Point Movement in a DSL for Higher-Order FEM Visualization

Teodoro Collin\textsuperscript{1} Charissee Chiw\textsuperscript{2} L. Ridgway Scott\textsuperscript{1} John Reppy\textsuperscript{1} Gordon Kindlmann\textsuperscript{1}

\textsuperscript{1}Department of Computer Science, University of Chicago\textsuperscript{*} \textsuperscript{2}Galois, Inc.\textsuperscript{2}

\textbf{ABSTRACT}

Scientific visualization tools tend to be flexible in some ways (e.g., for exploring isosurfaces) 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 quantities within a mesh, we took an existing visualization domain-specific language, and added a mesh position and one form of data: regular grids [8, 10, 17, 22, 23, 35]. In their and support exploration of the relevant parameter spaces. then compile programs that both run efficiently on the given data and a separate description of how data and fields are formed, and that can take a high-level specifications of a visualization method, on a grid or mesh. Visualization research may benefit from systems ODEs, regardless of how exactly scalar or vector fields are defined for visualizing large biomedical imaging volumes are sensibly spe-

\textbf{Index Terms:} Software and its engineering—Software notations and tools—Context specific languages—DSLs

\section{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 character-

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1}
\caption{Diagram showing the relationship between data and visualization.}
\end{figure}

visualization algorithms specialized to finite elements avoid the cost of meshing existing visualization methods, or exploring the value of new methods (FEM) meshes on which those phenomena are simulated. 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 quantities within a mesh, we took an existing visualization domain-specific language, and added a mesh position and one form of data: regular grids [8, 10, 17, 22, 23, 35]. In their and support exploration of the relevant parameter spaces. then compile programs that both run efficiently on the given data and a separate description of how data and fields are formed, and that can take a high-level specifications of a visualization method, on a grid or mesh. Visualization research may benefit from systems ODEs, regardless of how exactly scalar or vector fields are defined for visualizing large biomedical imaging volumes are sensibly spe-

\textbf{Index Terms:} Software and its engineering—Software notations and tools—Context specific languages—DSLs

\section{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) \( u: \Omega \rightarrow \mathbb{R}^3 \), FEM uses a finite dimensional vector space of functions (function space) \( V \) to find an approximate solution, \( u_h \in V \), to \( u \). The space \( V \) is created by discretizing the world-space domain \( \Omega \) 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 \( CM \) 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 \( \Omega \). Within some \( K_i \subset \Omega \), the PDE solution being visualized \( u_h \) will be represented in a chosen basis \( \{ p_j \} \) for \( P \) as

\begin{equation}
\label{eq:1}
u_h(x)|_{K_i} = \sum_j c_j p_j(T_i^{-1}(x)).
\end{equation}

Using higher-order FEM, with non-linear \( T_i \), increases the computational cost for a \textit{naive} visualization algorithm to traverse just a single cell \( i \), since each of the many evaluations of \( T_i^{-1} \) in \( (1) \) require 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 $T^{-1}$ by replacing, when possible, world-space evaluation of the approximate solution $u_T$ via (1) with

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

where $f_i: K \to \mathbb{R}^n$ is the evaluation of $u_T$ 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., color mapping $u_T$), 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 $u_T, f_i, K$ 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]. Jalepalli 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 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 $u_T(x)$ 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 $u_T(x + v)$ 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 $T_i^{-1}(x + v) \approx T_i^{-1}(x) + (DT_i^{-1})^T v$. Second, the location where $T_i^{-1}(x) + (DT_i^{-1})^T v$ 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 $\sigma$ 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 $T_i(\sigma(x))$ and $T_j(x)$. 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 in 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 $|T_i(T_i^{-1}(x) + t(DT_i^{-1})^T v) - (x + tv)|$ 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 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
The full analysis is in the supplementary materials. A 80-line isosurfacing particle program, and 40 lines changed in a copy-pasted part. Disruption to existing code was minimal: 15 types from vectors to positions. While this seems extensive, besides point movement code (Fig. 2), changed field sampling to a function forward transformations. We declared FEM types and inputs, added evaluation and the details of point movement. In particular, we were consistent with adoption of many previous Diderot programs to a FEM context that adding or changing around 5000 lines of Standard ML, but this cost overloads "operator, the concise guided search implementation exists. With all this in place, positions can become arguments to an overload position. And to learn the corresponding position in the neighboring cell (if it exists). Meyer et al. also sample gradients and trees [37]. We also added queries on the reference cell geometry, for particle systems) can query the state of their neighbors via k-d 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 \( D^k f_i \) at \( T_i(x) \), so that tensor-valued (world-space) derivatives of \( T_i \) 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 Fig. 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 straightforward 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.

