

Point Movement in a DSL for Higher-Order FEM Visualization

Teodoro Collin¹

Charisee Chiu²

L. Ridgway Scott¹

John Reppy¹

Gordon Kindlmann¹

¹Department of Computer Science, University of Chicago*

²Galois, Inc.†

ABSTRACT

Scientific visualization tools tend to be flexible in some ways (e.g., for exploring isovalues) while restricted in other ways, such as working only on regular grids, or only on unstructured meshes (as used in the finite element method, FEM). Our work seeks to expose the common structure of visualization methods, apart from the specifics of how the fields being visualized are formed. Recognizing that previous approaches to FEM visualization depend on efficiently updating computed positions within a mesh, we took an existing visualization domain-specific language, and added a mesh position type and associated arithmetic operators. These are orthogonal to the visualization method itself, so existing programs for visualizing regular grid data work, with minimal changes, on higher-order FEM data. We reproduce the efficiency gains of an earlier *guided search* method of mesh position update for computing streamlines, and we demonstrate a novel ability to uniformly sample ridge surfaces of higher-order FEM solutions defined on curved meshes.

Index Terms: Software and its engineering—Software notations and tools—Context specific languages—DSLs

1 INTRODUCTION

Novel methods of high-performance and scalable scientific visualization typically support interactively exploring various parameters (e.g., volume rendering transfer functions, or streamline seedpoints), while constraining the *form* of data being visualized. That is, tools for visualizing large biomedical imaging volumes are sensibly specialized for the regular grids that such data is acquired on, just as fluid flow visualization tools are specialized for the finite element method (FEM) meshes on which those phenomena are simulated. Yet from the high-level mathematical standpoint of either characterizing existing visualization methods, or exploring the value of new ones, the specialization of tools to data forms is unfortunate: volume rendering is a kind of integration, and streamlines are solutions to ODEs, regardless of how exactly scalar or vector fields are defined on a grid or mesh. Visualization research may benefit from systems that can take a high-level specifications of a visualization method, and a separate description of how data and fields are formed, and then compile programs that both run efficiently on the given data and support exploration of the relevant parameter spaces.

Domain-specific languages (DSLs) for scientific visualization partially address this need by specializing for a class of algorithms and one form of data: regular grids [8, 10, 17, 22, 23, 35]. In their own ways, these languages work to separate the legible expression of visualization algorithms from technical details of data access or parallel computing. Our current work, however, explores whether a scientific visualization DSL can also be general with respect to data form, so that a program that works efficiently for data on a regular grid can, with minimal changes, also work on an unstructured mesh. Particularly challenging are higher-order finite element simu-

lations, which use higher-order polynomials in both the geometry of mesh elements (so they can be curved) and the form of solutions within elements (beyond affine functions), since this increases the implementation complexity of mere field evaluation.

We present *preliminary* work on extending the open-source compiler for a scientific visualization DSL [8, 19, 20], previously limited to regular grids, to also work on higher-order FEM data. Our long-term goal is to connect previous FEM visualization methods [11, 18, 25, 27–32] by simplifying how they can be expressed and combined in working code. Our current focus is just two kinds of visualizations, both involving computations on a discrete set of points: streamlines [11], and particle systems sampling surface features [25]. The inner loops of both methods share an essential element, *point movement*, i.e., incrementing a position within a higher-order FEM domain by some update vector. We seek a programming language that allows clearly legible implementations of point movement (as a specification of an aspect of the data form), and of the visualization method itself, to be combined in a single program.

Our main contribution approaches this by demonstrating how adding a type for representing FEM mesh positions to a DSL, and overloading operators on that type, simplifies implementing visualization methods that rely on point movement. A supporting contribution, ridge surface extraction in a curved finite element mesh by a particle system, uses the main contribution, and also suggests how other new visualizations could be created by combining general methods with specializations to data form. We hope our work (which itself will be made open-source available) can eventually lower the implementation cost of FEM visualization, as well as help extend standard visualization algorithms to other more general domains (e.g. manifolds) in which points and vectors have distinct significance.

2 FINITE ELEMENT METHOD (FEM) BACKGROUND

A brief explanation of a *simplified* and *typical* use of FEM will support a description of our work. For a solution to a partial differential equation (PDE) $u : \Omega \rightarrow \mathbb{R}^n$, FEM uses a finite dimensional vector space of functions (function space) V to find an approximate solution, $u_V \in V$, to u . The space V is created by discretizing the world-space domain Ω into a collection of disjoint cells $\{K_i\}$, defining a function space P_i on each K_i , and then combining all the P_i , e.g., $v \in V$ if and only if $v|_{K_i} \in P_i$ for all element indices i [6]. The per-element function space is typically $P_i = \{p \circ T_i^{-1} \mid p \in P\}$, where P is a finite dimensional function space on a convex polytope K (the reference cell), each T_i is an injective C^∞ mapping from K to world-space, and both P and T_i are polynomial. Consequently, finite element solvers do not need to explicitly represent the P_i (or compute T_i^{-1}), and can compute all quantities on the reference cell K [1, 4, 6, 7, 34].

Unfortunately, visualization naturally works in world-space Ω . Within some $K_i \subset \Omega$, the PDE solution being visualized u_V will be represented in a chosen basis $\{p_j\}$ for P as

$$u_V(x)|_{K_i} = \sum_j c_j p_j(T_i^{-1}(x)). \quad (1)$$

Using higher-order FEM, with non-linear T_i , increases the computational cost for a *naive* visualization algorithm to traverse just a single cell i , since each of the many evaluations of T_i^{-1} in (1) require

*e-mails: {teocollin,ridg,jhr,glk}@cs.uchicago.edu

†e-mail: chiu@galois.com

multiple Newton iterations. Moreover, as the visualization traverses world-space, for each point $x \in \Omega$ it needs to compute T_i^{-1} for many different cells i in order to find the i for which $x \in T_i(K)$ [30]. Visualization algorithms specialized to finite elements avoid the cost of T_i^{-1} by replacing, when possible, world-space evaluation of the approximate solution u_V via (1) with

$$f_i(x) := \sum_j c_j p_j(x), \quad (2)$$

where $f_i: K \rightarrow \mathbb{R}^n$ is the evaluation of u_V on the *reference cell* with respect to element i .

3 RELATED WORK

Many tools for visualizing FEM solutions (including ParaView [3], Gmsh [15], and GLVis [16]) use *tessellation*, i.e., approximating one higher-order element with multiple smaller affine elements [36]. This is good for simple visualizations (e.g. colormapping u_V), but more problematic for more complicated ones, such as volume rendering or those that require higher order derivatives [30]. The tessellation framework of Schroeder et al. is in principle general and accurate with respect to visualization method [38], but we are unaware of its application beyond isosurfaces and streamlines.

Our work follows a different strategy, advanced by Nelson et al., which directly visualizes elements, without tessellation. Via algorithms that directly manipulate u_V , f_i , K and T_i , these authors create fast and accurate methods for ray-tracing isosurfaces, cut surfaces, and volume rendering [27–29]. They also combine methods into ElVis, a GPU-based interactive GUI application, which offers some generality over data forms via a plugin architecture that supports a small set of visualization algorithms [30]. Our DSL, however, allows more room to explore implementation variation within or between data forms, at the expense of lower computational performance relative to hand-written low-level code.

A variety of other previous work investigates FEM visualization under various accuracy, expression, and performance constraints, as surveyed by Nelson et al. [30]. Some work focused on fast and accurate visualizations of curved quadratic and cubic elements [41] while other work achieved interactive volume rendering of higher order elements [40]. Later sections will describe in more detail the work of Coppola et al. [11] and Meyer et al. [25], which is most central for our current work. There is no prior work extracting ridge surfaces from FEM data, but Pagot et al. find ridge lines on affine meshes via PVO and new seed finding and streamline routines [31]. Jallepalli et al.’s smoothing of finite element data could usefully complement the visualization methods we target [18].

We also build on related work with DSLs. Being specific to some domain of algorithms, DSLs trade reduced flexibility of the language for (in principle) higher human productivity of writing new programs within that domain [24]. For our purposes we merely note FEM-related DSLs for formulating and solving PDEs [1, 2, 12, 34] as well as DSLs for processing and visualizing image and volume data [8, 10, 17, 22, 23, 26, 33, 35]. This list does not fairly describe the sophisticated approaches to high-performance computing [23] and computational scheduling [26, 33]. We build on Diderot, a visualization DSL limited to regular grids [8, 19, 20], but distinguished by offering the mathematical abstraction of a C^k tensor field. Our current work extends how Diderot fields are defined to include FEM, so that existing Diderot programs can be used with minimal changes, while introducing a new abstraction, a *mesh position*, which supports the convenient expression of previous methods of moving through the geometry of a curved FEM mesh [11, 25].

