pos) == 0) {
40       continue; // neither in field nor possibly near feature
41     }
42     vec3 step = fStep(pos);          // step towards feature
43     real aa = atent(|step|);         // sample opacity
44     if (aa == 0) { continue; }       // skip if no opacity
45     real gg = (normalize(step)•[0,0,1])^2; // 2-sided lighting
46     gray += tt*aa*((0.2 + 0.8*gg)); // ambient and diffuse
47     tt *= 1 - aa;                    // tranparencies multiply
48   }
49 } // end strand
50 initially [ray(ui,vi) | vi in 0..imgRes-1, ui in 0..imgRes-1];
```

Figure 2: *A minimal but complete volume renderer is made specific to isosurfaces only by* `fStep` *and* `fStrength` *on lines 11 and 12.*

Additional tests may help include or exclude part of a feature, such as requiring that ridges have sufficient *height* as well as sufficient strength. In particular, when extracting Sujudi-Haimes vortex cores [SH94] by the Parallel Vectors operator with ridge or valley lines of $h = (\mathbf{v}/|\mathbf{v}|) \cdot (\nabla \otimes \mathbf{v}/|\nabla \otimes \mathbf{v}|)$, we want h to be near ± 1. Our approach includes a *feature mask* function to be set and thresholded to afford this extra control as needed.

4. Methods

We implement our approach in Diderot, a domain-specific language for scientific visualization [CKR*12, KCS*16, CKR18]. We describe two Diderot programs, for volume rendering (Sec. 4.1) and for particle-based feature sampling (Sec. 4.2), and describe how each may be specialized with feature functions. Code for the features functions is listed in the context of volume rendering, but then re-used verbatim for particle-based feature sampling.

4.1. Direct Volume Rendering

The program in Fig. 2 volume renders isosurfaces and demonstrates the basic structure of Diderot programs. Input variables (lines 1–8) include rendering parameters, the isovalue, and the image data from which the C^2-continuous field `F` is created by convolving with the cubic B-spline (line 9). After computing ray and camera geometry (lines 14–18), the `atent` function (line 20) implements a modified $\alpha_{\text{tent}}(d)$ (4), parameterized by isosurface thickness `thick`. The `ray` strand (the unit of parallelism in Diderot) starting line 22 renders one ray. After computing ray geometry (lines 23–26), lines 27–30 initialize ray state, and the **`update`** method (starting line 31) implements one iteration of ray traversal and volume sampling, as explained in the code comments. The program ends (line 50) by creating an array of strands to be executed in parallel. Figure 3a shows how this program renders a small $64 \times 64 \times 32$ synthetic dataset (used throughout this section) containing a Möbius strip with seven Gaussian blobs along its circular core.

The Fig. 2 renderer demonstrates how feature specificity can be isolated to a few functions, in this case `fStep` and `fStrength`, on lines 11 and 12. The `fStep(x)` function, directly copied from $\mathbf{s}_{\text{iso}}(\mathbf{x})$ (3) of Sec. 3, determines opacity (line 43) and shading (line 45) based on `step = fStep(pos)` (line 42). The `fStrength` function (line 39) is used here to avoid divide-by-zero problems when computing feature steps.

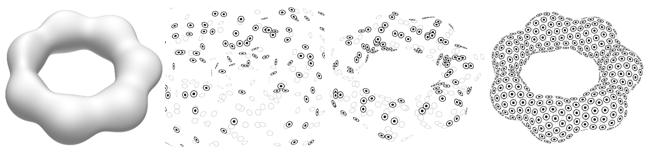

(a) *Rendered* **(b)** *iteration 0* **(c)** *iteration 5* **(d)** *iteration 55*

Figure 3: *Minimal but complete program results: Fig. 2 volume renderer creates (a), Fig. 4 particle system creates (b), (c), and (d; converged) after indicated iterations.*

Results in Sec. 5 come from a more complete volume renderer, listed in Appendix A. Compared to Fig. 2, this program renders in color (with a univariate colormap of the underlying scalar value) and it offers more control over ray geometry and rendered appearance, but it too hinges on the same `fStep` and `fStrength` functions. The top row of Fig. 1 shows a variety of features all volume rendered from the same Möbius strip synthetic dataset used in Fig. 3a, all created with the program in Appendix A (used without change for Fig. 1a). Copying (8) from Sec. 3, we can change `fStep` and `fStrength` to show critical points instead:

```
function vec3 fStep(vec3 x) = // critical points
  -inv(∇⊗∇F(x))•∇F(x);
function real fStrength(vec3 x) = |∇⊗∇F(x)|;
```

This produces Fig. 1b. The image clarity benefits from `fStrength` (the Frobenius norm of the Hessian) and a user-defined threshold to give opacity only to critical points near significant second-order variation. Direct volume rendering offers visual feedback to help determine such thresholds. For consistency one colormap of scalar data value is used for all renderings in Fig. 1, which for this synthetic dataset clearly distinguishes in Fig. 1(b) between maxima (blue) and saddle points (orange).

The following implements $\mathbf{s}_{rs}$ and $\mathbf{s}_{rl}$ for ridge surfaces and lines (Fig. 1(c) and (e)), which depends on Hessian eigensystems.

```
 1 input real rad ("Inter-particle potential radius");
 2 input real eps ("General convergence threshold");
 3 input real v0 ("Which isosurface to sample");
 4 input vec3{} ipos ("Initial point positions");
 5 input image(3)[] vol ("Data to analyze");
 6 field#2(3)[] F = bspln3 ⊛ clamp(vol); // convolve w/ recon kernel
 7 // Only these three "f" functions are specific to isosurfaces
 8 function vec3 fStep(vec3 x) = (v0 - F(x))*∇F(x)/(∇F(x)•∇F(x));
 9 function tensor[3,3] fPerp(vec3 x) {
10   vec3 norm = normalize(∇F(x));
11   return identity[3] - norm⊗norm;
12 }
13 function real fStrength(vec3 x) = |∇F(x)|;
14 function real phi(real r) = (1 - r)^4;  // univariate potential
15 function real phi'(real r) = -4*(1 - r)^3;
16 function real enr(vec3 x) = phi(|x|/rad);
17 function vec3 frc(vec3 x) = phi'(|x|/rad) * (1/rad) * x/|x|;
18 // Strands first find feature, then interact w/ or make neighbors
19 strand point (vec3 pos0, real hh0) {
20   output vec3 pos = pos0;    // current particle position
21   real hh = hh0;             // energy gradient descent stepsize
22   vec3 step = [0,0,0];       // energy+feature steps this iter
23   bool found = false;        // whether feature has been found
24   int nfs = 0;               // number feature steps taken
25   update {
26     if (!inside(pos, F) || fStrength(pos) == 0) {
27       die; // not in field domain & not possibly near feature
28     }
29     if (!found) {                  // looking for feature
30       step = fStep(pos);           // one step towards feature
31       pos += step;
32       if (|step|/rad > eps) {      // took a substantial step
33         nfs += 1;
34         if (nfs > 10) { die; }     // too slow to converge
35       } else { found = true; }     // else converged on feature
36     } else {       // feature found; interact with other points
37       pos += fStep(pos);           // refine feature sampling
38       step = [0,0,0];              // initialize output step
39       real oldE = 0;               // energy at current location
40       vec3 force = [0,0,0];        // force on me from neighbors
41       int nn = 0;                  // number of neighbors
42       foreach (point P in sphere(rad)) {
43         oldE += enr(P.pos - pos);
44         force += frc(P.pos - pos);
45         nn += 1;
46       }
47       if (0 == nn) {                  // no neighbors, so create one
48         new point(pos + [0.5*rad,0,0], hh);
49         continue;
50       }                               // else interact w/ neighbors
51       force = fPerp(pos)•force;       // no force perp. to fStep(pos)
52       vec3 es = hh*force;             // energy step along force
53       if (|es| > rad) {               // limit motion to radius
54         hh *= rad/|es|;               // decrease stepsize and step
55         es *= rad/|es|;
56       }                               // now |es| <= rad
57       vec3 fs = fStep(pos+es);        // find step towards feature
58       if (|fs|/|es| > 0.5) {          // feature step too big
59         hh *= 0.5;                    // try again w/ smaller step
60         continue;
61       }
62       vec3 oldpos = pos;
63       pos += fs + es;                 // take steps, find new energy
64       real newE = sum { enr(pos - P.pos) | P in sphere(rad) };
65       if (newE - oldE > 0.5*(pos - oldpos)•(-force)) {
66         pos = oldpos;                 // energy didn't go down enough;
67         hh *= 0.5;                    // try again w/ smaller step
68         continue;
69       }
70       hh *= 1.1;                      // cautiously increase stepsize
71       step = fs + es;                 // record steps taken
72       if (nn < 5) {                   // add neighbor if have too few
73         new point(pos + 0.5*rad*normalize(es), hh);
74       }
75     } // else found
76   } // update
77 }
78 global {
79   bool allfound = all { P.found | P in point.all};
80   real maxstep = max { |P.step| | P in point.all };
81   if (allfound && maxstep/rad < eps) { stabilize; }
82 }
83 initially { point(ipos[ii], 1) | ii in 0 .. length(ipos)-1 };
```

Figure 4: *A minimal but complete surface feature sampler is made specific to isosurfaces only by three feature functions starting line 8.*