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.
were found after experimenting with parameters (feature strength which could create discontinuities in the ridge surface itself. A 300-

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].

<table>
<thead>
<tr>
<th>Step Size (columns)</th>
<th>Error Parameter (rows)</th>
<th>Error Difference (seconds)</th>
<th>Speedup Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>10^{-4}</td>
<td>0.036s (2.524)</td>
<td>0.049s (2.220)</td>
</tr>
<tr>
<td></td>
<td>10^{-5}</td>
<td>0.097s (9.183)</td>
<td>0.149s (6.735)</td>
</tr>
<tr>
<td>0.002</td>
<td>10^{-6}</td>
<td>5.353s (2.600)</td>
<td>6.530s (2.369)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.036s (2.524)</td>
<td>0.049s (2.220)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.097s (9.183)</td>
<td>0.149s (6.735)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.353s (2.600)</td>
<td>6.530s (2.369)</td>
</tr>
</tbody>
</table>

Figure 4: Particle based sampling of an isosurface in the form of a rounded cube, contained with 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. 4, showing a sampling of the isocontour $x^6 + y^6 + z^6 = 1$ in a cylindrical mesh with hexitic $P$. 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 hexitic $P$. 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

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


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.

2 Streamlines

We provide line by line commentary on Figure 1, 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 (nV) 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 nV_i, 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 though 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).

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.

3 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 1 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 4. Lines 1 through 45 are basically identical to those for guided search in Figure 1. 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.

This result is repeated with the larger particle system program, which is featured for ridge surfaces in Figure 5 and Figure 6.
bining these figures yields the full program. In the first part of the code, Figure 5, 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 pos* start 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 6, we find the core particle system code. In Figure 6, 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.
input real errorMax = 1.0;

tensor[dim] worldDelta = x;

if (x.isValid())
    return (x);

real time = 1;

foreach (int i in 0..timeSteps ){
    tensor[dim] nPos = cmp.refPos + time * refDelta;
    if(K.isInside(nPos)) {
        position{mesh_t} nmp = cmp.mc.meshPos(nPos);
        //check error:
        if(|nmp.worldPos() - cmp.worldPos()| > errorMax){
            vec3 guess = (cmp.worldPos() + time * dPos);
            if(cmp.mc.isInside(guess)){ //guess current cell
                return(cmp.mc.meshPos(cmp.mc.inverseTransform(guess)));
            } else { //default to naive scheme
                return(meshData.findPos((x.worldPos() + dPos)));}
        } else {
            return(nmp);
        }
    } else {
        tensor[dim] nRefDelta = normalize(refDelta);
        real eTime = K.exit(cmp, nRefDelta);
        time-=eTime / |refDelta|;
        if(eTime == -1){
            return(mesh.findPos(x.worldPos() + worldDelta));
        } position(mesh_t) nmp = K.exitPos(cmp, nRefDelta);
        if (!nmp.isValid) { //left the mesh
            vec3 truth = (x.worldPos() + dPos + (1-time));
            real error = |nmp.worldPos() - truth|;
            if(error > errorMax){
                //if too much error, naive scheme:
                return(meshData.findPos((x.worldPos() + dPos)));
            } if (time < timeEps) { //ran out of time.
                return(nmp);
            } cmp = nmp;
        } //spent too much time - use naive scheme
        return(mesh.findPos(x.worldPos() + worldDelta));
    }
}

Figure 2: A guided search implementation that check errors. This code can be substituted into Figure 1 in place of the addition overload to create a error checking guided search RK2 program.

overload position(mesh_t) + (position(mesh_t) x,

4 tensor[dim] deltaPos) {
5 if (!x.isValid())
6 return(x);
7 return(meshData.findPos(x.worldPos() + dPos));
8 }

Figure 3: An overloaded position operator that implements the naive scheme. This code can be substituted into Figure 1 in place of the addition overload to create a naive scheme RK2 program.
used this program with eps=0.005, rad=0.5, and iso=0.0.