4 METHODS

4.1 Point Movement via Guided Search

Many scientific visualization algorithms enjoy *spatial coherence*: field evaluation at (world-space) position x will likely be followed

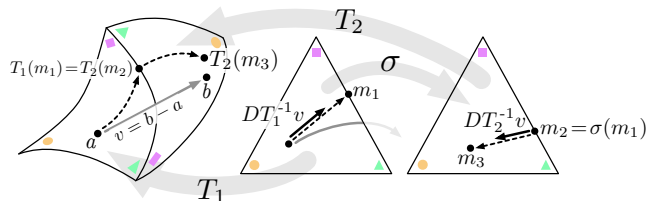


Figure 1: Illustration of guided search to move a towards b , with two world-space cells (left) and two copies of reference space (center and right), each with transforms T_1 and T_2 . Colored shapes at cell vertices clarify how reference spaces connect. At a , the velocity $v = b - a$ is transformed by DT_1^{-1} to give a reference space velocity $DT_1^{-1}v$ along which traversal begins. At cell boundary m_1 , the permutation σ between reference vertices determines the start $m_2 = \sigma(m_1)$ of the next cell traversal, now along $DT_2^{-1}v$ at m_2 . More cells may be encountered, until the computed path (shown as a dashed line) has run for time = 1, ending at m_3 . The point $T_2(m_3)$ approximates b .

by evaluation at a nearby $x + v$. As noted in §2, for simple methods, the computational expense (from finding the cell containing x , and finding $T_i^{-1}(x)$) of naively evaluating $u_V(x)$ via (1) might be avoided by evaluations in *reference space* via (2), and then forward mapping by T_i . Previous work with more complex visualization methods, however, demonstrates the value of rapidly approximating $x + v$ in a sequence of reference spaces, a technique we term *point movement*, so that $u_V(x + v)$ can be found faster than via naive re-evaluation of (1). For streamlines, Coppola et al. name their method of point movement *guided search* [11], while a similar method underlies the isosurfacing particles of Meyer et al. [25].

Guided search builds on a few technical observations. First, for world-space position $x \in T_i(K)$ and update v , a first-order approximation of the updated position is $T_i^{-1}(x + v) \approx T_i^{-1}(x) + (DT_i^{-1}(x))v$. Second, the location where $T_i^{-1}(x) + (DT_i^{-1}(x))v$ exits K can be found via geometric computations on the shape of K , common in computer graphics [14]. Third, in most meshes, the mapping between reference positions in two adjacent cells is entirely determined by a simple permutation σ on the vertices of K . Thus for $x \in \partial K$ on the boundary of the reference cell for cells i and j , the same world space position is both $T_j(\sigma(x))$ and $T_i(x)$. Combining these ideas together yields the guided search algorithm illustrated in Fig. 1.

We make two observations about the context and implementation of guided search. Guided search has only been described as a part of a specific visualization ingredient: Coppola et al. presents guided search as a sub-step of RK4 integration [11]. In fact, it can be separated from any particular numerical or visualization method and framed as a method to update positions by a vector, a definition in affine geometry [39]. Second, guided search is complicated enough to warrant exploring the speed and accuracy of possible variants. For example, Coppola et al. also describe *error-checked guided search*, wherein the search defaults to the naive scheme to locate $x + v$ if $\|T_i(T_i^{-1}(x) + t(DT_i^{-1}(x))v) - (x + tv)\|$ exceeds some threshold. The same considerations of orthogonality and legibility that motivate creating DSLs also suggest clearly expressing the point movement method within the language.

4.2 FEM Data, Position types, and Overloading

To demonstrate point movement within a mesh as a programmable and orthogonal aspect of a FEM visualization algorithm, we augment an existing scientific visualization DSL with a new position type, overloaded operators on positions, and the ability to input FEM data. We chose the Diderot language because it already simplifies implementing streamlines and particle systems on regular grids [19, 20], and because its consistent use of a *field* abstraction facilitates introducing FEM solutions as a new underlying data form.

Space here permits a high level summary of the language changes

```

1  input int timeSteps=32;
2  const int dim = mesh.t.dim; // (this works in 2 or 3 dimensions)
3  input real timeEps = 0.0000001;
4  input mesh_t mesh;
5  refCell{mesh_t} K = mesh.refcell;
6  overload position{mesh_t} +(position{mesh_t} x, tensor[dim] delta){
7    if (!x.isValid){ return(x);}
8    real time = 1;
9    position{mesh_t} cmp = x; // current mesh pos
10   foreach (int i in 0..timeSteps ){ // ("mc"=mesh cell)
11     tensor[dim, dim] iJac = inv(V⊗(cmp.mc.transform)(cmp.refPos));
12     tensor[dim] refDelta = iJac • delta; // reference velocity
13     tensor[dim] nPos = cmp.refPos + time*refDelta;
14     if (K.isInside(nPos)) { // nPos is inside K
15       return(cmp.mc.meshPos(nPos)); // x+v=nPos in cmp.mc
16     } else { // we left the reference cell; compute when we left
17       real eTime = K.exit(cmp, normalize(refDelta));
18       time -= time/|refDelta|; // decrement time remaining
19       if (eTime == -1){ // invalid direction, use naive scheme
20         return(mesh.findPos(x.worldPos() + delta));
21       } // else find the exit location in the next cell
22       position{mesh_t} nmp = K.exitPos(cmp, normalize(refDelta));
23       if ( !nmp.isValid || time < timeEps){ // left mesh or
24         return(nmp); // ran out of time
25       }
26       cmp = nmp;
27     }
28   } // spent too much time; use naive scheme
29   return(mesh.findPos(x.worldPos() + delta));
30 }

```

Figure 2: An overloaded “+” operator implements a minimal version of guided search along with minimal context. Lines 1-5 declare the necessary inputs. Lines 7-9 set up the search. Lines 10-29 are the main body. At each iteration, lines 11 and 12 transform the change in position via the position’s cell’s transform. Lines 14 and 15 return a new position if the transformed velocity does not take the position outside the cell. Otherwise, lines 17-28 find the time of intersection (line 17), check that the intersection makes sense (line 19), find the position of this intersection on the next cell (line 22), check if this is the last step (line 23), and continue on (line 26). If the loop terminates, line 29 defaults to the naive scheme.

required to create a path from FEM data to existing language objects: domains, fields, and tensors. The domain of an FEM solution involves a mesh, reference cell domain K , and the $T_i(K)$; each requires a new language type and constructors via inputs or methods. Meshes are global (immutable) inputs to the program, supplying a sequence of cells on the mesh. The global solution u_V can be accessed as a field after providing a space type, a solution type, and an input.

For fields attached to cells, such as T_i , T_i^{-1} , and f_i , the fields are cell methods. To enjoy the benefits of sampling within a reference cell, cells provide a *transformed reference field* which supports evaluating values $f_i(x)$ and derivatives $D^n(f_i \circ T_i^{-1})$ at $T_i(x)$, so that tensor-valued (world-space) derivatives of u_V can be efficiently sampled from reference space. Meyer et al. also sample gradients and Hessians from the reference cell, and use a lengthy Einstein notation derivation to find the world-space derivatives [25]. All these mechanics are thankfully handled automatically by the Diderot compiler’s internal representation, itself based on Einstein notation [9].

To support the notion of a position on a mesh, we added a new position type that depends on a mesh type; other domain types could be supported later. Position values are constructed either with a point in reference space K and a mesh cell, or via a point in world space; the latter option corresponds to the naive scheme. Strands of Diderot computation can be associated with positions, so that strands (e.g., for particle systems) can query the state of their neighbors via k-d trees [37]. We also added queries on the reference cell geometry, to determine when a point leaves its cell by traveling in a direction, and to learn the corresponding position in the neighboring cell (if it exists). With all this in place, positions can become arguments to an overloaded “+” operator, the concise guided search implementation of Figure 2. Adding these capabilities to the compiler required adding or changing around 5000 lines of Standard ML, but this cost is once per data form as we can now use the compiler to explore the adoption of many previous Diderot programs to a FEM context that is consistent with §2. Below, we focus on just two Diderot programs.

