Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis

Gordon Kindlmann Charisee Chiw Nicholas Seltzer Lamont Samuels John Reppy

Department of Computer Science, University of Chicago

Abstract— Many algorithms for scientific visualization and image analysis are rooted in the world of continuous scalar, vector, and tensor fields, but are programmed in low-level languages and libraries that obscure their mathematical foundations. Diderot is a parallel domain-specific language that is designed to bridge this semantic gap by providing the programmer with a high-level, mathematical programming notation that allows direct expression of mathematical concepts in code. Furthermore, Diderot provides parallel performance that takes advantage of modern multicore processors and GPUs. The high-level notation allows a concise and natural expression of the algorithms and the parallelism allows efficient execution on real-world datasets.

Index Terms—Domain specific language, portable parallel programming, scientific visualization, tensor fields.

◆

1 INTRODUCTION

Scientific visualization and image analysis research seeks better ways to extract knowledge from scientific data. However, writing software is hard: it can be difficult to translate ideas and algorithms to working code, often due to a *semantic gap* between the mathematical concepts at the computational core of algorithms, and their actual expression in source code. This often happens with algorithms defined in terms of abstractions like fields or tensors, but implemented with library function calls or interfaces that do not reflect the basic mathematical structures. Another challenge is that visualization and analysis tools typically require parallel computing, either because new analysis methods may require more computation per iteration, or because meaningful evaluation of new methods requires their application to large real-world datasets. Datasets grow in size and complexity with continuing advances in scientific imaging modalities.

Domain-specific languages (DSLs) can address both of these challenges. By addressing a more narrowly-defined class of data types and operations than general-purpose languages, DSLs bridge the semantic gap by supporting mathematically idiomatic expression of algorithms. By targetting a specific class of algorithms, DSLs also facilitate compiling algorithms to efficient parallel execution, as well as achieving *portable parallelism*: mapping the same program to different parallel computing back-ends. Scientific visualization is an especially apt target for DSLs because many of its elementary abstractions are not directly supported by general-purpose languages, so the implementation of even basic methods obscures their underlying algorithmic simplicity. While many languages support vector- or matrix-valued variables, for example, we are not aware of languages that directly and idiomatically support the abstraction of a continuous scalar, vector, or tensor *field*, along with arithmetic and differential operations on fields. Fields, however, are fundamental to the basic definition of many core scientific visualization algorithms (e.g. ray-cast volume rendering, streamline integration, and fiber tractography).

Diderot is a portable parallel domain-specific language developed out of frustration with the lack of mathematical abstraction and the difficulty of parallel programming associated with research in scientific visualization and image analysis. Previous work [16] described an early version of Diderot, introducing its program structure, intermediate representations in compilation, computational abstraction of fields, and efficient parallel execution via pthreads. This paper presents a range of advances:

- **Portable parallelism**: With no more user intervention than setting a compiler command-line flag, Diderot can generate sequential code using SSE vector instructions, pthreads-based parallel code for shared-memory multicore systems, or OpenCL code for GPUs. Portable parallelism is increasingly useful in the context of rapidly evolving architectures for parallel computing.
- **Strand creation and communication**: The threads of execution (called *strands*) in Diderot can spawn new strands, strands can communicate with each other, and global reductions can be computed over the entire program. These features are important for algorithms such as particle systems that require creating dynamically new particles, and computing the interactions between particles.
- **Higher-order mathematical functions**: Fields in Diderot (scalar, vector, or tensor) can be operated on and combined as they would be in mathematics, creating new fields with expressions like "`G=∇F`" or "`H=(∇×V)•V`." While the previous compiler [16] had some higher-order operators, here we describe a general system for lifting tensor operations to fields and combinations of fields and tensors. Supporting idiomatic expressions of these operations simplifies implementing algorithms that depend on them.
- **Compiling to a library**: Rather than compiling to a self-contained executable, Diderot programs may be compiled to a C library with an API for setting global variables, defining the initial state of the computation, controlling program iterations, and accessing the computed output. This mechanism facilitates integrating Diderot into existing environments written in C or C++, as well as high-level languages that support C extensions.
- **Other language conveniences**: The Diderot programs shown here benefit from other new language features such as user-defined functions, dynamic sequences (*e.g.*, for generating streamline geometry), and operations like `clamp`, `mirror`, and `wrap` on the convolution domains of fields.