```
function vec3 fStep(vec3 x) {      // ridge surfaces
  vec3{3} E = evecs(∇⊗∇F(x));
  real{3} L = evals(∇⊗∇F(x));
  return -(1/L{2})*E{2}⊗E{2}•∇F(x);
}
function real fStrength(vec3 x) = // ridge surfaces
  -evals(∇⊗∇F(x)){2}/(fBias + |∇F(x)|);
function vec3 fStep(vec3 x) {      // ridge lines
  vec3{3} E = evecs(∇⊗∇F(x));
  real{3} L = evals(∇⊗∇F(x));
  return -(E{2}⊗E{2}/L{2} + E{1}⊗E{1}/L{1})•∇F(x);
}
function real fStrength(vec3 x) = // ridge lines
  -evals(∇⊗∇F(x)){1}/(fBias + |∇F(x)|);
```

In both cases, `fStep` and `fStrength` transcribe the mathematical definitions in (13), (12), and (20), using the sequence (indexed by `{}`) of Hessian eigenvectors `E` and eigenvalues `L`. Examples of `fStep` and `fStrength` functions for other extremal features (Sec. 3.2) are given with Results (Sec. 5).

4.2. Particle-based Feature Sampling

Being less common than direct volume rendering in visualization, the mechanics of particle systems and their implementation merit more detailed explanation. The Diderot program in Fig. 4 uses energy minimization to uniformly sample an isosurface. Similar to the volume renderer in Fig. 2, this uses functions `fStep`, `fPerp`, and `fStrength` (starting line 8) to isolate its specificity to isosurface features; the rest of the program is invariant with respect to the type of surface feature. The `fPerp` function returns a projection onto the space perpendicular to locally possible `fSteps`.

Based on a univariate potential energy function `phi` and its derivative (lines 14 and 15), functions `enr` and `frc` (lines 16 and 17) give the energy and force due to a particle at offset `x`, where the potential energy profile around each particle has circular support with radius `rad` (line 1). Each strand (line 19) computes the position of one particle, initialized with an initial set of points (lines 4 and 83, created by a pre-process to randomly sample the volume domain), and then updated through two stages of computation. In the first stage, while `!found` (line 29), the particle is transported onto the feature by successive applications of the `fStep` function, one step per iteration, until the step size is small enough to imply convergence, at which point `found` is set to **`true`** (line 35). In the second stage (line 36), each iteration computes one step of gradient descent through the potential energy created by neighboring particles (if a particle has no neighbors it creates one; line 47). This involves learning, at the current particle location `pos`, the energy and force due to neighbors (line 42), projecting out the force component aligned with `fStep` (line 51), taking a step along force (line 52), moving back to the feature with `fStep` (line 57), and then learning the energy at the new location (line 64). The comparison of feature step length `|fs|` and energy-reducing step length `|es|` on line 58 ensures that only small steps are needed to get back to the feature, after moving to reduce energy.

Finally, line 65 tests the Armijo condition for sufficient decrease [Arm66]. In a conservative field, force is the negative gradient of potential energy, so the dot product of `-force` with the change in position `(pos - oldpos)` should predict the change in potential `(E - oldE)`, which should be negative. The Armijo condition is that the actual decrease be at least some fraction of the prediction; line 65 uses fraction `0.5`. If the condition is not met, the search backtracks with a smaller step size (line 66). Otherwise, the step size is slightly increased (line 70, which speeds convergence), and the number of particles in the system is adjusted. Like the individual particle motion, the determination of whether to add new particles is made locally, by each particle, based on its number of neighbors: a new particle is added if the number of neighbors is less than five (line 72). The system as a whole stabilizes (line 81) when all strands have located the feature (line 79) and no particle has undergone significant motion (line 80). Results from this program are illustrated in Fig 3b, 3c, and 3d.

The remaining particle system results, on synthetic data here and on real-world data in Sec. 5, use a more complex Diderot program, shown in Appendix B. Its basic structure is the same as in Fig. 4, but it has three kinds of improvements: more controls on system behavior, a more demanding test of system convergence, and more robust handling of population control (and the ability to sample 0D, 1D, and 2D features in 3D fields), via an inter-particle potential function with a slight minimum. The increased system controls include the periodicity of attempts at population control (Fig. 4 does this at every iteration), a scaling of `fStep` to stabilize feature sampling, and separate convergence thresholds for feature sampling and for particle motion (Fig. 4 uses the same `eps` on lines 32 and 81).

The system convergence of the Appendix B program demands that all particles have `found` the feature, like the minimal program above. Rather than simply requiring no significant motion in the last iteration, however, it averages motion over all recent iterations, and demands that all particles have been consistently slow-moving, and that none died or birthed new particles recently. The geometric uniformity of the sampling is measured with the coefficient-of-variation of each particle's distance to its closest neighbor. The coefficient-of-variation (CV) is a dimensionless measure of dispersion computed as the standard deviation divided by the mean. A low CV of closest neighbor distance implies particle system uniformity. Some of these considerations are from earlier particle system work [KESW09], but others (such as CV-based convergence testing) are incremental improvements.

Better handling of particle system population (the number of particles) is required for reliable particle-based feature extraction in cases other than simple synthetic data. The system population is locally controlled according to each particle's number of neighbors, based on a minimum nn_{min} and maximum nn_{max} target number of neighbors. This permits handling 2D (surface) features, with $(nn_{min}, nn_{max}) = (6,8)$, versus 1D (curve) features, with $(nn_{min}, nn_{max}) = (2,3)$. To attain the target neighbor number range, particles add new neighbors with **`new`** (to build up the sampling of a feature that was only sparsely sampled by initialization), or remove themselves with **`die`** (to remove points from needlessly dense samplings). Particles next to the boundary of a feature, where feature strength falls below a threshold, will be missing some neighbors, and will thus try to create new neighbors. These new particles will be immediately removed, however, due to low `fStrength` (as in line 26 of Fig. 4) To avoid a stream of short-lived particles dying at feature boundaries, each particle maintains a count of how many new neighbors it has created, capped at nn_{max}.

A final conceptual difference between the simplistic particle program in Fig. 4 and the robust one in Appendix B is the sampling of features with boundaries. While inter-particle repulsion is adequate for most isosurfaces (as in Fig. 3d), problems arise when the isosurface is not entirely contained by the volume domain, or when the feature is not a closed surface, as with the open edge of a ridge surface where the feature strength falls below a threshold. In these cases, simple repulsion will force particles off the feature, failing to maintain a regular hexagonal sampling on surface features. Based on previous work [KESW09], we use a potential function $\phi(r)$ with a slight negative potential well to create a specific preferred inter-particle distance. Current work has adopted the $\phi(r)$ shown in (Appendix B) Fig. 11, a C^3-continuous function from $\phi(0) = 1$ to $\phi(r) = 0$ for $r \geq 1$, with a minima (well) at $\phi(2/3) = -0.001$.

4.3. Particle system post-processing

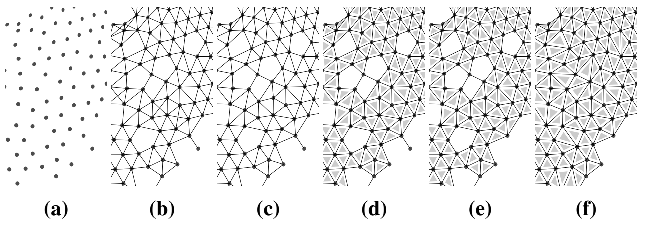


Figure 5: *Particles on surface features (a) are meshed by adding inter-particle edges (b), uncrossing edges (c), adding triangles (d), fixing stray edges (e), and filling holes (f).*

After computing a particle system to sample a surface feature, meshing the resulting points may be more useful for down-stream processing or visualization than the point locations themselves. We believe the high spatial regularity at convergence of our particle systems, combined with erring on the side of high sampling density, simplifies the meshing task. Our simple meshing approach seems adequate from our limited experience, though it would benefit from computational geometry analysis. Fig 5 illustrates the approach for an intermediate (non-converged) configuration with less regularity (Fig 5a). We first find edges for all inter-particle interactions (i.e. within each other's potential wells, Fig 5b), using a short utility Diderot program described in Appendix D.2. Subsequent steps are implemented in about 700 lines of C code. We find crossing edge pairs, remove the longer edge in each pair (Fig 5c), add triangles for length-3 edge loops (Fig 5d), and remove stray edges (Fig 5e). Short edge loops (length 4 or 5) with only one triangle per edge are holes, which we fill by adding minimal-length internal edges and associated triangles. This approach produced the meshes seen in the lower row of Fig 1. With particles sampling 1D features, we simply add edges for inter-particle interactions and sort vertices into polylines paths.

5. Results

These results were produced with the volume rendering and particle system programs in Appendices A and B, modified for each case by defining the field F of interest, and selecting the feature functions for the feature type of interest. The code for the feature functions are shared between the two programs. All results use $6 \times 6 \times 6$ support C^4-continuous reconstruction kernels available in Diderot, either `c4hexic` (which exactly reconstructs cubics), or the quintic B-spline `bspln5` (which smooths more).

5.1. Cell Nuclei as Local Maxima

Microscopic volumetric imaging often involves manipulating specimens so cell nuclei fluoresce [Str07]; maximum intensity projection (MIP) can then visualize the location of nuclei [LZP12]. Fig. 6 illustrates this with data acquired as part of research on neuron migration in zebrafish development [WP13]. These transgenic zebrafish express red fluorescent protein in the cell membranes of neurons of particular interest [MWRP10], shown as an isosurface in Fig. 6a, in addition to green fluorescent protein in all nuclei [PGVCO01]. In Fig. 6b, the MIP is computed only within the interior of the neurons of interest, which clarifies the result somewhat. The individual nuclei remain hard to discern, however. Fig. 6c renders the *local maxima only*, using the critical point step function (8), so that the number, location, and relative position of the nuclei become clear, as each is rendered as a shaded and roughly spherical blob. This is not possible with isosurfaces: the local maxima for each nuclei occur at widely varying intensities, so isosurfacing could not consistently reveal them all, nor create a spherical shape from maxima with different (negative) Hessian eigenvalues.

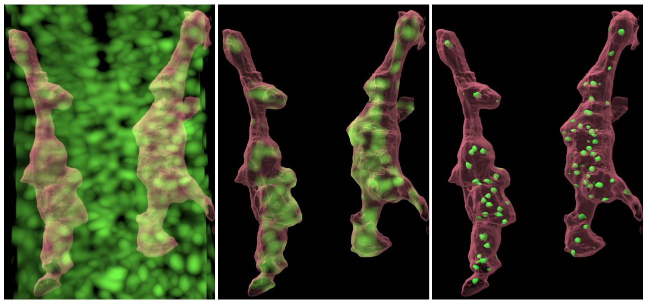

(a) *MIP* **(b)** *MIP within iso.* **(c)** *Maxima within iso.*

Figure 6: *Green cell nucleii can be traditionally rendered with MIP (a), or with MIP computed only within the membrane of cells of interest (b). Maxima rendering (c) clarifies nucleii number and depth.*

5.2. Ridge Surface Boundary Curve

While it is common in practice to bound the extent of ridge surface features by thresholding their feature strength, they also have an intrinsic boundary at certain degeneracies of the Hessian, where its two smallest eigenvalues become equal, as described by Schultz et al. [STS10]. They localize degeneracies on voxel faces with Newton steps towards the simultaneous roots of a set of discriminant functions of the Hessian components [ZP04], and then connect a polyline of degenerate Hessians across faces. Our approach starts by defining a new scalar field as the *mode* of the Hessian [CHDH00] ($+1$ when two small eigenvalues are equal, and -1 when two large eigenvalues are equal). We copy formulae from [CHDH00] to create a scalar field F of Hessian mode:

```
field#4[] F0 = c4hexic ⊛ img; // img == data volume
field#2(3)[3,3] E = ∇⊗∇F0 - identity[3]*trace(∇⊗∇F0)/3;
field#2(3)[] F = 3*sqrt(6)*det(E/|E|);
```

Then ridge lines of mode F, where it is close to 1, are the degenerate lines of interest. Fig 7 illustrates with a synthetic dataset involving three connected twisting sheets. The resolution and orientation of the grid gives rise to a complex configuration of two sheets joining, and one terminating at a degenerate line. We may easily volume render the degenerate line in isolation as a ridge line of mode (Fig 7d), or explicitly sample it with a particle system, and render the particles in context (Fig 7e,f)). The feature step `fStep` function used to render or sample the degenerate line is the same as for ridge lines in Sec. 4.1; only the definition (above) of scalar field F differs. While the previous Newton-step approach for isolating these features used trilinear interpolation of third derivatives [STS10, ZP04], ours requires fourth derivatives (Hessian of the mode of the Hessian), but with significantly smaller implementation complexity, and the ability to work with high-quality reconstructions.

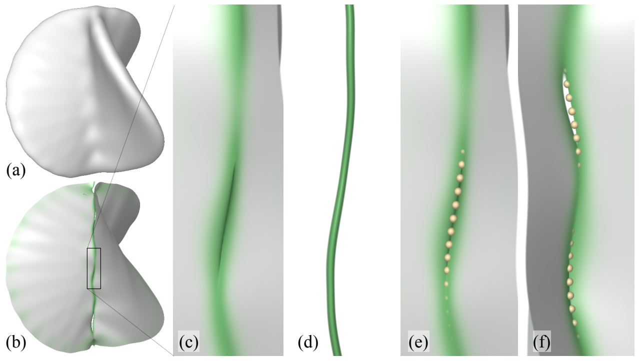


Figure 7: *A three-way intersection of sheets (a) creates a complex configuration of ridge surfaces (b), seen closer in (c). The analytic ridge boundary can be rendered in isolation (d), and regularly sampled with a particle system (e); different view (f).*

5.3. Crease Lines on Implicit Surfaces

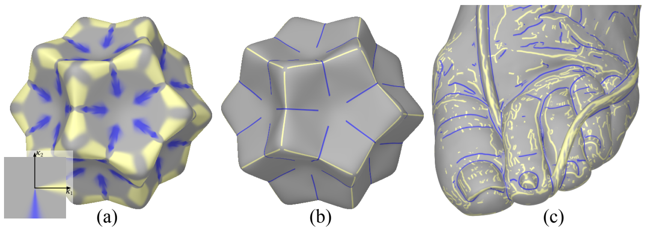


Figure 8: *Curvature-based transfer functions do a poor job (a) of isolating surface crease lines, but feature steps toward curvature extrema permit their clean rendering (b), (c).*

While volume rendering with curvature-based transfer functions can highlight the convex and concave portions of implicit surfaces [HKG00, KWTM03], Fig 8a illustrates that they do a poor job of capturing surface creases (Sec 3.2) on a synthetic volume of a dodecahedron with indented faces and edges of varying sharpness. To render crease lines properly, we create scalar fields `K1` and `K2` of curvature, based on formulae in [KWTM03]:

```
field#3(3)[3] N = -∇F/|∇F|; // F=C^4 scalar field
field#3(3)[3,3] P = identity[3] - N⊗N; // evals=1,1,0
field#2(3)[3,3] G = P•(∇⊗N)•P;          // evals=k1,k2,0
field#2(3)[] discrim = sqrt(2*G:G - trace(G)*trace(G));
field#2(3)[] K1 = (trace(G) + discrim)/2;
field#2(3)[] K2 = (trace(G) - discrim)/2;
field#2(3)[] KT = sqrt(K1^2 + K2^2);
```

Rendering crease lines does not require creating a new function to `fStep` in 3D towards the curvature extrema, but we do require the length of that step. Outward creases (maxima of κ_1 along $\mathbf{d}_1$) are rendered by setting material color `mcol` via:

```
1  if (KT(pos) < fMaskTh) {
2    mcol = gray; // low total curvature; uninteresting
3  } else {
4    vec3 d1 = col1span(G(pos) - K2(pos)*P(pos));
5    real k1' = d1•∇K1(pos);
6    real k1'' = d1•(∇ ⊗ ∇K1(pos))•d1;
7    mcol = lerp(gray, rcol, atent(1,|k1'/k1''|/3));
8  }
```

where `gray` and `rcol` are the colors for gray and the outward crease color; the code for inward crease rendering is essentially the same. The tensor `G(pos)-K2(pos)*P(pos)` on line 4 has eigenvalues $\kappa_1, 0, 0$; utility function `col1span` (App. D.1) returns the eigenvector (i.e. the curvature direction $\mathbf{d}_1$) for κ_1. From (15), the crease line feature step length is `|k1'/k1''|`, used on line 7 with the same `atent` of Fig. 2 line 20 to effect the crease line coloring, shown in Fig 8b. Results on a foot of the Visible Human female CT scan are in Fig 8c. Some creases indicate skin folds and toenail edges, while others demarcate the toenails, and the toes themselves. The ease of creating these renderings, which depend on fourth derivatives (Hessian of curvature), is notable.

5.4. Vortex Cores in 3D Flow

Vortex cores remain an important target for scientific visualization and feature extraction, because of the variety of possible definitions of cores. We consider two possible vortex core definitions.

The second principal invariant (the coefficient of quadratic term of the characteristic equation) of the Jacobian $\mathbf{J} = \nabla \otimes \mathbf{v}$ of a vector field $\mathbf{v}$, is (c.f. (6) of [CPC90]):

$$Q = \frac{\mathrm{tr}(\mathbf{J})^2 - \mathrm{tr}(\mathbf{J} \cdot \mathbf{J})}{2} \tag{23}$$

If $\mathbf{\Omega} = (\mathbf{J} - \mathbf{J}^T)/2$ and $\mathbf{S} = (\mathbf{J} + \mathbf{J}^T)/2$ are the anti-symmetric (vorticity tensor) and symmetric (rate-of-strain tensor) components of $\mathbf{J}$, then $Q = (|\mathbf{\Omega}|^2 - |\mathbf{S}|^2)/2 > 0$ implies that vorticity dominates strain [JH95, HAL05]. The Q-criterion identifies vortices with regions of high Q [HWM88]. We express a differentiable field of Q in Diderot by copying (23) with

```
field#4[] V = c4hexic ⊛ img; // img == data volume
field#3(3)[3,3] J = ∇⊗V;
field#3(3)[] F = (trace(J)^2 - trace(J•J))/2;
```

and visualize and extracting *ridge lines* of `F`$= Q$ with the feature functions previously described.

Alternatively, the Sujudi-Haimes condition identifies vortex cores as points where the flow direction is the sole real eigenvector of $\mathbf{J}$ [SH94]. This is equivalent to saying that $\mathbf{v}$ and $\boldsymbol{J}\boldsymbol{v}$ are parallel, and that the discriminant of $\mathbf{J}$ is negative (i.e. there are two complex conjugate eigenvalues) [RP96]. In Diderot:

```
field#2(3)[] F = (V/|V|) • (J•V/|J•V|);
field#2(3)[] A = -trace(J);
field#2(3)[] B = (trace(J)^2 - trace(J•J))/2; // == Q
field#2(3)[] C = -det(J);
field#2(3)[] discrim = A^2*B^2 - 4*B^3 \
                     - 4*A^3*C + 18*A*B*C - 27*C^2;
function real fMask(vec3 x) = -discrim(x);
```

Our approach here is a literal interpretation of the Parallel Vectors operator applied to $\mathbf{v}$ and $\boldsymbol{J}\boldsymbol{v}$: we could seek the ridge and valley lines of `F` at `F`$= +1$ and `F`$= -1$, respectively, with the additional requirement that `fMask` > 0 [PR99].

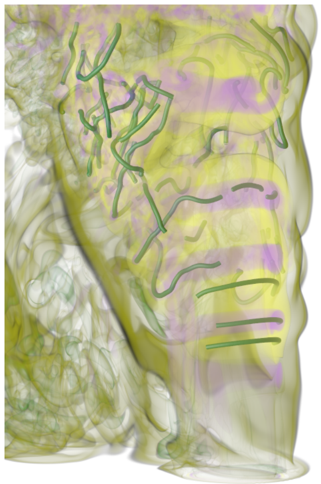


Figure 9: *Volume rendering of Q isosurfaces around extracted polylines of Q ridge lines*

Fig. 9 visualizes a small region of a turbulent reacting jet-in-crossflow simulation [GGK*12]. As described in [BGB*16], the complex variety of vortex structures merit study with dedicated flow analysis tools. Our goal was to see if a direct first-principles approach could produce something plausible. The rendering in Fig. 9 shows translucent isosurfaces of Q volume rendered by OSPRay, and color coded by the sign of Q. Within the volume rendering are tubes around polylines extracted from our particle-based feature extraction of the ridge lines of Q (Sec. 4.3). This demonstrates one value of our explicit feature extraction: integration with other visualization and analysis tools.

Our work on Parallel Vector Operator (PVO) features produced novel formulae for the gradient and Hessian of the dot product $(\mathbf{a}/|\mathbf{a}|) \cdot (\mathbf{b}/|\mathbf{b}|)$ between two vector fields $\mathbf{a}$ and $\mathbf{b}$, normalized. For Sujudi-Haimes vortex cores of vector field $\mathbf{v}$, $\mathbf{a} = \mathbf{v}$ and $\mathbf{b} = (\nabla \otimes \mathbf{v})\mathbf{v}$. As described in Appendix C, we modified the Diderot compiler to produce human-readable expressions of the intermediate representations used to compute PVO, which are compared to the PVsolve formulae of Van Gelder and Pang [GP09].

5.5. DTI Anisotropy Extremal Features

Scalar fields of tensor invariants provide a rich vocabulary for defining and exploring various possible features. Diffusion tensor image (DTI) visualization can use fractional anisotropy (FA) [Bas95] to highlight nervous tissue white matter. Our previous work [KCS*16] used Diderot to create a scalar field `F` of

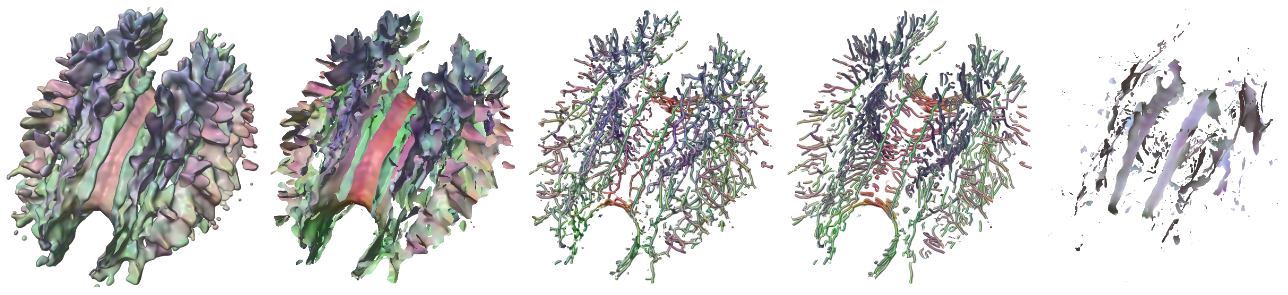

(a) *Isosurface* **(b)** *Ridge Surface* **(c)** *Ridge Line* **(d)** *Max Line* **(e)** *Min Surface*

Figure 10: *Different feature step functions reveal a variety of Fractional Anisotropy (FA) features from a DTI volume.*

FA, and render its isosurface, as in Fig. 10a (showing a human brain DTI from above, with a near cutting plane). With our new approach, Fig. 10b shows an FA ridge surface obtained by swapping out the isosurface `fStep` function (Fig. 2 line 11) with the one for ridge surfaces (Sec. 4.1). With another feature step function, Fig. 10c shows FA ridge lines [TKW08]. Fig. 10d shows, on the other hand, the *maximal lines of FA with respect to the medium and minor eigenvector of the diffusion tensor itself* (not the eigenvectors of the FA Hessian), using (18) and (22). With strength thresholds set to generate comparably dense visualizations, FA maximal lines (Fig. 10d) seem to have better coherent organization than FA ridge lines (Fig. 10c). Since both capture the cingulum bundles (two diagonal green paths near center), FA maximal lines merit further exploration as neuroanatomical markers. Fig. 10e shows another experimental DTI feature, using (15) and (21) to render surfaces of *minimal* FA with respect to the diffusion tensor minor eigenvector (spaning adjacent but orthogonal white matter paths [KTW07]), colored by the minor eigenvector orientation.

6. Discussion, Conclusions, Future Work

Our new approach unifies, both in mathematical formulation and in source code, the tasks of volume rendering and geometrically extracting features from 3D fields. These are complementary tasks: one can quickly volume render to ascertain the presence of features and set appropriate feature strength and feature mask (Sec. 3.3) thresholds, before running feature extraction and meshing. Using the exact same code to describe features for both tasks increases trust in the results, and legible code that mirrors math notation simplifies exploring new features. Our results can be integrated with other tools by saving extracted features to files read by other visualization tools (as in Fig. 9), or by linking a Diderot program, compiled to a library [KCS*16], into another application that controls its execution.

Our focus has been establishing a new way of implementing scientific visualization research, to address the bottleneck of human implementation time, rather than computational execution time. Still, the volume rendering and particle system programs can both execute in parallel, which makes them workable with sufficient cores, but we did not record any specific measurements of compile time, execution time, or parallel speedup. On a modern Mac laptop with 8 cores, most programs compiled in under 20 seconds; programs for particles in vector or tensor fields data took several minutes. Execution times ranged from under 5 seconds for scalar volume renderings, to several minutes for the more complex particle systems. One drawback is that any change to the field or feature functions requires recompiling the entire program. Some dynamic linking of feature-specific modules may be possible, but it would complicate the Diderot compiler's whole-program optimization.

Though the volume rendering and feature sampling programs are mature enough to work on real data, they favor implementation simplicity over computational efficiency or algorithmic sophistication. Volume rendering would be more efficient with adaptive sampling of rays, perhaps based on the Lipschitz constants used by Kalra and Barr [KB89]. The particle system currently makes no effort to vary density with feature curvature [MGW05], or to discover the optimal feature scale [KESW09]. With a meshing strategy more sophisticated than that of Sec. 4.3, the stringent convergence tests and ensuing computational expense of our particles could be reduced; at the opposite extreme of this are methods like AFront which compute vertices and meshing in one sweep [SSS06].

Except for the C code for particle system meshing (Sec. 4.3), all the code for generating our results is in Diderot, and nearly all of it is included in this manuscript and its Appendices. While mature visualization applications will likely require connections to other libraries and interfaces, we are holding ourselves to a standard for self-contained reproducibility that we believe may cultivate research interest in scientific visualization by simplifying how readers may explore and experiment with new methods.

7. Acknowledgments

We thank our collaborators Ana Beiriger and Prof. Victoria Prince for sharing the microscopy data used in Fig. 6. CT data in Fig. 8c courtesy of the NIH NLM Visible Human Project. Brain DTI data in Fig. 10 is courtesy of National Alliance for Medical Image Computing (NAMIC) and is available at `http://hdl.handle.net/1926/1687`. This work was supported by NSF grants CCF-1564298 and IOS-1555972.

References

[AK04] Amenta N., Kil Y. J.: Defining point-set surfaces. In *Computer Graphics (Proc. ACM SIGGRAPH) 2004* (2004), pp. 264–270. 2

[Arm66] Armijo L.: Minimization of functions having Lipschitz continuous first partial derivatives. *Pac. J. Math. 16*, 1 (1966), 1–3. 6

[Bas95] Basser P. J.: Inferring microstructural features and the physiological state of tissues from diffusion-weighted images. *Nuclear Magnetic Resonance in Biomedicine 8* (1995), 333–344. 9

[BAT11] BARAKAT S., ANDRYSCO N., TRICOCHE X.: Fast extraction of high-quality crease surfaces for visual analysis. *Computer Graphics Forum 30*, 3 (2011), 961–970. 2

[BGB*16] BREMER P.-T., GRUBER A., BENNETT J. C., GYULASSY A., KOLLA H., CHEN J. H., GROUT R. W.: Identifying turbulent structures through topological segmentation. *Comm. in Applied Mathematics and Computational Science 11*, 1 (2016), 37–53. 9

[BT10] BARAKAT S., TRICOCHE X.: An image-based approach to interactive crease extraction and rendering. *Procedia Computer Science 1*, 1 (2010), 1709 – 1718. ICCS 2010. 2

[CA97] CROSSNO P., ANGEL E.: Isosurface extraction using particle systems. In *Proc. IEEE Visualization* (1997), pp. 495–498. 3

[Car76] CARMO M. D.: *Differential Geometry of Curves and Surfaces*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976. 4

[CHDH00] CRISCIONE J. C., HUMPHREY J. D., DOUGLAS A. S., HUNTER W. C.: An invariant basis for natural strain which yields orthogonal stress response terms in isotropic hyperelasticity. *Journal of Mechanics and Physics of Solids 48* (2000), 2445–2465. 8

[Chi17] CHIW C.: *Implementing mathematical expressiveness in Diderot*. PhD thesis, University of Chicago, Chicago, IL, May 2017. 2

[CKR*12] CHIW C., KINDLMANN G., REPPY J., SAMUELS L., SELTZER N.: Diderot: A Parallel DSL for image analysis and visualization. In *Proc. SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI)* (June 2012), pp. 111–120. 2, 5

[CKR18] CHIW C., KINDLMANN G. L., REPPY J.: Compiling Diderot: From Tensor Calculus to C. *ArXiv e-prints* (Feb. 2018). arXiv:1802.06504. 2, 5

[CPC90] CHONG M. S., PERRY A. E., CANTWELL B. J.: A general classification of three-dimensional flow fields. *Physics of Fluids A: Fluid Dynamics 2*, 5 (1990), 765–777. 9

[Ebe96] EBERLY D.: *Ridges in Image and Data Analysis*. Kluwer Academic Publishers, 1996. 1, 2, 3, 4

[GGK*12] GROUT R. W., GRUBER A., KOLLA H., BREMER P.-T., BENNETT J. C., GYULASSY A., CHEN J. H.: A direct numerical simulation study of turbulence and flame structure in transverse jets analysed in jet-trajectory based coordinates. *Journal of Fluid Mechanics 706* (2012), 351–383. 9

[GM97] GUY G., MEDIONI G.: Inference of surfaces, 3d curves, and junctions from sparse, noisy, 3D data. *IEEE Transactions on Pattern Analysis and Machine Intelligence 19*, 11 (Nov. 1997), 1265–1277. 2

[GP09] GELDER A. V., PANG A.: Using PVsolve to analyze and locate positions of parallel vectors. *IEEE Transactions on Visualization and Computer Graphics 15*, 4 (July 2009), 682–695. 9, 17

[HAL05] HALLER G.: An objective definition of a vortex. *Journal of Fluid Mechanics 525* (2005), 1–26. 9

[Har83] HARALICK R. M.: Ridges and valleys on digital images. *Computer Vision, Graphics, and Image Processing 22* (1983), 28–38. 2, 3, 4

[HKG00] HLADUVKA J., KÖNIG A., GRÖLLERRÖLLER E.: Curvature-based transfer functions for direct volume rendering. In *Spring Conference on Computer Graphics 2000* (May 2000), vol. 16, pp. 58–65. 8

[HWM88] HUNT J. C. R., WRAY A., MOIN P.: Eddies, stream, and convergence zones in turbulent flows. In *Proceedings of the Summer Program* (1988), Center for Turbulence Research (Stanford University), pp. 193–208. 9

[JH95] JEONG J., HUSSAIN F.: On the identification of a vortex. *Journal of Fluid Mechanics 285* (1995), 69–94. 9

[KB89] KALRA D., BARR A. H.: Guaranteed ray intersections with implicit surfaces. In *Computer Graphics (Proc. ACM SIGGRAPH) 1989* (1989), pp. 297–306. 10

[KCS*16] KINDLMANN G., CHIW C., SELTZER N., SAMUELS L., REPPY J.: Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Trans. on Visualization and Computer Graphics (Proceedings VIS 2015) 22*, 1 (Jan. 2016), 867–876. 2, 5, 9, 10

[KESW09] KINDLMANN G. L., ESTÉPAR R. S. J., SMITH S. M., WESTIN C.-F.: Sampling and visualizing creases with scale-space particles. *IEEE Trans. on Visualization and Computer Graphics 15*, 6 (2009), 1415–1424. 7, 10

[KHW*09] KNOLL A., HIJAZI Y., WESTERTEIGER R., SCHOTT M., HANSEN C., HAGEN H.: Volume ray casting with peak finding and differential sampling. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1571–1578. 2

[KKS*12] KOTAVA N., KNOLL A., SCHOTT M., GARTH C., TRICOCHE X., KESSLER C., COHEN E., HANSEN C. D., PAPKA M. E., HAGEN H.: Volume rendering with multidimensional peak finding. *IEEE Pacific Visualization Symposium 2012, PacificVis 2012 - Proceedings vi* (2012), 161–168. 2

[Koe90] KOENDERINK J. J.: *Solid Shape*. The MIT Press, 1990. 4

[KTW07] KINDLMANN G., TRICOCHE X., WESTIN C.-F.: Delineating white matter structure in diffusion tensor MRI with anisotropy creases. *Medical Image Analysis 11*, 5 (2007), 492–502. 10

[KWTM03] KINDLMANN G., WHITAKER R., TASDIZEN T., MÖLLER T.: Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proc. IEEE Visualization 2003* (2003), pp. 513–520. 8, 9

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics 21*, 4 (1987), 163–169. 2

[Lev88] LEVOY M.: Display of surfaces from volume data. *IEEE Computer Graphics & Applications 8*, 5 (1988), 29–37. 3

[LZP12] LONG F., ZHOU J., PENG H.: Visualization and Analysis of 3D Microscopic Images. *PLoS Computational Biology* (2012). 8

[MB95] MONGA O., BENAYOUN S.: Using partial derivatives of 3D images to extract typical surface features. *Computer Vision and Image Understanding 61*, 2 (March 1995), 171–189. 4

[MGW05] MEYER M. D., GEORGEL P., WHITAKER R. T.: Robust particle systems for curvature dependent sampling of implicit surfaces. In *Proc. Shape Modeling and Applications (SMI)* (June 2005), pp. 124–133. 3, 10

[MLD94] MONGA O., LENGAGNE R., DERICHE R.: Crest lines extraction in volume 3d medical images: a multi-scale approach. In *Proceedings of 12th International Conference on Pattern Recognition* (Oct 1994), vol. 1, pp. 553–555 vol.1. 4

[MWRP10] MAPP O. M., WANNER S. J., ROHRSCHNEIDER M. R., PRINCE V. E.: Prickle1b mediates interpretation of migratory cues during zebrafish facial branchiomotor neuron migration. *Developmental Dynamics 239*, 6 (2010), 1596–1608. 8

[NKH13] NELSON B., KIRBY R. M., HAIMES R.: Gpu-based volume visualization from high-order finite element fields. *IEEE Transactions on Visualization & Computer Graphics 20*, 1 (2013), 70–83. 2

[NLKH12] NELSON B., LIU E., KIRBY R. M., HAIMES R.: ElVis: A system for the accurate and interactive visualization of high-order finite element solutions. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (Dec. 2012), 2325–2334. 2

[OMD*12] OBERMAIER H., MOHRING J., DEINES E., HERING-BERTRAM M., HAGEN H.: On mesh-free valley surface extraction with application to low frequency sound simulation. *Visualization and Computer Graphics, IEEE Transactions on 18*, 2 (2012), 270–282. 4

[PGVCO01] PAULS S., GELDMACHER-VOSS B., CAMPOS-ORTEGA J. A.: A zebrafish histone variant H2A.F/Z and a transgenic H2A.F/Z:GFP fusion protein for in vivo studies of embryonic development. *Development Genes and Evolution 211*, 12 (Dec. 2001), 603–610. 8

[PR99] PEIKERT R., ROTH M.: The "parallel vectors" operator - a vector field visualization primitive. In *Proc. IEEE Visualization* (1999), pp. 263–270. 1, 9, 17

[PS08] PEIKERT R., SADLO F.: Height ridge computation and filtering for visualization. In *Proc. Pacific Vis* (March 2008), Fujishiro I., Li H., Ma K.-L., (Eds.), pp. 119–126. 2

[RP96] ROTH M., PEIKERT R.: Flow visualization for turbomachinery design. In *Proceedings Visualization '96* (Oct. 1996), pp. 381–384. 9

[SH94] SUJUDI D., HAIMES R.: *Identification of Swirling Flow in 3D Vector Fields*. Tech. rep., Dept. of Aeronautics and Astronautics, MIT, Cambridge, MA, 1994. 5, 9

[SP07] SADLO F., PEIKERT R.: Efficient visualization of lagrangian coherent structures by filtered AMR ridge extraction. *IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization) 13*, 6 (Nov 2007), 1456–1463. 2

[SSS06] SCHREINER J., SCHEIDEGGER C., SILVA C.: High-quality extraction of isosurfaces from regular and irregular grids. *IEEE Trans. on Visualization and Computer Graphics 12*, 5 (Sept–Oct 2006), 1205–1212. 10

[Str07] STRAIGHT A. F.: Fluorescent protein applications in microscopy. In *Digital Microscopy, 3rd Edition*, vol. 81 Supplement C of *Methods in Cell Biology*. Academic Press, 2007, pp. 93–113. 8

[STS10] SCHULTZ T., THEISEL H., SEIDEL H.-P.: Crease surfaces: From theory to extraction and application to diffusion tensor MRI. *IEEE Trans. on Visualization and Computer Graphics 16*, 1 (2010), 109–119. 2, 4, 8

[SWTH07] SAHNER J., WEINKAUF T., TEUBER N., HEGE H. C.: Vortex and strain skeletons in Eulerian and Lagrangian frames. *IEEE Transactions on Visualization and Computer Graphics 13*, 5 (Sept. 2007), 980–990. 2

[TKW08] TRICOCHE X., KINDLMANN G., WESTIN C.-F.: Invariant crease lines for topological and structural analysis of tensor fields. *IEEE Trans. on Visualization and Computer Graphics 14*, 6 (2008), 1627–1634. 1, 10

[TM98] TANG C.-K., MEDIONI G.: Extremal feature extraction from 3-D vector and noisy scalar fields. In *Proc. IEEE Visualization '98* (Oct. 1998), pp. 95–102. 2

[WP13] WANNER S. J., PRINCE V. E.: Axon tracts guide zebrafish facial branchiomotor neuron migration through the hindbrain. *Development 140*, 4 (Feb. 2013), 906–915. 8

[ZP04] ZHENG X., PANG A.: Topological lines in 3D tensor fields. In *Proc. IEEE Visualization 2004* (Oct. 2004), pp. 313–320. 8

[ZS17] ZOBEL V., SCHEUERMANN G.: Extremal curves and surfaces in symmetric tensor fields. *The Visual Computer* (Oct. 2017). 2

Appendix A: Direct Volume Rendering in Diderot

The base program for the direct volume rendered figures is below. It was used as-is for Fig. 1. The program comments should support understanding its operation; some additional explanation follows.