{mesh specification:
  type mesh mesh_t = file("evalProg.json");
  // /space (F) specification:
  type functionSpace(mesh_t)[dim] fn_s_t = file("evalProg.json");
  \n  \n  type fenFunction(mesh_t)[dim] func_t = file("evalProg.json");
  \n  const int dim = mesh_t.dim; //dim of mesh

input nฝาก_t mesh_t; //input mesh
input fn_s_t space = fn_s_t(mesh); //input space
input func_t data = func_t(space); //input u_{(y)}
refCell(mesh_t) K = mesh.refcell; //get refCell

input int timeSteps=12; //guided search step limit
input real timeEps = 0.0000060; //time max
output position(mesh_t) + (position(mesh_t) x, tensor(dim) worldDelta) {
  if ((x.isValid) { return(x); });
  real time = 1.0; //time

  position(mesh_t) cmp = x; //current mesh pos
  foreach (int i in timeSteps) { //}
    tensor[dim] refDelta = 1.0c * worldDelta; //reference velocity
    tensor[nrs] posRef = cmp.REFPos(); //refAccel;
    if (K.isInside(nrs)) { //inside K
      refPos.pos = cmp.REFPos(nrs); //found x,y as cmp, mesh:
      refDelta.real = mesh.REFDelta(nrs); //get refDelta;
      real eTime = K.Outside(cmp, nrsRefDelta); //time,
      time -= eTime / refDelta; //update remaining time.
      if (eTime = -1) { //invalid direction, use naive scheme
        refPos.real = refPos.real + (mesh_t.REFDeltaP() + worldDelta);
      }
    }
    refPos.real = mesh.movePoint(cmp); //left mesh or
    refDelta.real = if (cmp.isValid()) { ranOutOfTime(); } else { }
    cmp = cmp + refDelta;
  }
  //spent too much time - use naive scheme
  return(refPos.real, mesh.newInstance());
}

//simple position subtraction operations
overload tensor(dim) - (position(mesh_t) x, position(mesh_t) y) {
  if (x.isValid && y.isValid {
    return(x.worldPos() - y.worldPos());
  } else { return(zeros(x, dim)); }
};

//particle system controls
input real rad = 0.91;
input real eps = 0.91;
input real v0 = 0.91;
input tensor[dim] ipos;

function tensor[dim] fStep(position(mesh_t) y) {
  if (y.isValid) { //border control
    tensor[dim] x = y.REFPos(); //get ref pos
    cell[(func_t) c = y.mc; //get the cell
    cell(func_t) f = data.funcCell(c); //get element info
    vec3 grad = V.f.transformedRefField(x); //sample grad
    tensor[dim] ret = (v0 - (f.transformedRefField)(x)) * grad / (grad * grad); //get new particle
    return ret; //return newton step
  } else { return(zeros(x, dim)); //a big step
  }
};

function tensor[dim] fPerp(position(mesh_t) y) {
  if (y.isValid) { //border control
    tensor[dim] x = y.REFPos(); //get ref pos
    cell[(func_t) c = y.mc; //get the cell
    cell(func_t) f = data.funcCell(c); //get element info
    vec3 normal = mesh.computedRefField((x)); //sample grad
    return identity(dim) - norm @ norm; //return fPerp
  } else { return(zeros(x, dim));
  }
};

function real fStrength(position(mesh_t) y) {
  if (y.isValid) { //border control
    cell[(func_t) c = y.mc; //get cell
    //particle system core code:
    function real phi(real r) = (r-1)²;
    function real phi(real r) = phi(r-
    function real en(tensor[dim] x) = phi([x]/rad); //get the en function
    function tensor[dim] frc(tensor[dim] x) = phi([x]/rad) * (1/rad) * x/|x|;
    //strand particle(position(mesh_t) pos, real hh){
      ch = 0.0; //changed the type
      output position(mesh_t) pos = pos6; //changed the type
      real hh = hh0;
      tensor[dim] step = zeros(dim);
      bool found = false;
      int nfs = 0;
      int test = 1;
      update {
        if (pos6.isValid || fStrength(pos) == 0 || hh == 0) { //add border control
          die;
        }
        if (found) { //step = fStep(pos);
          pos = pos + step;
          if (test++) { //first update
            if (nfs == 0) { //initialize
              real oldv = 0;
              tensor[dim] force = zeros(dim);
              int nn = 0;
              foreach (particle P in sphere(rad)) { //force
                oldv = enr(P.pos - pos); //force
                force += frc(P.pos - pos);
                nn++;
              }
            }
            if (0 == nn & pos.isInvalid()) { //add border control
              new particle(pos = [0.95, rad, 0.8], hh);
              continue;
            }
            force = fPerp(pos) * force;
            tensor[dim] es = hh * force;
            if (es > rad/10) { //check
              hh *= rad/10es;
              es *= rad/10es;
            }
            position(mesh_t) samplo = pos + es; //changed the type
            tensor[dim] fs = fStep(samplo); //get new particle
            if (fis[1] > 0.5) { //force
              hh *= 0.5;
              continue;
            }
            position(mesh_t) oldpos = pos; //changed the type
            pos = fs + es;
            real newv = sum (enr(pos - P.pos) | P in sphere(pos, rad));
            if (newv == oldv > 0.5 * (pos - oldpos)) { //factor }
              pos = oldpos;
              hh *= 0.5;
              continue;
            }
          } else { //check
            hh *= 1.1;
            step = fs + es;
            if (nn < 3) { //new particle
              new particle(pos + 0.5 * rad @ normal(es), hh);
            }
          }
        }
        update { //add an extra check (2 lines):
          bool allValid = all (P.pos.isValid | P in particle.all);
          bool allFound = all (P in particle.all);
          bool real maxStep = max (P.step) | P in particle.all;
          if (allFound && allValid && allStep < eps) { stabalize();
            create_collection (particle(mesh.findPos(x), 1) | in ipos);
          }
        }
      }
    }}