2 BACKGROUND AND RELATED WORK

There are a number of approaches to supporting scientific visualization and biomedical-image processing applications. A common way is to build upon a domain-specific library or toolkit such as the Visualization Toolkit (VTK) [42], ParaView [2], and the Insight Toolkit (ITK) [53]. Related to libraries and toolkits are software tools that target a specific visualization method in a highly configurable way, such as the wealth of GPU-based volume rendering tools [11, 38, 34, 41].

- *All authors are with Dept. of Computer Science, University of Chicago. E-mails: {glk,cchiw,nseltzer,lamonts,jhr}@cs.uchicago.edu*

Manuscript received 31 Mar. 2015; accepted 1 Aug. 2015; date of publication xx Aug. 2015; date of current version 25 Oct. 2015. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

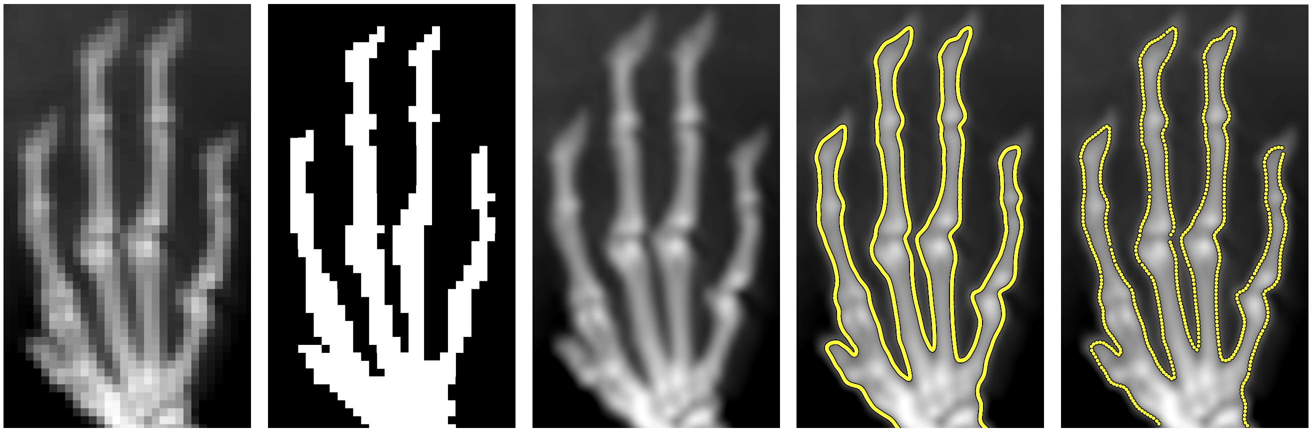

(a) Image data (b) Discrete thresholding (c) Continuous reconstruction (d) Dense isocontour (e) Isocontour particles

Fig. 1. Image data in Diderot are discretely sampled arrays (a), but Diderot is not designed for direct computations on arrays per se, like per-pixel thresholding (b). Rather, Diderot works with continuous fields (c) reconstructed by convolving data arrays with continuous kernels. Computations can involve many independent strands, such as to produce a dense isocontour sampling (d), or strands that communicate, such as uniform sampling of isocontours via interacting particles (e).

An alternative to libraries and toolkits are *domain-specific languages* (DSLs), which trade generality and flexibility for expressiveness and ease of use [32]. There is a long and rich history of DSLs [50], but here we limit our comparison to DSLs that have some commonality with our rationale and application domain.

DSLs can either be implemented as libraries in a host language (*embedded DSLs*) or as standalone programming languages. For example, Delite is a framework for implementing embedded parallel DSLs on heterogeneous processors [15, 10]. Delite uses Scala as its host language and provides sophisticated compilation support for mapping DSL features to efficient code. The distinction between libraries and embedded DSLs is fuzzy; systems like Protovis [8] and $\mathbb{D}^3$ [9] are implemented as libraries, but are sometimes described as DSLs.