```
1   input vec3 camEye ("camera look-from point");
2   input vec3 camAt ("camera look-at point");
3   input vec3 camUp ("camera pseudo-up vector");
4   input real camNear ("at-relative near clip distance");
5   input real camFar ("at-relative far clip distance");
6   input real camFOV ("vertical field-of-view angle");
7   input bool camOrtho ("orthographic (not perspective)") = false;
8   input int iresU ("image # horizontal samples");
9   input int iresV ("image # vertical samples");
10  input real rayStep ("ray inter-sample distance");
11  input real refStep ("opacity reference step length");
12  input real transp0 ("early ray stopping transparency") = 0.005;
13  input real thick ("approximate thickness of feature");
14  input real fStrTh ("feature strength threshold");
15  input real fMaskTh ("feature mask threshold") = 0;
16  input real fBias ("Bias in feature strength computing") = 0.0;
17  input real maxAlpha ("maximum opacity of feature");
18  input vec4 phong ("Phong Ka Kd Ks Sp") = [0.1, 0.7, 0.2, 100];
19  input vec3 litdir ("view-space light direction") = [-1,-2,-1];
20  input vec3 mcNear ("color at near clip plane") = [1,1,1];
21  input vec3 mcFar ("color at far clipping plane") = [1,1,1];
22  input real isoval ("which isosurface to render");
23  input image(3)[] vol ("data to render") = image("vol.nrrd");
24  input image(1)[3] cmap ("scalar colormap") = image("cmap.nrrd");
25  input vec2 cmmm ("min,max colormap range") = [0,0];
26
27  field#2(3)[] F = bspln3 ⊛ clamp(vol);
28  field#0(3)[] Fcm = F;  // colormap scalar field itself
29  field#0(1)[3] CM = tent ⊛ clamp(cmap); // 1-D colormap field
30
31  // Isosurface-specificity limited to these four functions
32  function vec3 fStep(vec3 x) =
33    (isoval - F(x))*∇F(x)/(∇F(x)•∇F(x));
34  function real fStrength(vec3 x) = |∇F(x)|;
35  function real fMask(vec3 x) = F(x);
36  function bool fTest(vec3 x) = true;
37
38  // Computing ray parameters and view-space basis
39  vec3 camN = normalize(camAt - camEye);// N: away from eye
40  vec3 camU = normalize(camN × camUp);  // U: right
41  vec3 camV = camN × camU;              // V: down
42  real camDist = |camAt - camEye|;
43  real camVmax = tan(camFOV*π/360)*camDist;
44  real camUmax = camVmax*iresU/iresV;
45  real camNearVsp = camNear + camDist; // near clip, view space
46  real camFarVsp = camFar + camDist;   // far clip, view space
47
48  // Convert light direction from view-space to world-space
49  vec3 litwsp = transpose([camU,camV,camN])•normalize(litdir);
50
51  // Core opacity function is a capped tent function
52  function real atent(real a0, real d)
53    = a0*clamp(0, 1, 1.5*(1 - |d|/thick));
54
55  function bool posTest(vec3 x)
56    = (inside(x, F)             // in field
57       && fStrength(x) > fStrTh // possibly near feature
58       && fMask(x) >= fMaskTh   // meets feature mask
59       && fTest(x));            // passes addtl feature criterion
60
61  // Each strand renders one ray through (rayU,rayV) on view plane
62  strand raycast(int ui, int vi) {
63    // Compute geometry of ray through pixel [ui,vi]
64    real rayU = lerp(-camUmax, camUmax, -0.5, ui, iresU-0.5);
65    real rayV = lerp(-camVmax, camVmax, -0.5, vi, iresV-0.5);
66    real rayN = camNearVsp;
67    vec3 UV = rayU*camU + rayV*camV;
68    vec3 rayOrig = camEye + (UV if camOrtho else [0,0,0]);
69    vec3 rayVec = camN + ([0,0,0] if camOrtho else UV/camDist);
70
71    // Opacity correction is via alphaFix; distance between
72    // ray samples is |rayVec|*rayStep
73    real alphaFix = |rayVec|*rayStep/refStep;
74    vec3 eyeDir = -normalize(rayVec);
75
76    // Unpack Phong parameters
77    real phKa = phong[0]; real phKd = phong[1];
78    real phKs = phong[2]; real phSp = phong[3];
79
80    output vec4 rgba = [0,0,0,0]; // ray output
81    vec3 rgb = [0,0,0]; // ray state is current color ...
82    real transp = 1;    // ... and current tranparency
83
84    update {
85      rayN += rayStep;           // increment ray position
86      if (rayN > camFarVsp) {    // ray passed far clip plane
87        stabilize;
88      }
89      vec3 pos = rayOrig + rayN*rayVec; // ray sample position
90      if (!posTest(pos)) {
91        continue;
92      }
93
94      vec3 step = fStep(pos);          // step towards feature
95      real aa = atent(maxAlpha, |step|); // opacity
96      if (aa == 0) { continue; }     // skip if no opacity
97      aa = 1 - (1 - aa)^alphaFix;    // opacity correction
98      vec3 snorm = -normalize(step); // "surface normal"
99      real dcomp = (snorm•litwsp)^2; // two-sided lighting
100     real scomp = |snorm•normalize(eyeDir+litwsp)|^phSp
101                  if phKs != 0 else 0.0;
102
103     // simple depth-cueing
104     vec3 dcol = lerp(mcNear, mcFar, camNearVsp, rayN, camFarVsp);
105     vec3 mcol = CM(lerp(0, 1, cmmm[0], Fcm(pos+step), cmmm[1]))
106                 if (cmmm[0] != cmmm[1]) else [1,1,1];
107     // light color is [1,1,1]
108     rgb += transp*aa*((phKa + phKd*dcomp)*modulate(dcol,mcol)
109                       + phKs*scomp*[1,1,1]);
110     transp *= 1 - aa;
111     if (transp < transp0) { // early ray termination
112       transp = 0;
113       stabilize;
114     }
115   }
116   stabilize {
117     if (transp < 1) {  // un-pre-multiply opacities
118       real aa = 1-transp;
119       rgba = [rgb[0]/aa, rgb[1]/aa, rgb[2]/aa, aa];
120     }
121   }
122 }
123 initially [ raycast(ui, vi)
124             | vi in 0..iresV-1, ui in 0..iresU-1 ];
```

The renderer is made specific to isosurface with the feature step `fStep` (line 32) and feature strength `fStrength` (line 34) functions. As described in Sec. 3 and demonstrated in Secs. 4 and 5, different feature step and strength functions will repurpose the renderer for different types of features. Vector and tensor field rendering will involve defining some derived scalar field `F` from the multi-variate data, rather than directly creating `F` from the data as in line 27 above. The feature mask function `fMask`, (line 35) described in Sec. 3.3, offers additional tunable control over what parts of a feature are worth seeing, and the test function `fTest` (line 36) is available as a further criterion for feature membership. These are used in the `posTest` function (line 55) function, which used on line 90 to skip over some ray samples.

Appendix B: Particle-based Feature Sampling in Diderot

The base program for the particle-based feature sampling is below. It was used as-is to generate the isosurface sampling seen in Fig. 1f. The program comments should support understanding its operation; additional explanations follow.