The language elements described above allow separating the expression of visualization algorithms from both the details of field evaluation and the details of point movement. In particular, we were able to modify existing Diderot programs for streamlines [19] and particles [20] in regular grids to work with FEM data via straight-forward transformations. We declared FEM types and inputs, added point movement code (Fig. 2), changed field sampling to a function that samples the reference field using a position, and changed several types from vectors to positions. While this seems extensive, besides the copy-pasted parts, disruption to existing code was minimal: 15 lines changed in a 30-line streamline program, 30 lines changed in an 80-line isosurfacing particle program, and 40 lines changed in a 300-line program for general feature sampling with particle systems. The full analysis is in the supplementary materials.

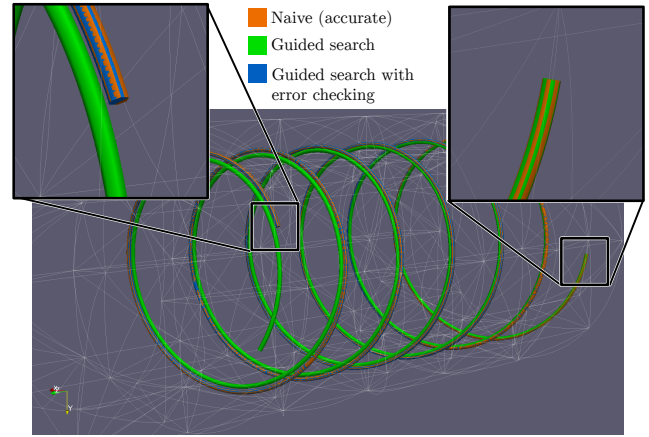


Figure 3: Streamlines in this synthetic vector field in a curved mesh should be helices of constant radius. Three schemes for updating position during integration are seeded at the same location (right inset), but guided search (green) diverges by the end (left inset). Guided search with error checking (blue) very closely follows the accurate and more expensive naive method (orange).

5 RESULTS

Our results all use curved meshes with cubic T_i transforms. Additional software was used to create meshes (gmsh [15]), finite element data (Firedrake [34]), and renderings (ParaView [3]). The Diderot code can be found in the supplementary materials.

To test streamlines, we created a curved mesh between two concentric cylinders. We then interpolated $f(x, y, z) = (y, -x, 0.1)$ onto a function space specified by the mesh and quadratic P . An RK2 streamline program with guided search produces the green path in Fig. 3. Also shown is an orange path produced via the more expensive naive scheme, which shows the accuracy of the blue path computed with the error-checked guided search noted in §4.1. This result is consistent with Coppola et al.’s accuracy analysis of these schemes [11]: standard guided search errs in ways that error-checked guided search avoids.

Coppola et al. also analyze the *performance* of guided search. Our preliminary results in Table 1 reproduce their finding that error-checked guided search runs 2 – 10 times faster than the naive approach; this is notable considering that our code is in a new high-level DSL. We are also encouraged by this speed-up because it justifies the compiler and language effort of §4.2 and facilitates future work on exploring new point movement techniques independently of visualization methods.

Table 1: For computing streamlines with error-checked guided search (as in Fig. 3), over various step sizes (rows) and error parameters (columns), the table gives run times in seconds and speed-ups (in parentheses) of guided search relative to the naive scheme. Timing comparisons use equal numbers of steps within the mesh. We note that the speedup results exhibit large variations between step sizes; we hypothesize that the variations occur because decreasing step size can unpredictably both increase speedup via reducing error checking and decrease speedup via potentially increasing the number of points on a path that are close to the initial guess of Newton’s method, the elemental center, potentially improving the naive scheme’s time [5].

Step Size	Error Parameter		
	10^{-4}	10^{-5}	10^{-6}
0.2	0.036s (2.524)	0.049s (2.220)	0.052s (2.056)
0.02	0.097s (9.183)	0.149s (6.735)	0.120s (7.846)
0.002	5.353s (2.600)	6.530s (2.369)	5.749s (2.423)

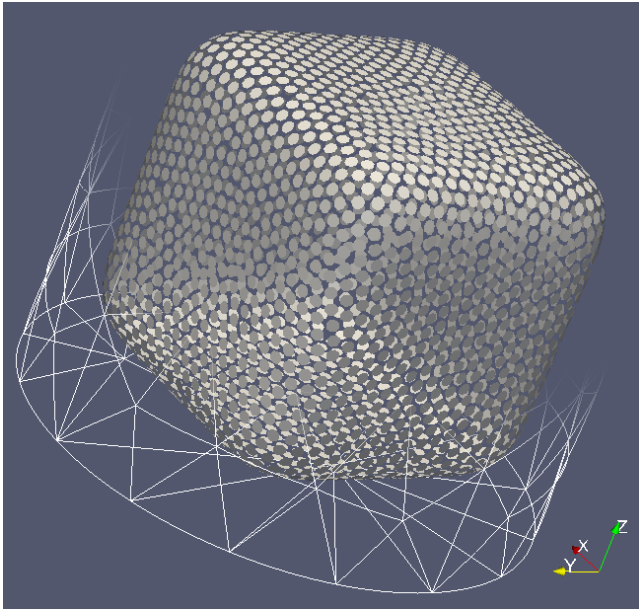


Figure 4: Particle based sampling of an isosurface in the form of a rounded cube, contained within a cylindrical mesh, the curved boundary of which is visible in the lower part.

Meyer et al. pioneered isosurfacing via particle systems on curved geometries [25]. It took us a few hours to adapt an 80-line minimalist isosurface sampling Diderot program [20] to produce Fig. 9, showing a sampling of the isocontour $x^6 + y^6 + z^6 = 1$ in a cylindrical mesh with hexic P . Our result lacks curvature-adaptive sampling [25], but it shows the viability of our approach.

We also sample a *ridge surface* of the function $f(x,y,z) = z^2 \sin(x^2 + y^2 + z^2)$, inspired by Eberly’s consideration of ridges in fluid flow (c.f. Fig. 6.49 in [13]). We created a curved mesh between two concentric spheres, and approximated f in a function space given by the mesh and hexic P . Since f is non-polynomial, the approximation is at most C^0 continuous across cell boundaries, which could create discontinuities in the ridge surface itself. A 300-line Diderot program for sampling general features with particle systems was adapted as described in §4.2, and the results in Fig. 5 were found after experimenting with parameters (feature strength threshold of 24, feature bias of 0.1). We spent more time creating the example function and mesh than we did writing and using the program. Therefore, we feel that the most interesting aspect of this

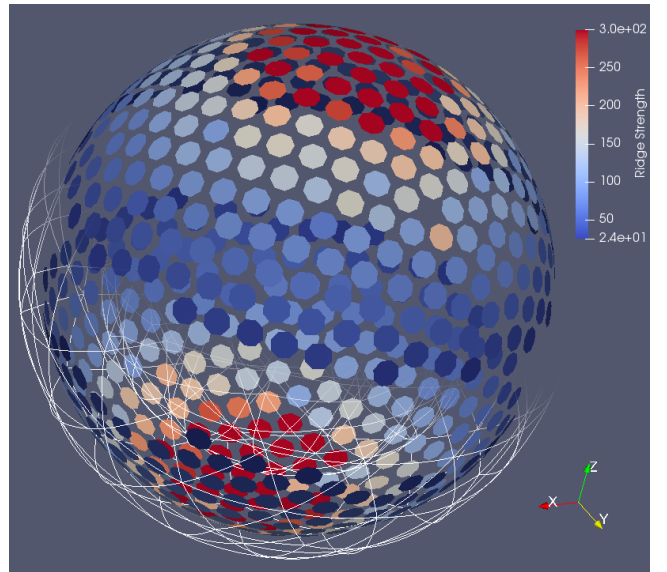


Figure 5: Particle based sampling of a ridge surface on a curved geometry, colormapped by feature strength. The curved edges of boundary elements of the mesh are shown in lower part.

result is in the DSL design. The conceptual orthogonality of guided search and particle system evolution was manifested as a clean separation in the code between the implementation of those two methods, such that the guided search code remained unchanged as the particle system code was changed from isocontours to ridge surfaces.