DSLs (and especially standalone DSLs) have several advantages over domain-specific libraries. First, their syntax can match the notation used by domain experts and their types and semantics can be designed to model the important domain concepts. Second, their implementations can use analyses and optimizations that take advantage of domain-specific knowledge and restrictions on the language's programming model. Third, they often provide a high-level program model that abstracts away from hardware and operating-system dependencies, providing greater portability. This last point is particularly relevant to *parallel DSLs*, such as Diderot, which can support portable parallel programming across a range of parallel hardware.

There are several recent examples of DSLs targeted at image processing, image analysis, and visualization. Scout is a DSL that extends the data-parallel programming model with *shapes* — regions of voxels in the image data — to accelerate visualization tasks on GPUs [31]. Whereas Diderot is design to compute over continuous tensor fields, Scout is designed for computations over discrete voxels, such as stencil algorithms. Halide is a DSL for pixel-based image processing (e.g. computational photography), with a novel mechanism for separating the definition of what is to be computed from the schedule that specifies how to compute it [39]. Halide demonstrates that high-level DSLs can both make programming easier and deliver high performance, but it addresses a different application domain from Diderot. Vivaldi [17] is a recent DSL that supports parallel volume-rendering applications on heterogeneous systems. It has some support for differential operators, but it does not support the full range of higher-order field operations that Diderot provides. Another recent project is ViSlang [40], which is a system for developing and integrating DSLs for visualization. It provides a concise way to define a programmatic interface to high-performance libraries for scientific visualization. The resulting DSLs can be integrated to form complete applications.

3 LANGUAGE DESIGN

3.1 Elements of Diderot programs

This section uses Fig. 1 to introduce elements of Diderot. Appendix A gives a more detailed summary of the language. Datasets visualized or analyzed in Diderot are discretely sampled arrays of scalar, vector, or tensor data (Fig. 1(a)). The **`image`** type holds such data:

```
image(2)[] I = image("hand.nrrd");
```

`image``(2)` says that the image is sampled over a 2D domain, and tensor *shape* specification `[]` says that the samples are just scalars. 3-vectors have shape `[3]`; 3×3 matrices are `[3,3]`. Diderot programs do not typically operate on the image array itself, or address its individual elements. Diderot is not designed for uniformly data-parallel image computations, such as per-pixel thresholding (Fig. 1(b)).

The visualization and analysis methods in the target domain of Diderot are defined in terms of continuous scalar, vector, or tensor fields. Accordingly, Diderot programs typically start by defining a field as the convolution of discretely sampled data with a continuous reconstruction kernel. For example,

```
field#1(2)[] F = ctmr ⊛ image("hand.nrrd");
```

defines a C^1 continuous ("`#1`") scalar ("`[]`") field `F` over a two-dimensional ("`(2)`") world-space, by convolving ("⊛") the image data in `hand.nrrd` with the Catmull-Rom cubic spline `ctmr`. Fig. 1(c) shows a dense sampling of this field.

Diderot programs are decomposed in terms of *strands*, either autonomous or interacting, which can move throughout fields. Fig. 1(d) shows a dense sampling of an isocontour of `F` found via Newton-Raphson iteration. Isosurface sampling in Fig. 1(d) is computed by:

```
strand isofind (vec2 pos0) {
  output vec2 x = pos0;
  int steps = 0;
  update {
    // Stop after too many steps or leaving field
    if (steps > stepsMax || !inside(x, F))
      die;
    // one Newton-Raphson iteration
    vec2 delta = -normalize(∇F(x)) * F(x)/|∇F(x)|;
    x += delta;
    if (|delta| < epsilon)
      stabilize;
    steps += 1;
  }
}
```

The heart of a Diderot program is the "**`update`**" function, which expresses one iteration of the algorithm. Strands can **`die`** to terminate without saving any output, or **`stabilize`** to save the most recently computed value of the variables tagged with **`output`**. Here, output position `x` (line 2) is saved when the Newton-Raphson iteration (line 9) converges to within `epsilon` (line 12).