```
input real fStrTh ("Feature strength threshold");
input real fMaskTh ("feature mask threshold") = 0;
input real fBias ("Bias in feature strength computing") = 0.0;
input real tipd ("Target inter-particle distance");
/* tipd is the only length or speed variable with data spatial
   units; everything else measures space in units of tipd */
input real mabd ("Min allowed birth distance (> 0.7351)") = 0.75;
input real travMax ("Max allowed travel to or on feature") = 10;
input int nfsMax ("Max allowed # feature steps ") = 20;
// these next three control the Gradient Descent in Energy
input real gdeTest ("Scaling in sufficient decrease test") = 0.5;
input real gdeBack ("How to scale stepsize for backtrack") = 0.5;
input real gdeOppor ("Opportunistic stepsize increase") = 1.2;
input real fsEps ("Conv. thresh. on feature step size");
input real geoEps ("Conv. thresh. on system geometry") = 0.1;
input real mvmtEps ("Conv. thresh. on point movement") = 0.01;
input real rpcEps ("Conv. thresh. on recent pop. changes") = 0.01;
input real pcmvEps ("Motion limit before PC") = 0.3;
input real isoval ("Which isosurface to sample") = 0;
input int verb ("Verbosity level") = 0;
input real sfs ("Scaling (<=1 for stability) on fStep") = 0.5;
input real hist ("How history matters for convergence") = 0.5;
// higher hist: slower change, more stringent convergence test
input int pcp ("periodicity of population control (PC)") = 5;
input vec3{} ipos ("Initial point positions");
input image(3)[] vol ("data to analyze");

field#2(3)[] F = bspln3 ⊛ clamp(vol);

// Isosurface-specificity limited to fDim and these 5 functions
int fDim = 2;
function vec3 fStep(vec3 x) =
  (isoval - F(x))*∇F(x)/(∇F(x)•∇F(x));
function tensor[3,3] fPerp(vec3 x) {
  vec3 norm = normalize(∇F(x));
  return identity[3] - norm⊗norm;
}
function real fStrength(vec3 x) = |∇F(x)|;
function real fMask(vec3 x) = F(x);
function bool fTest(vec3 x) = true;

function bool posTest(vec3 x) =
   (inside(x, F)              // in field
     && fStrength(x) > fStrTh // possibly near feature
     && fMask(x) >= fMaskTh   // meets feature mask
     && fTest(x));            // passes addtl feature criterion

// Each particle wants between nnmin and nnmax neighbors
int nnmin = 6 if (2==fDim) else 2 if (1==fDim) else 0;
int nnmax = 8 if (2==fDim) else 3 if (1==fDim) else 0;

/* Potential function (found with Mathematica) phi(r):
   phi(0)=1, phi(r)=0 for r >= 1, with minima (potential well)
   phi'(2/3)=0 and phi(2/3)=-0.001.  Phi(r) is C^3
   continuous across the well and with 0 for r >= 1. Potential
   well induces good packing with energy minimization. */
function real phi(real r) {
  real s=r-2.0/3;
  return
    1 + r*(-5.646 + r*(11.9835 + r*(-11.3535 + 4.0550625*r)))
  if r < 2.0/3 else
    -0.001 + ((0.09 + (-0.54 + (1.215 - 0.972*s)*s)*s)*s)*s
  if r < 1 else 0;
}
function real phi'(real r) { // phi'(r) = d phi(r) / dr
  real t=3*r-2;
  return
    -5.646 + r*(23.967 + r*(-34.0605 + 16.22025*r))
  if r < 2.0/3 else
    0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))
  if r < 1 else 0;
}
real phiWellRad = 2/3.0;     // radius of potential well
real rad = tipd/phiWellRad; // actual radius of potential support
function real enr(vec3 x) = phi(|x|/rad);
function vec3 frc(vec3 x) = phi'(|x|/rad) * (1/rad) * x/|x|;

// pchist reflects periodicity of PC: pchist^(2*pcp) = hist
real pchist = hist^(1.0/(2*pcp));

int iter = 0;         // iteration counter
real rpc = 1;         // recent population change
int popLast = -1;    // population at last iteration

/* Finds a number in [0,1) roughly proportional to the low 32
   bits of significand of given real x. NOTE: ONLY useful only
   when compiling with --double */
function real urnd(real x) {
  if (x==0) return 0;
  real l2 = log2(|x|);
  real frxp = 2^(l2-floor(l2)-1); // in [0.5,1.0), like frexp(x)
  // use iter to make different values for same x
  return fmod((2^20 + 2*iter)*frxp, 1);
}

// Given vec3 (and iter), a random-ish value uniformly in [0,1)
function real v3rnd(vec3 v)
  = fmod(urnd(v[0]) + urnd(v[1]) + urnd(v[2]), 1);

// Given vec3 (and iter), a big random-ish integer
function real genID(vec3 v) = floor(1000000*v3rnd(v));

/* Is this an iteration in which to do population control (PC)?
   If not, pcIter() returns 0. Otherwise, returns 1 when should
   birth new particles, and -1 when should kill then off. This
   alternation is not due to any language limitations; it just
   plays well with the PC heuristics used here. */
function int pcIter() = ((iter/pcp)%2)*2 - 1
                        if (pcp>0 && iter>0 && 0 == iter % pcp)
                        else 0;