6 DISCUSSION, CONCLUSIONS, AND FUTURE WORK

Our methods and results demonstrate that point movement and visualization algorithms can be orthogonal and composable. We have shown how conceptually orthogonal algorithmic components can be cleanly expressed as separate pieces of code in a DSL. The ridge surface example additionally shows that this separation extends in concept and in code to particle systems for a novel visualization target for FEM data. We hope these results convince readers of the potential of high-level DSLs to create new visualization programs by combining two separate specifications: one of the data form (regular grid or FEM data), and one of the core visualization algorithm.

Directions of ongoing and future work are organized around Diderot, guided search, and the expression of visualization algorithms in general. With respect to Diderot, we hope to add other data forms beyond regular grids and finite element data such as Riemannian manifolds, where position movement is given by solving a variational problem to find a geodesic. With respect to guided search, we wish to augment it with information from run-time or compile-time, and we wish to explore its application to visualization methods that are less sensitive to errors in position location (i.e., volume rendering or predictor-corrector schemes used in PVO line tracing [21, 31]). Finally, we suspect that many existing implementations of visualization algorithms currently specialized to a particular data form may contain ideas that are as orthogonal and composable as guided search is for FEM data. We hope to foster further research by uncovering those ideas and exploring how they can be combined into new visualization algorithms, implemented in idiomatic and re-usable code.

ACKNOWLEDGMENTS

This work was supported by NSF grant CCF-1564298.

REFERENCES

- [1] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3, 2015.
- [2] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9:1–9:37, Mar. 2014.
- [3] U. Ayachit. *The ParaView Guide (Full Color Version): A Parallel Visualization Application*. Kitware, Incorporated, 2015.
- [4] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.
- [5] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer New York, 1998.
- [6] S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer New York, 2008.
- [7] C. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. D. Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. Kirby, and S. Sherwin. Nektar++: An open-source spectral/hp element framework. *Comp. Phys Comm.*, 192:205–219, 2015.
- [8] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *Proc. PLDI*, pages 111–120, June 2012.
- [9] C. Chiw, G. L. Kindlmann, and J. H. Reppy. Compiling Diderot: From tensor calculus to C. *CoRR*, abs/1802.06504, 2018.
- [10] H. Choi, W. Choi, T. M. Quan, D. G. C. Hildebrand, H. Pfister, and W.-K. Jeong. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE TVCG (Proc. SciVis)*, 20(12):2407–2416, Dec. 2014.
- [11] G. Coppola, S. Sherwin, and J. Peiró. Nonlinear particle tracking for high-order elements. *Comp. Phys. Comm.*, 172(1):356–386, 2001.
- [12] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proc. SC11*, pages 1–12, Nov 2011.
- [13] D. Eberly. *Ridges in Image and Data Analysis*. Springer Netherlands, 1996.
- [14] C. Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. CRC Press, 2004.
- [15] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Numer. Methods Eng.*, 79(11):1309–1331, 2009.
- [16] GLVis: OpenGL finite element visualization tool. glvis.org.
- [17] M. Hašan, J. Wolfgang, G. Chen, and H. Pfister. Shadie: A domain-specific language for volume visualization. Draft paper; available at <http://miloshasan.net/Shadie/shadie.pdf>, 2010.
- [18] A. Jallepalli, J. Docampo-Sanchez, J. K. Ryan, R. Haimes, and R. M. Kirby. On the treatment of field quantities and elemental continuity in FEM solutions. *IEEE TVCG*, 24(1):903–912, 2018.
- [19] G. Kindlmann, C. Chiw, N. Seltzer, L. Samuels, and J. Reppy. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE TVCG*, 22(1):867–876, 2016.
- [20] G. L. Kindlmann, C. Chiw, T. Huynh, A. Gyulassy, J. Reppy, and P.-T. Bremer. Rendering and extracting extremal features in 3D fields. *Comp. Graph. Forum*, 37(3):525–536, June 2018.
- [21] G. D. Kontopidis and D. E. Limbert. A predictor-corrector contouring algorithm for isoparametric 3D elements. *International Journal for Numerical Methods in Engineering*, 19(7):995–1004, jul 1983.
- [22] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: A data-parallel programming language for graphics processors. *J. Par. Comp.*, 33:648–662, Nov. 2007.
- [23] P. McCormick, C. Sweeney, N. Moss, D. Prichard, S. K. Gutierrez, K. Davis, and J. Mohd-Yusof. Exploring the construction of a domain-aware toolchain for high-performance computing. In *Proc. WOLFHPC*, pages 1–10, 2014.
- [24] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comp. Surv.*, 37(4):316–344, 2005.
- [25] M. Meyer, B. Nelson, R. Kirby, and R. Whitaker. Particle systems for efficient and accurate high-order finite element visualization. *IEEE TVCG*, 13(5):1015–1026, 2007.
- [26] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling Halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.
- [27] B. Nelson, R. Haimes, and R. M. Kirby. GPU-based interactive cut-surface extraction from high-order finite element fields. *IEEE TVCG*, 17(12):1803–1811, 2011.
- [28] B. Nelson and R. M. Kirby. Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering. *IEEE TVCG*, 12(1):114–125, 2006.
- [29] B. Nelson, R. M. Kirby, and R. Haimes. GPU-based volume visualization from high-order finite element fields. *IEEE TVCG*, 20(1):70–83, 2014.
- [30] B. Nelson, E. Liu, R. M. Kirby, and R. Haimes. Elvis: A system for the accurate and interactive visualization of high-order finite element solutions. *IEEE TVCG*, 18(12):2325–2334, 2012.
- [31] C. Pagot, D. Osmari, F. Sadlo, D. Weiskopf, T. Ertl, and J. Comba. Efficient parallel vectors feature extraction from higher-order data. *Comp. Graph. Forum*, 30(3):751–760, 2011.
- [32] C. A. Pagot, J. E. Vollrath, F. Sadlo, D. Weiskopf, T. Ertl, and J. L. D. Comba. Interactive isocontouring of high-order surfaces. In *Scientific Visualization: Interactions, Features, Metaphors*, pages 276–291. Schloss Dagstuhl, 2011.
- [33] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. PLDI*, pages 519–530. ACM, 2013.
- [34] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, 2016.
- [35] P. Rautek, S. Bruckner, M. E. Gröller, and M. Hadwiger. ViSlang: A system for interpreted domain-specific languages for scientific visualization. *IEEE TVCG*, 20(12):2388–2396, 2014.
- [36] J.-F. Remacle, N. Chevaugneon, É. Marchandise, and C. Geuzaine. Efficient visualization of high-order finite elements. *Int. J. Numer. Methods Eng.*, 69(4):750–771, 2006.
- [37] J. Reppy and L. Samuels. Bulk-synchronous communication mechanisms in diderot. Presented at the *Compilers for Parallel Computing Workshop (CPC '15)*, Jan. 2015.
- [38] W. Schroeder, F. Bertel, M. Malaterre, D. Thompson, P. Pebay, R. O'Bara, and S. Tendulkar. Methods and framework for visualizing higher-order finite elements. *IEEE TVCG*, 12(4):446–460, 2006.
- [39] I. R. Shafarevich and A. O. Remizov. *Linear Algebra and Geometry*. Springer, 2012.
- [40] M. Üffinger, S. Frey, and T. Ertl. Interactive high-quality visualization of higher-order finite elements. *Computer Graphics Forum*, 29(2):337–346, may 2010.
- [41] D. F. Wiley. *Approximation and Visualization of Scientific Data Using Higher-order Elements*. PhD thesis, University of California, Davis, 2003.

7 SUPPLEMENTARY MATERIALS FOR POINT MOVEMENT IN A DSL FOR HIGHER-ORDER FEM VISUALIZATION

8 SUMMARY OF SUPPLEMENTAL MATERIALS

In this document, we will include the commented code that is used by “Point Movement in a DSL for Higher-Order FEM Visualization.” Each section will provide Diderot snippets that could be used to produce the figures in the paper. These snippets are complete in the sense that they are either full programs in their own right or could easily be combined with another snippet featured here to make a full program. In addition, each section will provide some brief commentary on the code, highlighting both the new ideas in the code not evident before and the work required to create these snippets from older Diderot programs. The snippets included below are:

1. A streamline program that uses guided search.
2. An overloaded function for the error checked guided search scheme.
3. An overloaded function for the naive scheme.
4. A small particle system used to create isosurfaces.
5. A larger particle system used to create ridge surfaces.