Figure 4: A complete though minimal particle system that uses guided search and is aimed at interfaces. We note that the figure in the paper used this program with eps=0.005, rad=0.5, and iso=0.0.
89 86 82 80 77 75 72 71 66 62 51 48 47 45 42 39 38 36 32 25 21 15 10
//input int
//input real
//input real
//input real
//input real
//input real
//input real
overload tensor
{mesh_t} K = mesh.refcell;
if ( pos.isValid){
  return (pos.mesh Pos(x.worldPos()) + worldDelta);
}
else
{
  return (cmp.mc.transform)(cmp.refPos);
}

//particle system controls
input int pcps ("Periodicity of population control (PC)") = 3;
input int intDim = 2;

//Further Functions:
function vec3 fstep(position(mesh_t) pos) {
  if(pos.isValid){
    vec3 x = pos.refPos;
    cell(mesh_t) c = pos.mc;
    cell(func_t) f = data.funcCell(c);
    tensor<dim, dim> H = V*(f.transformedRefField)(x);
    vec3 E = nevcs(0)[2];
    mat3 m = identity[3] - E*E;
    return (m*(pos.mesh Pos(x.worldPos())) + x);
  } else
  return ((0,0,0,0));
}

function real fstrength(position(mesh_t) pos) {
  if(pos.isValid){
    vec3 x = pos.refPos;
    cell(mesh_t) c = pos.mc;
    cell(func_t) f = data.funcCell(c);
    tensor<dim, dim> H = V*(f.transformedRefField)(x);
    vec3 E = nevcs(0)[2];
    mat3 m = identity[3] - E*E;
  } else return(zeros(dim, dim));
}

function real fmask(position(mesh_t) x) = 0.0;
function bool ftest(position(mesh_t) x) = true;
function bool postTest(position(mesh_t) x) =
  (x.isInside(x) && fMask(x) >= fMaskTh) // possibly near feature
  && fMask(x) >= fMaskTh // meets feature mask
  && fTest(x)); // passes addtl feature criterion

//Auxiliary Particle system stuff:
int nnmin = 6 if (2==Dim) else 2 if (1==Dim) else 0;
int nnmax = 8 if (2==Dim) else 2 if (1==Dim) else 0;
function real phi(real r) {
  r<2.0 / 3;
  return (0.5,64 + r*(11.935 + r*(-11.335 + 4.855625r)));
}
if r<0.0 / 3 else
  0.0111 + (-0.69 + (-1.215 + 0.9725^2)^3)^3)^3
}
if r<1.0 else;
}

real phiWellRad = 2.34;
real rad = tipd/psRad;
function real en(vec3 x) = phi((x/|x|));
function vec3 frc(vec3 x) = phi ((x/|x|) * (1/|rad|) * x/|x|);

real pchist = hist(1.6/(2*pcps));
int iter = 0;
real rpc = 1;
int poptast = -1;
function real urnd(real x) {
  if (x<0) return 0;
  real l2 = log((x));
  real frap = 2*(l2(log(2))-1);
  return fmod((2^28 + 2^iter)^frap, 1);
}

function real v3rnd(position(mesh_t) p){ //type change
if(p.isValid){
  //border control
  return 0;
}
vec3 v = p.refPos;