// Strands first find feature, then interact w/ or make neighbors
strand point (vec3 p0, real hh0) {
  output vec3 pos = p0; // current particle position
  real ID = genID(p0);  // strand identifier
  real hh = hh0;        // energy gradient descent stepsize
  vec3 step = [0,0,0];  // energy+feature steps this iter
  bool found = false;   // whether feature has been found
  int nfs = 0;          // number feature steps taken
  real trav = 0;        // total distance traveled
  real mvmt = 1;        // average of recent movement
  real closest = rad;   // distance to closest neighbor
  int born = 0;         // how many particles I have birthed
  bool first = true;    // first time through update
  update {
    if (!posTest(pos)) {
      die;
    }
    if (travMax > 0 && trav > travMax) {  // too much travel
      die;
    }
    if (!found) { // ------------------------ looking for feature
      if (nfsMax > 0 && nfs > nfsMax) {    // too many steps
        die;
      }
      step = sfs*fStep(pos); // one step towards feature
      pos += step;
      mvmt = lerp(|step|/tipd, mvmt, hist);
      if (mvmt > fsEps) {        // still moving
        trav += |step|/tipd;
        nfs += 1;
      } else {          // found feature, prepare for code below
        found = true;
        mvmt = 1;
        trav = 0;
      }
    } else { // ------------------ feature found; minimize energy
      // if feature is isolated points, we're already done
      if (0 == fDim) { stabilize; }
      step = sfs*fStep(pos); pos += step; trav += |step|/tipd;
      real oldE = 0;              // energy at current location
      vec3 force = [0,0,0];       // force on me from neighbors
      int nn = 0;                 // number of neighbors
      foreach (point P in sphere(rad)) {
        vec3 off = P.pos - pos;
        if (|off|/tipd < fsEps && ID <= P.ID) {
          // with 0-D features or unlucky intialization, points
          // can really overlap; point w/ lower ID dies
          die;
        }
        oldE += enr(off);
        force += frc(off);
        nn += 1;
      }
      if (0 == nn) { // else fDim is 1 or 2
        // No neighbors; create one if possible
        if (!( pcIter() > 0 && born < nnmax )) { continue; }
        // Ensure new pos is near feature, for all
        // feature dimensions and orientations
        vec3 noff0 = fPerp(pos)•[tipd,0,0];
        vec3 noff1 = fPerp(pos)•[0,tipd,0];
        vec3 noff2 = fPerp(pos)•[0,0,tipd];
173         vec3 noff = noff0;
174         noff = noff if |noff| > |noff1| else noff1;
175         noff = noff if |noff| > |noff2| else noff2;
176         // noff is now longest of noff0, noff1, noff2
177         vec3 npos = tipd*normalize(noff) + pos;
178         npos += sfs*fStep(npos);
179         if (posTest(pos)) {
180           new point(npos, hh); born += 1;
181         }
182         continue;
183       }
184       // Else I did have neighbors; interact with them
185       vec3 es = hh*fPerp(pos)●force; // energy step along force
186       if (|es| > tipd) {             // limit motion to tipd
187         hh *= tipd/|es|;             // decrease stepsize, step
188         es *= tipd/|es|;
189       }                              // now |es| <= tipd
190       vec3 fs = sfs*fStep(pos+es);   // step towards feature
191       if (|fs|/(fsEps*tipd + |es|) > 0.5) {
192         hh *= 0.5; // feature step too big, try w/ smaller step
193         continue;
194       }
195       vec3 oldpos = pos;
196       pos += fs + es;            // take steps, find new energy
197       real newE = 0;
198       closest = rad;
199       // find mean neighbor offset (mno) to know (opposite)
200       // direction in which to add new particles with PC
201       vec3 mno = [0,0,0];
202       nn = 0;
203       foreach (point P in sphere(rad)) {
204         vec3 off = P.pos - pos;
205         newE += enr(off);
206         closest = min(closest, |off|);
207         mno += off;
208         nn += 1;
209       }
210       mno /= nn;
211       // test the Armijo sufficient decrease condition
212       if (newE - oldE > gdeTest*(pos - oldpos)●(-force)) {
213         // backtrack because energy didn't go down enough
214         hh *= gdeBack;    // try again next time w/ smaller step
215         if (0 == hh) {
216           die; // backtracked all the way to hh=0!
217         }
218         pos = oldpos;
219         continue;
220       }
221       hh *= gdeOppor; // opportunistically increase stepsize
222       step += fs + es;
223       trav += |step|/tipd;
224       mvmt = lerp(|step|/tipd, mvmt, hist);
225       if (|step|/tipd < pcmvEps && pcIter() != 0) {
226         // can do PC only if haven't moved a lot
227         if (pcIter()>0       // this is an iter to add
228             && newE<0        // already in a potential well
229             && nn<nnmin      // have fewer than expected neighbors
230             && born<nnmax) { // haven't birthed too many times
231           vec3 npos = pos - tipd*normalize(mno);
232           npos += sfs*fStep(npos); npos += sfs*fStep(npos);
233           bool birth = true;
234           if (fDim == 2 && nn >= 4) {
235             foreach (point P in sphere(npos, tipd*mabd)) {
236               birth = false; // too close to existing point
237             }
238             if (birth) {
239               // Have nn neighbors: too few (nnmin > nn).
240               // Try adding a new neighbor with a probability
241               // that scales with nnmin-nn.
242               birth = v3rnd(pos) < (nnmin - nn)/real(nnmin);
243             }
244           }
245           if (birth && posTest(npos)) {
246             new point(npos, hh); born += 1;
247           }
248         } else if (pcIter() < 0 && newE > 0 && nn > nnmax) {
249           // Have too many neighbors, so maybe die. If I have
250           // nn neighbors, they probably also have nn neighbors.
251           // To have fewer, that is, nnmax neighbors, we all
252           // die with chance of nn-nnmax out of nn.
253           if (v3rnd(pos) < (nn - nnmax)/real(nn)) {
254             die;
255           }
256         }
257       }
258     } // else found
259     first  = false;
260   } // update
261 }
262 global {
263   int pop = numActive();
264   int pc = 1 if pop != popLast else 0;
265   rpc = lerp(pc, rpc, pchist);
266   bool allfound = all { P.found | P in point.all};
267   real percfound =
268     100* mean { 1.0 if P.found else 0.0 | P in point.all};
269   real meancl = mean { P.closest | P in point.all };
270   real varicl = mean { (P.closest - meancl)^2 | P in point.all };
271   real covcl = sqrt(varicl) / meancl;
272   real maxmvmt = max { P.mvmt | P in point.all };
273   print("======= finished iter ", iter, " w/ ", pop, ")",
274         "; %found=", percfound,
275         "; mean(hh)=", mean { P.hh | P in point.all},
276         "; mean(cl)=", meancl,
277         "; COV(cl)=", covcl,
278         "; max(mvmt)=", maxmvmt,
279         "; pc=", pc,
280         "; rpc=", rpc,
281         "\n");
282   if (allfound             // all particles have found the feature
283       && covcl < geoEps    // and system is geometrically uniform
284       && maxmvmt < mvmtEps  // and nothing's moving much
285       && rpc < rpcEps) { // and pop. hasn't changed recently
286     print("======= Stabilizing ", numActive(), " (iter ", iter, ")",
287           "; COV(cl)=", covcl, " < ", geoEps,
288           "; max(mvmt)=", maxmvmt, " < ", mvmtEps,
289           "; rpc=", rpc, " < ", rpcEps,
290           "\n");
291     stabilize;
292   }
293   iter += 1;
294   popLast = pop;
295 }
296 initially { point(ipos[ii], 1) | ii in 0 .. length(ipos)-1 };
```

As with the direct volume renderer, the code specific to one feature is isolated to one place: the statement of feature dimension `fDim` (line 31), and the feature functions starting on line 32. Relative to the volume renderer, the new feature function is `fPerp` (line 34), which projects onto the orthogonal complement of the possible local feature steps.

Compared with the basic particle system program (Fig. 4), the program is longer and more complex, but the basic structure is the same. There is still a univariate inter-particle potential energy $\phi(r)$, is implemented as `phi` (line 57), which is a piecewise polynomial with a slight potential well at $r = 2/3$. The function is graphed in

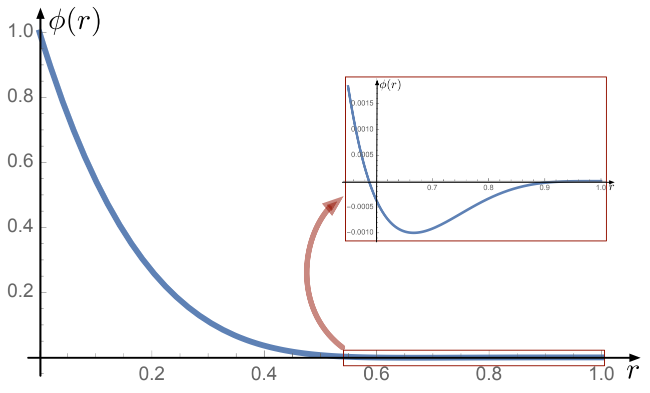


Figure 11: *Graph of inter-particle potential function* $\phi(r)$

Fig. 11, which includes an inset that vertically expands the plot over interval $[0.55, 1]$ to clarify the location and shape of the potential well. The relative shallowness of the potential well compared to height at $r = 0$ ensures that energy minimization separates close particles before it attempts to produce uniform spacing.

The functions over 3D space for energy (`enr` on line 75) and force (`frc` on line 76) are defined as they were in simple Fig. 4

program. The control of the population of the particle system is probabilistic in flavor, using function `v3rnd` (line 97) which generates from a **`vec3`** `v` a value in $[0,1)$ by combining the low-order bits of the X, Y, and Z coordinates of `v` (as exposed by `urnd` on line 88) with the current program iteration count. The current version of Diderot lacks a pseudo-random number generator. The same `v3rnd` is used in the `genID` function (line 101) used to assign to each strand a number (hopefully unique), which proves useful for debugging. A unique per-strand identifier that is thread-safe and stable across iterations is currently not available in Diderot. The periodicity of considering to add or kill particles is controlled by `pcIter` (line 108).

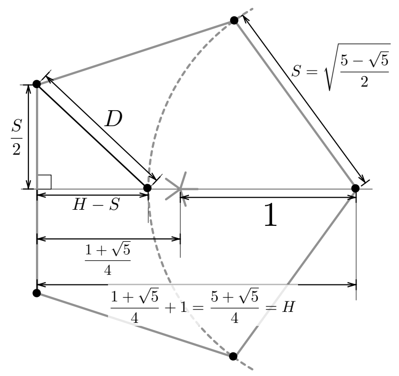
As in the simple particle system (Fig. 4), each program strand computes the position of one particle. Each particle starts (with `found=`**`false`**, line 118) looking for the feature of interest with repeated `fSteps` (lines 132 through 146) while ignoring other particles, after which (lines 147 through 258) particles interact with each other to produce a uniform feature sampling. This second phase includes careful mechanisms for population control. If particles have no neighbors (lines 165 through 183), an effort is made to create a new neighbor close to the feature, using `fPerp`. Computing energy at the updated location (lines 203 through 210) includes computing a mean offset to neighbors `mno`, which is used later (line 231) as part of determining where to try add a new particle in case of under-population. Because the $\phi(r)$ function in the minimal Fig. 4 particle system program was purely repulsive, the last energy gradient descent direction could play that role (Fig. 4 line 73), but here the $\phi(r)$ includes a potential well, so the geometric information in `mno` is useful. If the particle has not predictably moved downhill in energy (line 212), it backtracks and tries again on the next iteration.

Otherwise (line 221), with predictable energy descent, the records of recent motion are updated (line 222), and, if recent motion is small, population control is considered (local estimates of particle density mean less if the system is rapidly moving). Precautions are taken to ensure that the intended location of the a new particle are not too close to an existing one, via the minimum allowed birth distance `mabd` parameter (input line 7, used line 235). This parameter is subtle: if too high, significant holes are never filled in, and if too low, then the pentagonal arrangements of points that may appropriately minimize energy on higher curvature surfaces may trigger the birth of multiple particles, each trying to create a local hexagonal packing (wherein every particle will see $nn_{\text{min}} = 6$ neighbors). Fig. 12 illustrates the geometric reasoning involved in setting `mabd`. If particles, separated by S, have formed a pentagon, then if one adds a new particle at distance S, it will have distance D from another particle on the other side of the pentagon; $D/S \approx 0.735085$. Setting `mabd` higher than this (0.75 works in our experience) prevents pentagonal holes from triggering excessive births. Subsequent meshing can fill the whole by adding two edges and three triangles.

The chances of creating a new particle (if the `mabd` test passes, line 242) or of a particle exiting the computation (line 253) depend on the relationship between the number of neighbors `nn` and the target range of neighbor numbers [`nnmin`,`nnmax`]. The intent is that after one or two periods of population control, the system has roughly the correct number of particles and can proceed to distribute them in a uniform way. While this code with these parameter settings worked adequately to produce our current results, we hope that further computational and geometric analysis can demonstrate the theoretical stability and robustness of the method.

In the final part of the program, the global update (line 262), the particle system state is measured to test for convergence (line 282), which includes tests on the recent stability of particle position and number, as well as their spatial uniformity, as measured by the coefficient-of-variation of distances to interacting neighbors.

$$\frac{D}{S} = \sqrt{\frac{H^2}{S^2} - \frac{2H}{S} + \frac{5}{4}} = \frac{1}{\sqrt{1+\sqrt{\frac{1}{10}\left(5+\sqrt{5}\right)}}} \approx 0.735085$$



$S = \sqrt{\frac{5-\sqrt{5}}{2}}$
D
$\frac{S}{2}$
$H - S$
1
$\frac{1+\sqrt{5}}{4}$
$\frac{1+\sqrt{5}}{4} + 1 = \frac{5+\sqrt{5}}{4} = H$

Figure 12: *Geometric derivation of lower bound on* `mabd` *parameter to avoid filling pentagonal holes in sampling*

Appendix C: Human-readable Diderot intermediate representation

The ability of the Diderot compiler to generate code that computes higher-order derivatives of vector and tensor fields has enabled our work to date. How any compiler converts the surface programming langauge into working code requires multiple stages of internal or intermediate representation. We thought it might be interesting to see what the Diderot compiler is doing with the expressions associated with extremal features, by modifying the (open-source) compiler to print some of its intermediate representations. We show here human-readable expressions for gradient and Hessian of the Parallel Vector operator used for many vector field features [PR99].

If we consider two 3D vector fields $\mathbf{a}(\mathbf{x})$ and $\mathbf{b}(\mathbf{x})$ (these two letters are more easily distinguished than the standard $\mathbf{u}(\mathbf{x})$ and $\mathbf{v}(\mathbf{x})$), the Parallel Vector Operator (PVO) $\mathbf{a}||\mathbf{b}$ is true at points $\mathbf{x}$ where $\mathbf{a}(\mathbf{x})$ is parallel to $\mathbf{b}(\mathbf{x})$, i.e.

$$(\mathbf{a}||\mathbf{b})(\mathbf{x}) \;\Leftrightarrow\; P(\mathbf{x}) = \frac{\mathbf{a}(\mathbf{x})}{|\mathbf{a}(\mathbf{x})|} \cdot \frac{\mathbf{b}(\mathbf{x})}{|\mathbf{b}(\mathbf{x})|} = \pm 1 \tag{24}$$

Our approach to visualizing or extracting $\mathbf{a}||\mathbf{b}$ involves finding the Newton step towards $\mathbf{a}||\mathbf{b}$. Since $\mathbf{a}||\mathbf{b}$ are particular ridge and valley lines of $\mathbf{a}\cdot\mathbf{b}/(|\mathbf{a}||\mathbf{b}|)$ (where the height is $+1$ and -1, respectively), we need the gradient and Hessian of $(\mathbf{a}/|\mathbf{a}|)\cdot(\mathbf{b}/|\mathbf{b}|)$ to compute the Newton step with (12) of Sec. 3.1.

We modified the Diderot compiler to learn expressions for these derivatives, by printing LaTeXor Unicode formattings of the intermediate representation. Starting with a minimal program to evaluate once the gradient of the PVO:

```
1  input image(3)[3] A;
2  input image(3)[3] B;
3  field#2(3)[3] a = bspln3 ⊛ A;
4  field#2(3)[3] b = bspln3 ⊛ B;
5
6  field#2(3)[] P = (a/|a|)•(b/|b|); // the PVO
7
8  strand f(int i) {
9    output tensor[3] r = ∇P([0,0,0]);
10   update {
11     stabilize;
12   }
13 }
14 initially [ f(i) | i in 0..0];
```

Our modified compiler generated:

$$\frac{((A\bullet\nabla\otimes B)+(B\bullet\nabla\otimes A))}{(|A|*|B|)} \tag{25}$$

$$-\Big(\frac{((B\bullet A)*(A\bullet\nabla\otimes A))}{(|A|*|B|*(A\bullet A))}+\frac{((B\bullet A)*(B\bullet\nabla\otimes B))}{(|B|*|A|*(B\bullet B))}\Big), \tag{26}$$

which we manually post-processed to find:

$$\nabla P = \frac{\mathbf{a}\cdot\nabla\otimes\mathbf{b} \;+\; \mathbf{b}\cdot\nabla\otimes\mathbf{a} \;-\; \mathbf{a}\cdot\mathbf{b}\left(\frac{\mathbf{a}\cdot\nabla\otimes\mathbf{a}}{\mathbf{a}\cdot\mathbf{a}}+\frac{\mathbf{b}\cdot\nabla\otimes\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}\right)}{|\mathbf{a}||\mathbf{b}|}. \tag{27}$$

We were not previously familiar with this expression of ∇P, which (to first order) points towards (or away from) where $\mathbf{a}$ and $\mathbf{b}$ are parallel. Terms like $\mathbf{a}\cdot\nabla\otimes\mathbf{b}$ are the Jacobian $\nabla\otimes\mathbf{b}$ of $\mathbf{b}$, contracted on the left by $\mathbf{a}$, which can be thought of as a sum over the rows of $\nabla\otimes\mathbf{b}$, weighted by the components of $\mathbf{a}$. The ∇P expression could also be derived by hand, but it was a nearly automatic side-effect of our modified Diderot compiler. The expression for ∇P is symmetric in switching $\mathbf{a}$ and $\mathbf{b}$, which is reassuring.

For comparison, Van Gelder and Pang, also interested in iterative methods to extract PVO features, derive (with a page of careful explanation) this condition for a step $\boldsymbol{\varepsilon}$ from $\mathbf{x}$ such that $\mathbf{x}+\boldsymbol{\varepsilon}$ satisfies $\mathbf{a}||\mathbf{b}$ (c.f. (29) in [GP09]):

$$\mathbf{q}+\left(\mathbf{I}-\frac{\mathbf{b}\otimes\mathbf{b}}{\mathbf{b}\cdot\mathbf{v}}\right)(\nabla\otimes\mathbf{a}-s\nabla\otimes\mathbf{b})\boldsymbol{\varepsilon}$$
$$-\left(\frac{\mathbf{b}\otimes\mathbf{q}}{\mathbf{b}\cdot\mathbf{b}}\nabla\otimes\mathbf{b}+\frac{\mathbf{q}\otimes\mathbf{a}}{\mathbf{a}\cdot\mathbf{a}}\nabla\otimes\mathbf{a}\right)\boldsymbol{\varepsilon}=\mathbf{0} \tag{28}$$

where

$$\mathbf{q}=\left(\mathbf{I}-\frac{\mathbf{b}\otimes\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}\right)\mathbf{a} \tag{29}$$

is the component of $\mathbf{a}$ orthogonal to $\mathbf{b}$. The authors then describe how $\boldsymbol{\varepsilon}$ may then by computed as the solution to a system of equations as part of an iterative search. They chose a mathematical formulation that is not symmetric in switching $\mathbf{a}$ and $\mathbf{b}$.

We were curious if our modified Diderot compiler could produce a human-readable expression for the Hessian of $P(\mathbf{x}) = \frac{\mathbf{a}(\mathbf{x})}{|\mathbf{a}(\mathbf{x})|}\cdot\frac{\mathbf{b}(\mathbf{x})}{|\mathbf{b}(\mathbf{x})|}$, which is inverted as to compute, via (12), the feature step of our approach. By changing line 9 in the program above to include `r = ∇⊗∇P([0,0,0]);` our modified compiler generated a lengthy expression:

$$\frac{(((\nabla\otimes A)^T\bullet\nabla\otimes B)+(A\bullet\nabla\otimes\nabla\otimes B)+((\nabla\otimes B)^T\bullet\nabla\otimes A)+(B\bullet\nabla\otimes\nabla\otimes A))}{(|B|*|A|)}$$
$$+\frac{(((B\bullet A)*((A\bullet\nabla\otimes A)\otimes(A\bullet\nabla\otimes A)))+(2*(B\bullet A)*((A\bullet\nabla\otimes A)\otimes(A\bullet\nabla\otimes A))))}{(|B|*|A|*((A\bullet A))^2)}$$
$$+\frac{(((B\bullet A)*((B\bullet\nabla\otimes B)\otimes(A\bullet\nabla\otimes A)))+((B\bullet A)*((A\bullet\nabla\otimes A)\otimes(B\bullet\nabla\otimes B))))}{((B\bullet B)*|A|*|B|*(A\bullet A))}$$
$$+\frac{((B\bullet A)*((B\bullet\nabla\otimes B)\otimes(B\bullet\nabla\otimes B)))}{(|B|*|A|*((B\bullet B))^2)}+\frac{(2*|B|*(B\bullet A)*((B\bullet\nabla\otimes B)\otimes(B\bullet\nabla\otimes B)))}{(|A|*(B\bullet B)*((B\bullet B))^2)}$$
$$-\Big(\frac{((((A\bullet\nabla\otimes A)\otimes(B\bullet\nabla\otimes A)))+(((A\bullet\nabla\otimes A)\otimes(A\bullet\nabla\otimes B)))+((B\bullet A)*((\nabla\otimes A)^T\bullet\nabla\otimes A))+((B\bullet A)*(A\bullet\nabla\otimes\nabla\otimes A))+(((A\bullet\nabla\otimes B)\otimes(A\bullet\nabla\otimes A)))+(((B\bullet\nabla\otimes A)\otimes(A\bullet\nabla\otimes A))))}{(|B|*|A|*(A\bullet A))}$$
$$+\frac{((((B\bullet\nabla\otimes B)\otimes(B\bullet\nabla\otimes A)))+(((A\bullet\nabla\otimes B)\otimes(B\bullet\nabla\otimes B)))+(((B\bullet\nabla\otimes B)\otimes(A\bullet\nabla\otimes B)))+(((B\bullet\nabla\otimes A)\otimes(B\bullet\nabla\otimes B))))}{(|A|*|B|*(B\bullet B))}$$
$$+\frac{(((|B|*(B\bullet A)*((\nabla\otimes B)^T\bullet\nabla\otimes B))+(|B|*(B\bullet A)*(B\bullet\nabla\otimes\nabla\otimes B)))}{(|A|*(B\bullet B)*(B\bullet B))}\Big)$$

With some manual post-processing (factoring common terms and regrouping), we develop an expression for the Hessian of P:

$$\nabla\otimes\nabla P = \frac{\begin{array}{c}(\nabla\otimes\mathbf{b})^T\cdot\nabla\otimes\mathbf{a} + (\nabla\otimes\mathbf{a})^T\cdot\nabla\otimes\mathbf{b} + \mathbf{a}\cdot\nabla\otimes\nabla\otimes\mathbf{b} + \mathbf{b}\cdot\nabla\otimes\nabla\otimes\mathbf{a} \\ +\mathbf{a}\cdot\mathbf{b}\left(\frac{3(\mathbf{b}\cdot\nabla\otimes\mathbf{b})\otimes(\mathbf{b}\cdot\nabla\otimes\mathbf{b})}{(\mathbf{b}\cdot\mathbf{b})^2} + \frac{3(\mathbf{a}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{a})}{(\mathbf{a}\cdot\mathbf{a})^2} + \frac{(\mathbf{a}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{b}\cdot\nabla\otimes\mathbf{b})+(\mathbf{b}\cdot\nabla\otimes\mathbf{b})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{a})}{(\mathbf{a}\cdot\mathbf{a})(\mathbf{b}\cdot\mathbf{b})}\right) \\ -\frac{\mathbf{a}\cdot\mathbf{b}\big(\mathbf{b}\cdot\nabla\otimes\nabla\otimes\mathbf{b}+(\nabla\otimes\mathbf{b})^T\cdot\nabla\otimes\mathbf{b}\big) + (\mathbf{a}\cdot\nabla\otimes\mathbf{b}+\mathbf{b}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{b}\cdot\nabla\otimes\mathbf{b}) + (\mathbf{b}\cdot\nabla\otimes\mathbf{b})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{b}+\mathbf{b}\cdot\nabla\otimes\mathbf{a})}{\mathbf{b}\cdot\mathbf{b}} \\ -\frac{\mathbf{a}\cdot\mathbf{b}\big(\mathbf{a}\cdot\nabla\otimes\nabla\otimes\mathbf{a}+(\nabla\otimes\mathbf{a})^T\cdot\nabla\otimes\mathbf{a}\big) + (\mathbf{a}\cdot\nabla\otimes\mathbf{b}+\mathbf{b}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{a}) + (\mathbf{a}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{b}+\mathbf{b}\cdot\nabla\otimes\mathbf{a})}{\mathbf{a}\cdot\mathbf{a}}\end{array}}{|\mathbf{a}||\mathbf{b}|} \tag{30}$$

Review of this expression reveals that it too is symmetric in switching $\mathbf{a}$ and $\mathbf{b}$. $\nabla\otimes\nabla\otimes\mathbf{a}$ is the Hessian of vector field $\mathbf{a}$, a third-order tensor that, when right multiplied by offset $\boldsymbol{\varepsilon}$, gives the local change in the Jacobian. While it would also be possible to derive $\nabla\otimes\nabla P$ by hand, the automated operation of a compiler may be more trustworthy. We show this expression for $\nabla\otimes\nabla P$ to demonstrate functionality that is otherwise hidden inside the Diderot compiler, and to document a complicated formula that others may find useful if implementing Newton steps towards PVO features without Diderot.

Appendix D: Utility programs in Diderot

We we include, for the sake of completeness, other Diderot programs and functions that were used to compute results or generate figures.

D.1. Finding 1D column-space (`col1span`)

```
// finds vector spanning 1D columnspace
function vec3 col1span(tensor[3,3] m) {
  vec3 ret = [0,0,0];
  vec3 c0 = m[:,0]; // extract columns
  vec3 c1 = m[:,1];
  vec3 c2 = m[:,2];
  vec3 c = c0;
  // learn which column is longest
  int which = 0;
  if (|c1| > |c|) { c = c1; which = 1; }
  if (|c2| > |c|) { c = c2; which = 2; }
  // starting with longest column, add in other columns,
  // negating as needed to get longest (most accurate) sum
  if (0 == which) {
    ret = c0;
    ret += c1 if c1•c0 > 0 else -c1;
    ret += c2 if c2•c0 > 0 else -c2;
  } else if (1 == which) {
    ret = c1;
    ret += c0 if c0•c1 > 0 else -c0;
    ret += c2 if c2•c1 > 0 else -c2;
  } else { // 2 == which
    ret = c2;
    ret += c0 if c0•c2 > 0 else -c0;
    ret += c1 if c1•c2 > 0 else -c1;
  }
  // normalize result if possible
  return normalize(ret) if |ret|>0 else [0,0,0];
}
```

The above function is used as part of surface crease line rendering (Sec. 5.3), to find the single eigenvector of a symmetric 3×3 matrix associated with the sole non-zero eigenvalue. This amounts to finding a vector that spans the column space of the matrix, which the above function does by finding the longest possible sum of (possibly negated) columns in the given matrix, and then normalizing.

D.2. Finding edges between particles (`edge.diderot`)

```
 1  input vec3{} ipos ("vertex positions") = load("pos.nrrd");
 2  input real rad ("radius within which verts are edge-connected");
 3
 4  strand point (int ii, vec3 pp) {
 5    // the output of this program is what it print()s,
 6    // rather than this "output" variable foo.
 7    output real foo=0;
 8    int ID = ii;    // record our index in vert list
 9    vec3 pos = pp; // record spatial position
10    update {
11      // the sphere() test implicitly depends on pos
12      foreach (point P in sphere(rad)) {
13        if (ID < P.ID) {
14          // only report each edge once
15          print(ID, "\n", P.ID, "\n");
16        }
17      }
18      stabilize;
19    }
20  }
21
22  initially { point(ii, ipos[ii]) | ii in 0 .. length(ipos)-1 };
```

The above utility program is used for the first stage of meshing feature sampling results systems (Sec. 4.3): connecting neighboring vertices together. Because the particle system tends to produce very uniform samplings at and near convergence, the test for whether two vertices (as represented by two particles) should be considered edge-connected is reduced to knowing if they interacted in the last iteration. Because for this work we have not yet attempted to vary sampling density based on feature characteristics, this is in turn equivalent to asking whether two particles are within the potential function $\phi(r)$ support of each other. Assuming the $\phi(r)$ described in Appendix B, with its potential well at $r = 2/3$, the radius `rad` given to on line 2 should be $3/2$ of the target inter-particle distance (`tipd`, Appendix B line 4) used for particle system computation. A k-d tree created by Diderot run-time based on the special `pos` position variable (line 9) ensures that the **`sphere`** test (line 12) is executed efficiently.

D.3. PostScript mesh drawing (`epsdraw.diderot`)

The program below is included for the sake of completeness since it is used for figure generation (Fig 1 bottom row, Fig. 3(b,c,d), and Fig. 5). It produces a PostScript depiction of small particle systems and their meshes, by computing world-to-view and view-to-screen transformations via homogeneous coordinates. With its ability to label all edges, vertices, and faces in a vector graphics output, it was used for debugging the Appendix B particle system program, and its subsequent meshing. This is not, however, a typical or especially informative Diderot program. Like `edge.diderot` above, the useful output of this program is via its many `print` statements, rather than typical per-strand computed output. Diderot currently has no means of sorting strands based on computed results, so the PostScript commands to draw each element are printed to a single line of text, which starts with "Z `pop`" where Z is screen depth. Sorting these lines as a post-process ensures that PostScript will draw closer elements after (on top of) further elements.