Finally, we note that the figure captions provide information about parameters that we used to create our figures if they are unstated in the paper.

9 STREAMLINES

We provide line by line commentary on Figure 6, which features a complete guided search streamline program. Line 1 declares the version of the compiler that we use, which is 3.0. Lines 2 through 7 declare the mesh, function space (V), and function types (u_V) that the program uses. The JSON files used in these declarations will be documented further in future work, but we note that we automatically generate them for Firedrake’s meshes, spaces, and functions.

Lines 10 through 12 take inputs for the mesh, space, and function. Line 13 provides the reference cell from the mesh. Lines 18 through 44 are featured in the paper as the guided search algorithm. Lines 46 through 54 supply an auxiliary function, nV , to unpack positions and evaluate the field f_i on the reference cell; this process consists in taking out a mesh cell, reference position, and then using the information about u_V , the data, to get a field, which is sampled at the reference position. The auxiliary function checks the validity of the position and allows for some sort of border control. Lines 55 through 57 simply control the streamline algorithm, RK2. Lines 58 through 73 are the streamline algorithm with some modifications to use positions: First, line 64 takes the input vector to the strand, line, and converts it to a position. Second, lines 66 and 69 provide border control by checking for the validity of the positions (in older Diderot versions, this was done via an inside function.) Finally, line 67 records the world position of a point on the streamline. The core RK2 algorithm is legible in lines 68 and 70, which use the function nV to sample a field at a position as in the older Diderot streamlines program.

We now roughly measure the changes required to turn a vanilla Diderot streamline program to this program. A standard Diderot streamline program would live in lines 46 through 73 after the addition of image inputs and field declarations. The changes in the function, nV , defined from 46 to 53, are changes in the field evaluation and represent an additional 5 to 10 lines from the original program. The strand definition is changed at lines 64 (type change and conversion), line 66 for checking the validity, line 67 for recording the world position, line 68 for a type change, and line 69 for another validity check. Thus, the total changes to the streamline

program amount to about 15 lines of code besides the addition of the FEM inputs and guided search (line 1 through line 44)

In Figure 7, we show the overloaded position operator with an error checking functionality. An error max parameter is required as a new input on line 1 and we now provide commentary on its usage i.e the additions present in the error checked guided search. The error parameter is used on lines 14 and 33 to check if the computed reference position corresponds to a world space position that is close to the correct world space position i.e the world space position of a naive position update. If on either line 14 or 33, the condition is met, the program continues as in the previous figure, but otherwise, the new program tries to recover somehow. On line 14, the code is finishing inside the reference cell, and, therefore, can use the current cell to check if the correct world space position corresponds to a reference position in the current cell. If the current cell does contain the correct reference position, the correct reference space position is computed via T_i^{-1} , but if the current cell does contain the correct reference position, the naive scheme is used. On line 33, the position is currently on a boundary so use of T_i^{-1} is inappropriate; thus, the naive scheme is used immediately on line 34.

Finally, in Figure 8, we provide an overload for the naive scheme. This is provided for clarity and does not require comment.

10 PARTICLES

In this section, we will provide the complete programs used to make the particle system figures. The position addition overload in these programs does not differ from Figure 6 nor do the creation of types or finite element inputs. Ergo, in this section, we will focus on the changes to the core particle systems programs; besides the addition of the overloaded functions and finite element inputs, how many lines were changed in porting these programs? How much did the main loop change? We stress that one should examine the prior work on Diderot to understand the full particle system programs and that here we mainly seek to point out the limited extent to which FEM versions differ from the original programs.

We first consider the shorter particle systems program used to create the isosurface in the paper, displayed in Figure 9. Lines 1 through 45 are basically identical to those for guided search in Figure 6. Line 46 through 52 implement position subtraction, which is a standard position operation, via taking world space differences if both positions are valid, and otherwise returning zero. Lines 54 through 168 implement a small particle system for an isosurface. Lines 54 through 58 are parameters to the system as in the original program. Lines 61 through 89 implement feature strength, feature step, and feature perpendicular functions that sample positions; these functions are considered inputs to the particle system code. These are different from the original versions of these functions, but they are trivially the same only sampling through the reference space via unpacking the position, accessing the current cells, and acquiring the transformed reference fields described in the paper. In terms of lines of code, each differs by 5 to 10 lines from the original, leading to an additional roughly 10 to 20 lines, but, we note these functions live outside the core particle system code. The core of the particle systems program lies in lines 89 through 168. We observe that this is basically identical to the original program except for 7 lines (lines 96, 97, 105, 128, 144, 163, 166). Each line has a comment explaining the change relative to the old program, but we note that 4 lines differ by type, two lines use the validity method of a position to do border control, and only lines 163 and 166 implement new functionality. In short, for a core loop of 72 lines, only 2 to 4 were added or changed non-trivially. In short, the core logic of the small particle system program is basically unchanged in the conversion to use guided search modulo the addition of the overloads on positions and the specification of finite element data. Examining our analysis, we see that besides the FEM inputs and position overloads, the program features at most 30 lines of additions or changes.

```

1 #version 3.0
2 //mesh specification:
3 type mesh mesh_t = file("evalProg.json");
4 //space (V) specification:
5 type functionSpace[mesh_t][dim] fns_t = file("evalProg.json");
6 //u_{V} specification:
7 type femFunction[fns_t] func_t = file("evalProg.json");
8 const int dim = mesh_t.dim; //dim of mesh
9
10 input mesh_t mesh; //input mesh
11 input fns_t space = fns_t(mesh); //input space
12 input func_t data = func_t(space); //input u_{V}
13 refCell{mesh_t} K = mesh.refcell; //get refCell
14
15 input int timeSteps=32; //guided search step limit
16 input real timeEps = 0.0000001; //1 - time max
17 //Overload + operator (see text for comments):
18 overload position{mesh_t} + (position{mesh_t} x,
19 tensor[dim] worldDelta) {
20 if (!x.isValid){ return(x);}
21 real time = 1;
22 position{mesh_t} cmp = x;
23 foreach (int i in 0..timeSteps ){
24 tensor[dim, dim] iJac = inv(V⊗(cmp.mc.transform))(cmp.refPos);
25 tensor[dim] refDelta = iJac • worldDelta;
26 tensor[dim] nPos = cmp.refPos + time * refDelta;
27 if (K.isInside(nPos)) {
28 return(cmp.mc.meshPos(nPos));
29 } else {
30 tensor[dim] nRefDelta = normalize(refDelta);
31 real eTime = K.exit(cmp, nRefDelta);
32 time -=eTime / |refDelta|;
33 if(eTime == -1){
34 return(mesh.findPos(x.worldPos() + worldDelta));
35 }
36 position{mesh_t} nmp = K.exitPos(cmp, nRefDelta);
37 if ( !nmp.isValid || time < timeEps){
38 return(nmp);
39 }
40 cmp = nmp;
41 }
42 }
43 return(mesh.findPos(x.worldPos() + worldDelta));
44 }
45 //evaluate u_{V} at a pos
46 function tensor[dim] nV(position{mesh_t} x){
47 if(x.isValid){ //border control
48 cell{mesh_t} c = x.mc; //get cell
49 tensor[dim] ref = x.refPos(); //get refPos
50 //f_{i}(y) where y = x.refPos()
51 tensor[dim] val = data.funcCell(c).refField(ref);
52 return(val/|val|);
53 } else {return(zeros[dim]);} //border control
54 }
55 input tensor[dim][] startPoints; //streamline start points
56 input real stepSize = 0.01; //RK2 stepsize
57 input int stepMax = 32; //RK2 steps max
58
59 strand line(tensor[dim] startPos){
60 output tensor[dim][] stream = {};
61 int step = 0;
62 //find position corresponding to
63 //global position of startPos
64 position{mesh_t} cPos = mesh.findPos(startPos);
65 update { //trace if position is valid
66 if(!cPos.isValid() || step == stepMax){stabilize;}
67 stream = stream@{cPos.worldPos()};
68 position{mesh_t} intermed = cPos + 0.5 * stepSize * nV(cPos);
69 if(!intermed.isValid){stabilize;} //validity of substep
70 cPos = cPos + stepSize * nV(intermed);
71 step+=1;
72 }
73 } create_collection {line(x) | x in startPoints};