For a more complete example of a Diderot program, Fig. 2 lists a particle-based isocontour sampler (Fig. 1(e)), a simple 2-D version of the method of Meyer et al. [33], without curvature-dependent density. At a high level, the program starts with (lines 1-6) the declaration of global inputs, to be set either on the command-line when compiling to a stand-alone executable, or via API when compiling to a library. After creating (line 8) the scalar field in which the isocontour `F(x)=0` will be sampled by interacting particles (according to pairwise energy `phi` and force `phi'`), the strand definition includes local strand state (lines 15-19) and the **`update`** method, which proceeds in two stages.

```
 1  input real radius;       // particle interaction radius
 2  input real epsilon;      // convergence criterion
 3  input int  res;          // initial grid sampling resolution
 4  input real isoval;       // isovalue
 5  input real xmin; input real xmax; // sampling extent
 6  input real ymin; input real ymax;
 7  real limit = radius/7;   // limit on motion
 8  field#1(2)[] F = ctmr ⊛ image("hand.nrrd") - isoval;
 9  // inter-particle energy, and its derivative
10  function real  phi(real x) =
11     (1 - |x|/radius)^4 if (|x| < radius) else 0.0;
12  function real phi'(real x) =
13    -(4/radius)*(1 - |x|/radius)^3 if |x| < radius else 0.0;
14  strand point (vec2 pos0) {
15    output vec2 pos = pos0; // particle position
16    vec2 delta = [0,0];     // change in position
17    bool foundIso = false;  // initial isocontour search done
18    real hh = 1;            // line search step size
19    int iter = 0;           // iteration count
20    update {
21      iter += 1;
22      if (!foundIso) {
23        if (iter > 10 || !inside(x, F))
24          die;
25        // Newton-Raphson
26        delta = -normalize(∇F(pos)) * F(pos)/|∇F(pos)|;
27        pos += delta;
28        if (|delta| < epsilon*radius) {
29          foundIso = true;
30        }
31      } else { // we have foundIso
32        real energy=0;  vec2 force=[0,0];  int ncount = 0;
33        foreach (point P in sphere(radius)) {
34          ncount += 1;
35          vec2 r_ij = pos - P.pos;
36          energy += phi(|r_ij|);
37          force -= normalize(r_ij)*phi'(|r_ij|);
38        }
39        vec2 norm = -normalize(∇F(pos));
40        if (ncount <= 1 && iter % 10 == 0) {
41          vec2 npos = pos + 0.2*radius*[norm[1],-norm[0]];
42          new point(npos);
43        }
44        // project force onto tangent plane
45        force -= norm⊗norm•force;
46        if (|force| > 0) { // take gradient descent step
47          delta = hh*force;
48          if (|delta| > limit) { // enforce speed limit
49            hh *= limit/|delta|;
50            delta = hh*force;
51          }
52          vec2 posLast = pos;
53          pos += delta;
54          // take Newton-Raphson steps back to surface
55          pos -= normalize(∇F(pos)) * F(pos)/|∇F(pos)|;
56          pos -= normalize(∇F(pos)) * F(pos)/|∇F(pos)|;
57          delta = pos - posLast; // record change in this iter
58          real energyNew = 0;
59          foreach (point P in sphere(radius))
60            energyNew += phi(|pos - P.pos|);
61          // Line search with backtracking
62          if (energyNew > energy - 0.3*delta•force) {
63            hh *= 0.5; // backtrack
64            pos = posLast;
65          } else {
66            hh *= 1.1; // bigger step next time
67          }
68        }
69      }
70    }
71  }
72  global{
73    real motion = mean{ |P.delta|/radius | P in point.all };
74    if (motion < epsilon)
75      stabilize;
76  }
77  initially { point([lerp(xmin, xmax, 0, ui, res-1),
78                     lerp(ymin, ymax, 0, vi, res-1)])
79                | vi in 0..(res-1), ui in 0..(res-1) };
```

Fig. 2. Complete Diderot program for particle-based isocontour sampling of scalar field `F` at isovalue `isoval`, producing Fig. 1(e).

First, while not `foundIso`, particles move towards the isocontour, keeping only those that stay in the field domain and converge within 10 iterations. Second, particles query their `radius` neighborhood to determine current energy and force (lines 33-38) due to nearby particles. If the particle has less than two neighbors, occasionally a **`new`** particle will created nearby to fill in the gap. The force from neighbors is projected to the tangent plane (line 45) and the particle position is updated and re-projected onto the isocontour (lines 55, 56). The new location is assessed by computing the new energy (lines 59, 60) as part of a backtracking line search (line 62) [3]: the energy should have gone down by an amount roughly predicted by the force, or else we backtrack and try again on the next iteration (line 64).