return fmod(urnd(v) + urnd(v)+1) + urnd(v), 1));
}

function real gen1D(position(mesh_t) x) = floor(1009690^xd(x));
function int pcIter(){
  if (pcp<9 & & iter<9 & & g == iter & & x) return ((iter+pcp+3)/2 - 1);
  else return (0);
}
Figure 5: Part 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, IFbias=0.1, fbd=0.01, fSps=eGeoEps=mvtEps=0.1, rpcEps=0.01, pcmvEps=0.3, sfst=hist=0.5, pcp=5.
strand point (position(mesh_t) p0, real hh0) { // changed the type
  output position(mesh_t) p0 = p0; // changed the type
  real ID = genID(p0);
  real hh = hh0;
  vec3 step = [0.8, 0.8];
  bool found = false;
  int nfs = 0;
  real trav = 0;
  real mvmt = 1;
  real closest = trav;
  int born = 0;
  bool first = true;
  update {
    if (!posTest(pos)) {
      die;
    } else if (travMax > 0 && trav > travMax) {
      die;
    } else if (!found) {
      if (nfsMax > 0 && nfs < nfsMax) {
        die;
      } else {
        step = sfs*fstep(pos);
        pos = pos + step;
        mvmt = lerpi(step/tipd, mvmt, hist);
        if (mvmt > fsEps) {
          trav += [step/tipd];
          nfs += 1;
        } else {
          found = true;
          mvmt = 1;
          trav = 0;
        }
      }
    } else {
      if (hh <= 0.8) {
        step = fsfstep(pos);
        pos = pos + step; trav += [step/tipd];
        real oldh = hh;
        vec3 force = [0,0,0];
        int nn = 0;
        foreach (point P in sphere(rad)) {
          vec3 off = P.pos - pos;
          if (!off/tipd < fsEps & ID < P.ID) {
            die;
          }
        }
        older = enr(off);
        force += inc(off);
        nn += 1;
      } else {
        if (hh >= 0.8) {
          die;
        } else {
          if (newN > 0 & & nn > nnmax) {
            continue;
          } else {
            vec3 es = hh*fsfstep(pos)*force;
            if (es > tipd) {
              hh = tipd/|es|;
            }
          }
        }
      }
    }
  }
  if (nn <= nnmin) {
    stabilize;
  }
  if (nn >= 100) {
    stabilize;
  }
  foreach (point P in sphere(rad)) {
    vec3 nosi = fPerp(pos)*[tipd,0,0];
    vec3 nosf = fPerp(pos)*[tipd,0,0];
    vec3 nosf2 = fPerp(pos)*[0,0,tipd];
    vec3 nosf1 = fPerp(pos)*[0,0,tipd];
    vec3 nosf off = nosi if ([off] > [off]) else nosf1;
    vec3 nosf = nosf2 if ([off] > [off]) else nosf2;
    if (nosf < nosf) {
      nosf = nosf1;
    } else {
      nosf = nosf2;
    }
    if (posTest(pos)) {
      new point(npos, hh); born += 1;
    }
  }
  continue;
}
vec3 es = hh*fsfstep(pos)*force;
if (es > tipd) {
  hh = tipd/|es|;
}
vec3 fs = fsfstep(pos-es);
if (fs/fstep(max(pos, es)) > 0.5) {
  hh = 0.5;
  continue;
}
// changed the type;
position(mesh_t) idlpos = pos;
vec3 up = fs = es;
pos = pos + up;
}

real newE = 0;
closest = rad;
vec3 nno = [8,0,0];
nn = 0;
foreach (point P in sphere(rad)) {
  vec3 off = P.pos - pos;
  newE += enr(off);
  closest = min(closest, |off|);
  mno += off;
  nn += 1;
  mno /= nn;
  if (newE < oldE) {
    gdeEps := (pos - oldpos)*(-force);
    hh = gdeRack);
    if (hh > 0) {
      pos = oldpos;
      continue;
    } else {
      if (nn >= nnmin) {
        stabilize;
      }
    }
    if (birth) {
      birth = vrand(pos) < (nnmin - nn)/real(nnmin);
    } else if (birth & postTest(pos)) {
      new point(npos, hh); born += 1;
    }
  }
}
if (pcIter() < 0 & & newE > 0 & & nn > nnmax) {
  if (vrand(pos) < (nn - nnmax)/real(nn)) {
    die;
  }
}
first = false;
}
update {
  int pop = numActive();
  int pc = 1 if pop = poplast else 0;
  rpc = lerpi(pc, rpc, pcHist);
  bool allfound = all [P.found | P in point.all];
  real perc = (mean(p.pos, P.pos) - P.pos) / P.pos;
  if (perc < 0.8) {
    continue;
  }
  real mmean = mean (P.closest - mean))/2 | P in point all;}
  real covcl = sqrt(varlc)/meancl;
  real maxmvmt = maxx (P.mvmt | P in point.all;)
  // added new convergence test
  // to find avoid saving invalid positions;
  bool allValid = all (Pnpos in valid | P in point.all);
  if (!allValid) {
    & & covcl < geoeps
    & & maxmvmt < wmveps
    & & rpc < rpcEps & allValid) { // use this new test.
      stabilize;
    }
}
iter += 1;
poplast = pop;
create_collection ( point(mesh.findPos(p, 1) | p in ipos);