```

Figure 6: A complete RK2 streamline program that uses guided search

This result is repeated with the larger particle system program, which is featured for ridge surfaces in Figure 10 and Figure 11. Combining these figures yields the full program. In the first part of the code, Figure 10, we find the particle system controls, guided search controls, and particle system auxiliary functions. Lines 1 through 53 provide FEM inputs and guided search, as in the previous examples. Lines 54 through 75 provide the particle system controls. Lines 78 through 126 provide the feature step, perpendicular, strength, mask, and test functions that these particle systems take as inputs. As before, these are the same as their vanilla versions, but they take positions, unpack them, and sample from the reference cell. We don't provide additional commentary on these functions as the conceptual extent to which they differ from the original Diderot programs is the same as in the previous particle system. The step, perpendicular, and strength functions require a few additional lines (at most 5 each) where as the mask, test, and posTest functions change by 1 line, 1 line, and 2 lines respectively. Thus lines 78 through 126 feature at most 20 additions or changes to the code. Lines 127 through 180 are almost identical to the original code, but the v3rand and genID functions (lines 175 and 166) need to use positions instead of vectors, creating another 4 changes. Moving on to the second part of the large particle system, Figure 11, we find the core particle system code. In Figure 11, all comments have been removed except those that indicate that a change has been made from the original program. We find that there are no more than 7 lines of changes and that only the last two lines, which check that all positions are valid before allowing convergence, are substantial changes. Combining this with our analysis of the first part and, as before, discounting the added overloads and FEM inputs, the total changes between this program and its regular grid version amount to fewer than 35 lines of code.

```

1  input real errorMax = 1.0;
2  overload position{mesh_t} + (position{mesh_t} x,
3                               tensor[dim] worldDelta) {
4      if (!x.isValid){ return(x);}
5      real time = 1;
6      position{mesh_t} cmp = x;
7      foreach (int i in 0..timeSteps ){
8          tensor[dim, dim] iJac = inv(V⊗(cmp.mc.transform)(cmp.refPos));
9          tensor[dim] refDelta = iJac • worldDelta;
10         tensor[dim] nPos = cmp.refPos + time * refDelta;
11         if (K.isInside(nPos)) {
12             position{mesh_t} nmp = cmp.mc.meshPos(newPos);
13             //check error:
14             if(!nmp.worldPos() - (x.worldPos() + dPos) | > errorMax){
15                 vec3 guess = (cmp.worldPos() + time * dPos);
16                 if(cmp.mc.isInside(guess)){ //guess current cell
17                     return(cmp.mc.meshPos(cmp.mc.inverseTransform(guess)));
18                 } else { //default to naive scheme
19                     return(meshData.findPos(x.worldPos() + dPos));}
20             }
21         return(nmp);
22     } else {
23         tensor[dim] nRefDelta = normalize(refDelta);
24         real eTime = K.exit(cmp, nRefDelta);
25         time=eTime / |refDelta|;
26         if(eTime == -1){
27             return(mesh.findPos(x.worldPos() + worldDelta));
28         }
29         position{mesh_t} nmp = K.exitPos(cmp, nRefDelta);
30         if (!nmp.isValid){return nmp;} //left the mesh
31         vec3 truth = (x.worldPos() + dPos * (1-time));
32         real error = |nmp.worldPos() - truth|;
33         if(error > errorMax){
34             //if too much error, naive scheme:
35             return(meshData.findPos( (x.worldPos() + dPos)));
36         }
37         if (time < timeEps){ //ran out of time.
38             return(nmp);
39         }
40         cmp = nmp;
41     }
42 } //spent too much time - use naive scheme
43 return(mesh.findPos(x.worldPos() + worldDelta));
44 }

```

Figure 7: A guided search implementation that check errors. This code can be substituted into Figure 6 in place of the addition overload to create a error checking guided search RK2 program.

```

1  overload position{mesh_t} + (position{mesh_t} x,
2                               tensor[dim] deltaPos)
3  {
4      if (!x.isValid()){
5          return(x);
6      }
7      return(meshData.findPos(x.worldPos() + dPos));
8  }

```

Figure 8: An overloaded position operator that implements the naive scheme. This code can be substituted into Figure 6 in place of the addition overload to create a naive scheme RK2 program.


```

1 #version 3.0
2 //mesh specification:
3 type mesh mesh_t = file("evalProg.json");
4 //space (V) specification:
5 type functionSpace{mesh_t}[dim] fns_t = file("evalProg.json");
6 //u_{V} specification:
7 type femFunction{fns_t} func_t = file("evalProg.json");
8 const int dim = mesh_t.dim; //dim of mesh
9
10 input mesh_t mesh; //input mesh
11 input fns_t space = fns_t(mesh); //input space
12 input func_t data = func_t(space); //input u_{V}
13 refCell{mesh_t} K = mesh.refcell; //get refCell
14
15 input int timeSteps=32; //guided search step limit
16 input real timeEps = 0.0000001; //1 - time max
17 overload position{mesh_t} + (position{mesh_t} x,
18     tensor{dim} worldDelta) {
19     if (!x.isValid){ return(x);}
20     real time = 1; //time
21     position{mesh_t} cmp = x; //current mesh pos
22     foreach (int i in 0..timeSteps){
23         tensor{dim, dim} iJac = inv(V⊗(cmp.mc.transform)(cmp.refPos));
24         tensor{dim} refDelta = iJac • worldDelta; //reference velocity
25         tensor{dim} nPos = cmp.refPos + time * refDelta;
26         if (K.isInside(nPos)) { //inside K
27             return(cmp.mc.meshPos(nPos)); //found x+v as (cmp.mc, nPos)
28         } else { //we left the reference cell - compute when:
29             tensor{dim} nRefDelta = normalize(refDelta);
30             real eTime = K.exit(cmp, nRefDelta);
31             time -= eTime / |refDelta|; //update time remaining.
32             if (eTime == -1){ //invalid direction, use naive scheme
33                 return(mesh.findPos(x.worldPos() + worldDelta));
34             } //find the exit location in the next cell.
35             position{mesh_t} nmp = K.exitPos(cmp, nRefDelta);
36             if ( !nmp.isValid || time < timeEps){ //left mesh or
37                 return(nmp); //ran out of time.
38             }
39             cmp = nmp;
40         }
41     } //spent too much time - use naive scheme
42     return(mesh.findPos(x.worldPos() + worldDelta));
43 }
44
45 //simple position subtraction operations
46 overload tensor{dim} - (position{mesh_t} x, position{mesh_t} y)
47 {
48     if (x.isValid && y.isValid {
49         return(x.worldPos() - y.worldPos());
50     } else {return(zeros{dim});}
51 }
52
53 //particle system controls
54 input real rad = 0.01;
55 input real eps = 0.01;
56 input real v0 = 0.0;
57 input tensor{dim}[] ipos;
58
59
60
61 function tensor{dim} fStep(position{mesh_t} y){
62     if(y.isValid){
63         tensor{dim} x = y.refPos; //border control
64         cell{mesh_t} c = y.mc; //get ref pos
65         cell{func_t} f = data.funcCell(c); //get the cell
66         vec3 grad = V(f.transformedRefField)(x); //get element info
67         tensor{dim} ret = (v0 - (f.refField)(x)) * grad / (grad • grad); //sample grad
68         return ret; //return newton step
69     } else {return [∞,∞,∞];} //a big step
70 }
71
72 function tensor{dim, dim} fPerp(position{mesh_t} y){
73     if (y.isValid){
74         tensor{dim} x = y.refPos; //border control
75         cell{mesh_t} c = y.mc; //get ref pos
76         cell{func_t} f = data.funcCell(c); //get the cell
77         vec3 norm = normalize(V(f.transformedRefField)(x)); //get element info
78         return identity{dim} - norm ⊗ norm; //return fPrep
79     }
80 } else return(zeros{dim, dim});
81 }
82 function real fStrength(position{mesh_t} y){
83     if(y.isValid{
84         cell{mesh_t} c = y.mc; //border control
85         cell{func_t} f = data.funcCell(c); //get element info
86         return |V(f.transformedRefField)(y.refPos)|;
87     } else {return(0.0);}
88 }
89 //particle system core code:
90 function real phi(real r) = (1-r)^4;
91 function real phi'(real r) = -4 * (1-r)^3;
92 function real enr(tensor{dim} x) = phi(|x|/rad);
93 function tensor{dim} frc(tensor{dim} x) = phi'(|x|/rad) * (1/rad) * x/|x|;
94
95
96 strand particle(position{mesh_t} pos0, real hh0){ //changed the type
97     output position{mesh_t} pos = pos0; //changed the type
98     real hh = hh0;
99     tensor{dim} step = zeros{dim};
100     bool found = false;
101     int nfs = 0;
102     int test = 1;
103     int testp = 0;
104     update {
105         if(!pos.isValid || fStrength(pos) == 0 || hh == 0){ //add border control
106             die;
107         }
108         if(!found) {
109             step = fStep(pos);
110             pos = pos + step;
111             if(|step|/rad > eps){
112                 nfs += 1;
113                 if(nfs > 10) { die; }
114             } else { found = true; testp=1; }
115         }
116         else {
117             pos = pos + fStep(pos);
118             step = zeros{dim};
119             real oldE = 0;
120             tensor{dim} force = zeros{dim};
121             int nn = 0;
122
123             foreach (particle P in sphere(rad)){
124                 oldE += enr(P.pos - pos);
125                 force += frc(P.pos - pos);
126                 nn += 1;
127             }
128             if (0 == nn && pos.isValid){ //added border control
129                 new particle(pos + [0.5*rad, 0,0 ], hh);
130                 continue;
131             }
132             force = fPerp(pos) • force;
133             tensor{dim} es = hh*force;
134             if(|es| > rad){
135                 hh *= rad/|es|;
136                 es *= rad/|es|;
137             }
138             position{mesh_t} samplePos = pos + es; //changed the type
139             tensor{dim} fs = fStep(samplePos);
140             if (|fs|/|es| > 0.5){
141                 hh *= 0.5;
142                 continue;
143             }
144             position{mesh_t} oldPos = pos; //changed the type
145             pos += fs + es;
146             real newE = sum {enr(pos - P.pos) | P in sphere(pos, rad)};
147             if (newE - oldE > 0.5 * (pos - oldPos) • (-force) ) {
148                 pos = oldPos;
149                 hh *= 0.5;
150                 continue;
151             }
152
153             hh *= 1.1;
154             step = fs + es;
155             if (nn < 5){
156                 new particle(pos + 0.5 * rad * normalize(es), hh);
157             }
158         }
159     }
160 }
161
162 update { //add an extra check (2 lines):
163     bool allValid = all {P.pos.isValid | P in particle.all};
164     bool allFound = all {P.found | P in particle.all};
165     real maxStep = max {|P.step| | P in particle.all};
166     if (allFound && allValid && maxStep/rad < eps) {stabilize;}
167 }
168 create_collection {particle(mesh.findPos(x), 1) | x in ipos};