```
input vec3{} ipos ("point positions");
input int{} edg ("edges as pairs of point indices");
input int{} tri ("triangles as triplets of point indices");
int pntNum = length(ipos);
int edgNum = 0 if (edg[0] == 0 && edg[1] == 0)
             else length(edg)/2;
int triNum = 0 if (tri[0] == 0 && tri[1] == 0 && tri[2] == 0)
             else length(tri)/3;
input image(3)[] img ("data to analyze") = image("vol.nrrd");
input vec3 camEye ("camera look-from point") = [6, 9, 2];
input vec3 camAt ("camera look-at point") = [0, 0, 0];
input vec2 clasuv ("Camera Look-at Shift at along U,V") = [0,0];
input vec3 camUp ("camera pseudo-up vector") = [0, 0, 1];
input real camNear ("at-relative near clip distance") = -3;
input real camFar ("at-relative far clip distance") = 3;
input real camFOV ("vertical field-of-view angle") = 15;
input bool camOrtho ("orthographic (not perspective)") = false;
input int iresU ("image # horizontal samples") = 640;
input int iresV ("image # vertical samples") = 480;
input real clwid ("circle line width (in world space!)") = 0.01;
input real elwid ("edge line width (in screen space!)") = 0.1;
input real revth ("draw reversed edges this much thicker") = 6;
input bool cfill ("should fill circle") = true;
input bool bvcull ("back vertex culling") = false;
input real label ("if > 0, font size for labeling things") = 0;
input real crd ("circle radius");
input real drd ("dot radius");
input real frgray ("front-facing gray") = 0.3;
input real egray ("edge gray") = 0;
input real bkgray ("back-facing gray") = 0.8;
input real trigray ("triangle gray") = 0.8;
input real scl ("scaling") = 120;
/* this string identifies what kind of feature should be drawn,
   which matters for choosing how to determine the apparent
   orientation of the disc used to indicate each vertex */
input string feat ("FEAT-ISO, FEAT-RSF, FEAT-VSF, or FEAT-RLN");

// computing ray parameters and view-space basis
vec3 camN_ = normalize(camAt - camEye);// N: away from eye
vec3 camU_ = normalize(camN_ × normalize(camUp)); // U: right
vec3 camV_ = camN_ × camU_;                  // V: down
// now with camAtShift
vec3 camN = normalize(camAt + clasuv[0]*camU_
                      + clasuv[1]*camV_  - camEye);
vec3 camU = normalize(camN × normalize(camUp));
vec3 camV = camN × camU;
real camDist = |camAt + clasuv[0]*camU_
                + clasuv[1]*camV_  - camEye|;
real camVmax = tan(camFOV*π/360)*camDist;
real camUmax = camVmax*iresU/iresV;
real camNearV = camNear + camDist;   // near clip, view space
real camFarV = camFar + camDist;     // far clip, view space

real hght = 2*camVmax;
real width = hght*iresU/iresV;

// determine view transforms
tensor[4,4] WtoV = [
 [camU[0], camU[1], camU[2], -camU●camEye],
 [camV[0], camV[1], camV[2], -camV●camEye],
 [camN[0], camN[1], camN[2], -camN●camEye],
 [0, 0, 0, 1]];
tensor[4,4] perspVtoC = [
 [2*camDist/width, 0, 0, 0],
 [0, 2*camDist/hght, 0, 0],
 [0, 0, (camFarV+camNearV)/(camFarV-camNearV),
  -2*camFarV*camNearV/(camFarV-camNearV)],
 [0, 0, 1, 0]];
tensor[4,4] orthoVtoC = [
 [2/width, 0, 0, 0],
 [0, 2/hght, 0, 0],
 [0, 0, 2/(camFarV-camNearV),
  -(camFarV+camNearV)/(camFarV-camNearV)],
 [0, 0, 0, 1]];
tensor[4,4] VtoC = orthoVtoC if camOrtho else perspVtoC;
tensor[4,4] CtoS = [
 [scl*camUmax, 0, 0, 0],
 [0, scl*camVmax, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1]];

field#2(3)[] F = bspln3 ⊛ clamp(img);

// undo homogeneous coords
function vec3 unh(vec4 ch) =
   [ch[0]/ch[3], ch[1]/ch[3], ch[2]/ch[3]];
// convert to homogeneous coords
function vec4 hom(vec3 c) = [c[0], c[1], c[2], 1];
// how to approximate surface "normal"
function vec3 snorm(vec3 p) {
  vec3 ret=[0,0,0];
  if (feat == "FEAT-ISO") {
      ret = normalize(-∇F(p));
  } else if (feat == "FEAT-RSF") {
      ret = evecs(∇⊗∇F(p)){2};
  } else if (feat == "FEAT-VSF" || feat == "FEAT-RLN") {
      ret = evecs(∇⊗∇F(p)){0};
  } else if (feat == "FEAT-CTP") {
      ret = -camN;
  } else {
      ret = [nan,nan,nan];
  }
  return ret;
}
bool snsgn = true  if (feat == "FEAT-ISO") else
             false if (feat == "FEAT-RSF") else
             false if (feat == "FEAT-VSF") else
             false if (feat == "FEAT-RLN") else
             false;

strand draw (int ii) {
  output real foo=0;
  update {
    // only one strand prints preamble
    if (ii==0) {
      print("%!PS-Adobe-3.0 EPSF-3.0\n");
      print("%%Creator: Diderot\n");
      print("%%Title: awesome figure\n");
      print("%%Pages: 1\n");
      print("%%BoundingBox: ", -scl*camUmax, " ", -scl*camVmax,
            " ", scl*camUmax, " ", scl*camVmax, "\n");
      print("%%EndComments\n");
      print("%%BeginProlog\n");
      print("%%EndProlog\n");
      print("%%Page: 1 1\n");
      print("gsave\n");
      print(-scl*camUmax, " ", -scl*camVmax, " moveto\n");
      print(scl*camUmax, " ", -scl*camVmax, " lineto\n");
      print(scl*camUmax, " ", scl*camVmax, " lineto\n");
      print(-scl*camUmax, " ", scl*camVmax, " lineto\n");
      print("closepath clip\n");
      print("gsave newpath\n");
      print("1 -1 scale\n");
      if (label > 0) {
        print("/Times-Roman findfont\n");
        print(label, " scalefont setfont\n");
      }
    }
    if (ii <= pntNum-1) {
      /*
      p_: position of center of glyph to draw
      q_: from p, in direction towards eye, but tangent
         (normal to normal); should get the most fore-shortening
      r_: from p, in direction perpendicular to q's offset from p
      _w: world-space coords
      _s: screen-space coords
      */
      vec3 pw = ipos[ii];
      vec3 nw = snorm(pw);
      if (|nw| >= 0) {
        // nn == Nothing along Normal
        tensor[3,3] nn =  identity[3] - nw⊗nw;
        vec3 toeye = normalize(camEye - pw);
        vec3 qo = drd*normalize(nn●toeye);
        vec3 qw = pw + qo;
        vec3 ro = drd*normalize(nw×qo);
        vec3 rw = pw + ro;
        vec3 ps = unh(CtoS●VtoC●WtoV●hom(pw));
        vec3 qs = unh(CtoS●VtoC●WtoV●hom(qw));
        vec3 rs = unh(CtoS●VtoC●WtoV●hom(rw));
        if (-1 <= ps[2] && ps[2] <= 1
            && (!snsgn || !bvcull || nw●toeye > 0)) {
          print(ps[2], " pop ");
          print("gsave ");
          print(ps[0], " ", ps[1], " translate ");
          real gray = frgray if (!snsgn) else
                      frgray if (nw●toeye > 0) else bkgray;
          vec3 rso = [[1,0,0],[0,1,0],[0,0,0]]●(rs - ps);
          vec3 qso = [[1,0,0],[0,1,0],[0,0,0]]●(qs - ps);
          print(180*atan2(rso[1],rso[0])/π, " rotate ");
          print(|rso|, " ", |qso|, " scale ");
          print(gray, " setgray ");
          if (clwid > 0) {
            print(clwid/drd, " setlinewidth ");
            print("0 0 ", crd/drd," 0 360 arc closepath ");
            if (cfill) { print("gsave 1 setgray fill grestore ");}
            print("stroke ");
            if (frgray == gray) {
              print("0 0 1 0 360 arc closepath fill ");
            }
          } else {
            print("0 0 1 0 360 arc closepath fill ");
          }
          print("grestore ");
          print("% vi=", ii, "\n");
          if (label > 0) {
            print(ps[2]-0.1, " pop gsave 0.5 setgray newpath ",
                  ps[0], " ", ps[1],
                  " moveto 1 -1 scale (v", ii, ") show grestore\n");
          }
        }
      }
    } else if (ii <= pntNum+edgNum-1) {
      int ei=ii-pntNum; // edge index
      int pi0 = edg[0 + 2*ei];
      int pi1 = edg[1 + 2*ei];
      if (pi0 != pi1) {
        vec3 pw0 = ipos[pi0];
        vec3 pw1 = ipos[pi1];
        vec3 nw0 = snorm(pw0);
        vec3 nw1 = snorm(pw1);
        if (|nw0| >= 0 && |nw1| >= 0) {
          vec3 toeye0 = normalize(camEye - pw0);
          vec3 toeye1 = normalize(camEye - pw1);
          if (!snsgn || (toeye0●nw0 > 0 && toeye1●nw1 > 0)) {
            vec3 ps0 = unh(CtoS●VtoC●WtoV●hom(pw0));
            vec3 ps1 = unh(CtoS●VtoC●WtoV●hom(pw1));
            real ez = min(ps0[2], ps1[2]);
            if (-1 <= ez && ez <= 1) {
              print(ez, " pop ");
              print(egray, " setgray ",
                    elwid*(1 if pi0 < pi1 else revth),
                    " setlinewidth ",
                    ps0[0], " ", ps0[1], " moveto ", ps1[0],
                    " ", ps1[1], " lineto stroke % ei=",
                    ei, "\n");
              if (label > 0) {
                print(ez-0.1, " pop ");
                vec3 ms = lerp(ps0, ps1, 0.5);
                print("gsave 0.5 setgray newpath ", ms[0],
                      " ", ms[1], " moveto (e", ei,
                      ") 1 -1 scale show grestore\n");
              }
            }
          }
        }
      }
    } else {
      int ti = ii-pntNum-edgNum; // tri index
      int pi0 = tri[0 + 3*ti];
      int pi1 = tri[1 + 3*ti];
      int pi2 = tri[2 + 3*ti];
      if (!(pi0 == pi1 && pi1 == pi2)) { // not a fake triangle
```

```
        vec3 pw0 = ipos[pi0];
        vec3 pw1 = ipos[pi1];
        vec3 pw2 = ipos[pi2];
        vec3 pwm = (pw0 + pw1 + pw2)/3;
        vec3 nwm = snorm(pwm);
        vec3 toeye = normalize(camEye - pwm);
        if (!snsgn || toeye●nwm > 0) {
          vec3 ps0 = unh(CtoS●VtoC●WtoV●hom(lerp(pwm,pw0,0.5)));
          vec3 ps1 = unh(CtoS●VtoC●WtoV●hom(lerp(pwm,pw1,0.5)));
          vec3 ps2 = unh(CtoS●VtoC●WtoV●hom(lerp(pwm,pw2,0.5)));
          real ez = min(min(ps0[2], ps1[2]), ps2[2]);
          if (-1 <= ez && ez <= 1 && trigray <= 1) {
            print(ez, " pop ");
            print(trigray, " setgray ",
                  ps0[0], " ", ps0[1], " moveto ",
                  ps1[0], " ", ps1[1], " lineto ",
                  ps2[0], " ", ps2[1],
                  " lineto closepath fill % ti=", ti, "\n");
          }
        }
      }
    }
    if (ii == (pntNum+edgNum+triNum)-1) {
        print("-2 pop ");
        print("grestore grestore\n");
    }
    stabilize;
  }
}

initially { draw(ii) | ii in 0 .. (pntNum+edgNum+triNum)-1 };
```