```

Figure 9: A complete though minimal particle system that uses guided search and is aimed at isosurfaces. We note that the figure in the paper used this program with $\text{eps}=0.005$, $\text{rad}=0.5$, and $\text{iso}=0.0$.

```

1 #version 3.0
2 //mesh specification:
3 type mesh mesh_t = file("evalProg.json");
4 //space (V) specification:
5 type functionSpace{mesh_t}[dim] fns_t = file("evalProg.json");
6 //u_{V} specification:
7 type femFunction{fns_t} func_t = file("evalProg.json");
8 const int dim = mesh_t.dim; //dim of mesh
9
10 input mesh_t mesh; //input mesh
11 input fns_t space = fns_t(mesh); //input space
12 input func_t data = func_t(space); //input u_{V}
13 refCell{mesh_t} K = mesh.refcell; //get refCell
14
15 input int timeSteps=32; //guided search step limit
16 input real timeEps = 0.0000001; //1 - time max
17 //Overload + operator (see text for comments):
18 overload position{mesh_t} + (position{mesh_t} x,
19 tensor{dim} worldDelta) {
20 if (!x.isValid){ return(x);}
21 real time = 1;
22 position{mesh_t} cmp = x;
23 foreach (int i in 0..timeSteps){
24 tensor{dim, dim} iJac = inv(V⊗(cmp.mc.transform)(cmp.refPos));
25 tensor{dim} refDelta = iJac * worldDelta;
26 tensor{dim} nPos = cmp.refPos + time * refDelta;
27 if (K.isInside(nPos)) {
28 return(cmp.mc.meshPos(nPos));
29 } else {
30 tensor{dim} nRefDelta = normalize(refDelta);
31 real eTime = K.exit(cmp, nRefDelta);
32 time-=eTime / |refDelta|;
33 if(eTime == -1){
34 return(mesh.findPos(x.worldPos() + worldDelta));
35 }
36 position{mesh_t} nmp = K.exitPos(cmp, nRefDelta);
37 if (!nmp.isValid || time < timeEps){
38 return(nmp);
39 }
40 cmp = nmp;
41 }
42 }
43 return(mesh.findPos(x.worldPos() + worldDelta));
44 }
45
46 //Overload - operator
47 overload tensor{dim} - (position{mesh_t} x, position{mesh_t} y)
48 {
49 if (x.isValid && y.isValid {
50 return(x.worldPos() - y.worldPos());
51 } else {return(zeros{dim});}
52 }
53 //particle system controls
54 input real fStrTh ("Feature strength threshold");
55 input real fMaskTh ("feature mask threshold") = 0;
56 input real fBias ("Bias in feature strength computing") = 0.0;
57 input real tipd ("Target inter-particle distance") = 1.0;
58 input real mabd ("Min allowed birth distance (> 0.7351)") = 0.75;
59 input real travMax ("Max allowed travel to or on feature") = 10;
60 input int nfsMax ("Max allowed # feature steps") = 20;
61 input real gdeTest ("Scaling in sufficient decrease test") = 0.5;
62 input real gdeBack ("How to scale stepsize for backtrack") = 0.5;
63 input real gdeOppor ("Opportunistic stepsize increase") = 1.2;
64 input real fsEps ("Conv. thresh. on feature step size");
65 input real geoEps ("Conv. thresh. on system geometry") = 0.1;
66 input real mvmtEps ("Conv. thresh. on point movement") = 0.01;
67 input real rpcEps ("Conv. thresh. on recent pop. changes") = 0.01;
68 input real pcmvEps ("Motion limit before PC") = 0.3;
69 input real isoval ("Which isosurface to sample") = 0;
70 input int verb ("Verbosity level") = 0;
71 input real sfs ("Scaling (<=1 for stability) on fStep") = 0.5;
72 input real hist ("How history matters for convergence") = 0.5;
73 input int pcp ("periodicity of population control (PC)") = 5;
74 input vec3[] ipos ("Initial point positions");
75 input int fDim = 2;
76
77 //Freature Functions:
78 function vec3 fStep(position{mesh_t} pos) {
79 if(pos.isValid){
80 vec3 x = pos.refPos;
81 cell{mesh_t} c = pos.mc;
82 cell{func_t} f = data.funcCell(c);
83 vec3 g = V(f.transformedRefField)(x);
84 tensor{dim, dim} H = V⊗V(f.transformedRefField)(x);
85 vec3[3] E = evecs(H);
86 real[3] L = evals(H);
87 vec3 up = -(1/L[2])*E[2]⊗E[2] % g;
88 return up;
89 } else {return([∞,∞,∞]);}
90 }
91
92 function tensor[3,3] fPerp(position{mesh_t} pos) {
93 if(pos.isValid){
94 vec3 x = pos.refPos;
95 cell{mesh_t} c = pos.mc;
96 cell{func_t} f = data.funcCell(c);
97 tensor{dim, dim} H = V⊗V(f.transformedRefField)(x);
98 vec3 E2 = evecs(H)[2];
99 mat3 m = identity[3] - E2⊗E2;
100 return m;
101 } else {return(zeros{dim, dim});}
102 }
103
104 function real fStrength(position{mesh_t} pos) {
105 if(pos.isValid){
106 vec3 x = pos.refPos;
107 cell{mesh_t} c = pos.mc;
108 cell{func_t} f = data.funcCell(c);
109 vec3 g = V(f.transformedRefField)(x);
110 tensor{dim, dim} H = V⊗V(f.transformedRefField)(x);
111 real str = -evals(H)[2]/(fBias + |g|);
112 return str;
113 } else {return(0.0);}
114 }
115
116 function real fMask(position{mesh_t} x) = 0.0;
117 function bool fTest(position{mesh_t} x) = true;
118
119 function bool posTest(position{mesh_t} x) =
120 (x.isValid // Valid test replaces border control
121 && fStrength(x) > fStrTh // possibly near feature
122 && fMask(x) >= fMaskTh // meets feature mask
123 && fTest(x)); // passes addtl feature criterion
124
125 //End feature functions
126 //Auxiliary Particle system stuff:
127 int nnmin = 6 if (2==fDim) else 2 if (1==fDim) else 0;
128 int nnmax = 8 if (2==fDim) else 3 if (1==fDim) else 0;
129
130 function real phi(real r) {
131 real s=r-2.0/3;
132 return
133 1 + r*(-5.646 + r*(11.9835 + r*(-11.3535 + 4.0550625*r)))
134 if r < 2.0/3 else
135 -0.001 + ((0.09 + (-0.54 + (1.215 - 0.972*s)*s)*s)*s
136 if r < 1 else 0;
137 }
138 function real phi'(real r) {
139 real t=3*r-2;
140 return
141 -5.646 + r*(23.967 + r*(-34.0605 + 16.22025*r))
142 if r < 2.0/3 else
143 0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))
144 if r < 1 else 0;
145 }
146 real phiWellRad = 2/3.0;
147 real rad = tipd/phiWellRad;
148 function real enr(vec3 x) = phi(|x|/rad);
149 function vec3 frc(vec3 x) = phi'(|x|/rad) * (1/rad) * x/|x|;
150
151 real pchist = hist^(1.0/(2*pcp));
152
153 int iter = 0;
154 real rpc = 1;
155 int popLast = -1;
156
157 function real urnd(real x) {
158 if (x==0) return 0;
159 real l2 = log2(|x|);
160 real frxp = 2^(l2-floor(l2)-1);
161 return fmod((2^20 + 2*iter)*frxp, 1);
162 }
163
164 function real v3rnd(position{mesh_t} p){ //type change
165 if(!p.isValid){ //border control
166 return 0;
167 }
168 vec3 v = p.refPos; //get ref pos
169 return(fmod(urnd(v[0]) + urnd(v[1]) + urnd(v[2]), 1));
170 }
171
172 //Another type change:
173 function real genID(position{mesh_t} v) = floor(1000000*v3rnd(v));
174
175 function int pCIter(){
176 if (pcp>0 && iter>0 && 0 == iter % pcp) {return (((iter/pcp)%2)*2 - 1);}
177 else {return 0;}
178 }
179
180 }

```

Figure 10: Part 1 of a particle system that uses guided search and is aimed at ridge surfaces. This section contains the search and the particle system parameters. The parameters used to produce the figure in the paper are $fStrTh=24$, $fBias=0.1$, $tipd=0.1$, $fsEps=geoEps=mvmtEps=0.1$, $rpcEps=0.01$, $pcmvEps=0.3$, $sfs=hist=0.5$, $pcp=5$.

```

1 strand point (position{mesh_t} p0, real hh0) { //changed the type
2 output position{mesh_t} pos = p0; // changed the type
3 real ID = genID(p0);
4 real hh = hh0;
5 vec3 step = [0,0,0];
6 bool found = false;
7 int nfs = 0;
8 real trav = 0;
9 real mvmt = 1;
10 real closest = rad;
11 int born = 0;
12 bool first = true;
13 update {
14   if (!posTest(pos)) {
15     die;
16   }
17   if (travMax > 0 && trav > travMax) {
18     die;
19   }
20   if (!found) {
21     if (nfsMax > 0 && nfs > nfsMax) {
22       die;
23     }
24     step = sfs*fStep(pos);
25     pos = pos + step;
26     mvmt = lerp(|step|/tipd, mvmt, hist);
27     if (mvmt > fsEps) {
28       trav += |step|/tipd;
29       nfs += 1;
30     } else {
31       found = true;
32       mvmt = 1;
33       trav = 0;
34     }
35   } else {
36
37     if (0 == fDim) { stabilize; }
38     step = sfs*fStep(pos); pos = pos + step; trav += |step|/tipd;
39     real oldE = 0;
40     vec3 force = [0,0,0];
41     int nn = 0;
42     foreach (point P in sphere(rad)) {
43       vec3 off = P.pos - pos;
44       if (|off|/tipd < fsEps && ID <= P.ID) {
45         die;
46       }
47       oldE += enr(off);
48       force += frc(off);
49       nn += 1;
50     }
51     if (0 == nn) {
52       if (!( pcIter() > 0 && born < nnmax )) { continue; }
53       vec3 noff0 = fPerp(pos)*[tipd,0,0];
54       vec3 noff1 = fPerp(pos)*[0,tipd,0];
55       vec3 noff2 = fPerp(pos)*[0,0,tipd];
56       vec3 noff = noff0;
57       noff = noff if |noff| > |noff1| else noff1;
58       noff = noff if |noff| > |noff2| else noff2;
59       //changed the type:
60       position{mesh_t} npos = pos + tipd*normalize(noff);
61       npos = npos + sfs*fStep(npos);
62       if (posTest(pos)) {
63         new point(npos, hh); born += 1;
64       }
65       continue;
66     }
67     vec3 es = hh*fPerp(pos) % force;
68     if (|es| > tipd) {
69       hh *= tipd/|es|;
70       es *= tipd/|es|;
71     }
72     vec3 fs = sfs*fStep(pos+es);
73     if (|fs|/(fsEps*tipd + |es|) > 0.5) {
74       hh *= 0.5;
75       continue;
76     }
77     //changed the type:
78     position{mesh_t} oldpos = pos;
79     vec3 up = fs + es;
80     pos = pos + up;
81     real newE = 0;
82     closest = rad;
83
84     vec3 mno = [0,0,0];
85     nn = 0;
86     foreach (point P in sphere(rad)) {
87       vec3 off = P.pos - pos;
88       newE += enr(off);
89       closest = min(closest, |off|);
90       mno += off;
91       nn += 1;
92     }
93     mno /= nn;
94
95     if (newE - oldE > gdeTest*(pos - oldpos)*(-force)) {
96       hh *= gdeBack;
97       if (0 == hh) {
98         die;
99       }
100      pos = oldpos;
101      continue;
102    }
103    hh *= gdeOppor;
104    step += fs + es;
105    trav += |step|/tipd;
106    mvmt = lerp(|step|/tipd, mvmt, hist);
107    if (|step|/tipd < pcmvEps && pcIter() != 0) {
108      if (pcIter()>0
109          && newE<0
110          && nn<nnmin
111          && born<nnmax) { //changed the type:
112        position{mesh_t} npos = pos + (-tipd*normalize(mno));
113        npos = npos + sfs*fStep(npos); npos = npos + sfs*fStep(npos);
114        bool birth = true;
115        if (fDim == 2 && nn >= 4) {
116          foreach (point P in sphere(npos, tipd*mabd)) {
117            birth = false;
118          }
119          if (birth) {
120            birth = v3rnd(pos) < (nnmin - nn)/real(nnmin);
121          }
122        }
123        if (birth && posTest(npos)) {
124          new point(npos, hh); born += 1;
125        }
126      } else if (pcIter() < 0 && newE > 0 && nn > nnmax) {
127        if (v3rnd(pos) < (nn - nnmax)/real(nn)) {
128          die;
129        }
130      }
131    }
132    }
133    first = false;
134  }
135 }
136
137 update {
138   int pop = numActive();
139   int pc = 1 if pop != popLast else 0;
140   rpc = lerp(pc, rpc, pchist);
141   bool allfound = all { P.found | P in point.all };
142   real percfound =
143     100* mean { 1.0 if P.found else 0.0 | P in point.all };
144   real meancl = mean { P.closest | P in point.all };
145   real varicl = mean { (P.closest - meancl)^2 | P in point.all };
146   real covcl = sqrt(varicl) / meancl;
147   real maxmvmt = max { P.mvmt | P in point.all };
148   //added new convergence test
149   //to find avoid saving invalid positions:
150   bool allValid = all { P.pos.isValid | P in point.all };
151   if (allfound
152       && covcl < geoEps
153       && maxmvmt < mvmtEps
154       && rpc < rpcEps && allValid) { // use this new test.
155     stabilize;
156   }
157   iter += 1;
158   popLast = pop;
159 }
160 create_collection { point(mesh.findPos(p), 1) | p in ipos};

```

Figure 11: Part 2 of a particle system that uses guided search and is aimed at ridge surfaces. This section contains the core of a Diderot program: the strand definition.