In addition to the code executed in parallel by strands, Diderot programs may also have global code that executes between iterations, to monitor the state of computation. In this example, lines 73-75 find average motion of all the strands, and terminate when the particle system has stabilized. The final part of the program is the initialization (line 77) of particles on a coarse grid covering the image domain.

3.2 Design rationale

The design of Diderot is motivated by a desire to speed the development and refinement of new image analysis and visualization algorithms, as evaluated on real-world datasets. Diderot addresses this motivation with two design goals, which are reflected in the two major facets of its design: ease of programming and parallel performance.

The ease of Diderot programming is fostered by notational and structural aspects of the language. The mathematical notation we write on whiteboards should work in programs, so Diderot supports tensor fields as computational values, and uses Unicode to idiomatically express mathematical operations[1]. At a structural level, computation in Diderot is organized into *strands* according to the structure of the algorithm *output*. The output may be an image (in the case of volume rendering), or a set of polylines (for streamlines), or simply a point set (for particle systems). For these examples, the Diderot program states the work required of each strand to produce one rendered pixel, one streamline, or one particle, respectively. This increases the flexibility of Diderot relative to languages that structure parallelism around the *input* data domain (pixels or voxels).

The second goal is to efficiently execute Diderot programs by using modern parallel hardware, such as GPUs. Diderot's execution model attempts to achieve expressiveness while avoiding features that would be difficult to map onto parallel hardware, or that would force programmers to worry about parallel programming challenges like data races, synchronization, and locality. We have therefore based the semantics of Diderot on a simple deterministic parallel programming model, described in Sec. 3.5. This model fits the iterative structure of typical scientific visualization algorithms, while allowing some flexibility in how the model is mapped to different hardware targets.

3.3 Computing with tensors

Diderot strives to mimic the "direct" or coordinate-free notation for vector and tensor expressions commonly used in continuum mechanics [27]. This motivates using `u ⊗ v` for the tensor product, instead of something like "`u v^T`", which depends on recognizing `u` and `v` as N-by-1 column vectors, and which does not generalize to higher-order tensors. Expressing computations in a coordinate-free way tends to clarify mathematical intent. The direct notation encouraged by Diderot also fosters dimensionally general code. For example, to turn Fig. 2 into a program to sample isosurfaces in *three* dimensions, the *only* changes required are replacing **`vec2`** and **`field`**`#1(2)[]` with `vec3` and **`field`**`#1(3)[]`, respectively, and adding a dimension to the strand initialization (line 72).

3.4 Reconstructing and computing with fields

Fields are created by convolving (notated with ⊛) data on regularly sampled grids with continuous reconstruction kernels. Diderot currently knows a variety of kernels that differ in their continuity and

---

[1] Unicode character entry is a possible concern, but the range of practical solutions is expanding: http://en.wikipedia.org/wiki/Unicode_input.

support (how many samples along each axis are needed in convolution). Besides the `ctmr` Catmull-Rom C^1 cubic spline (4-sample support) used in Fig. 1, other possible kernels include C^0 linear interpolation `tent` (2 samples), C^2 cubic B-spline `bspln3` (4 samples), and a C^4 piecewise hexic kernel `c4hexic` (6 samples) that accurately reconstructs cubic polynomials (a 4th-degree error filter or 4EF in the terminology of [36]).

Reconstruction of fields in Diderot is by *separable* convolution; the same kernel is convolved with each image data axis. For example, the convolution $F = V \circledast h$ of image data $V[i,j]$ with s-sample support kernel h is evaluated at image-space location (x_1, x_2) with

$$F(x_1,x_2) = \sum_{i,j} V[i,j]h(x_1 - i)h(x_2 - j); \qquad (1)$$

$$n_i = \lfloor x_i \rfloor \Rightarrow \alpha_i = x_i - n_i \in [0,1) \Rightarrow \qquad (2)$$

$$F(x_1,x_2) = \sum_{i,j=1-s/2}^{s/2} V[n_1+i, n_2+j]h(\alpha_1 - i)h(\alpha_2 - j). \qquad (3)$$

The statement (1) of separable convolution typical in image processing and computer graphics [35] shows how the continuity order of field F is the same as that of reconstruction kernel h, and that partial derivatives $\partial F/\partial x_i$ are reconstructed by convolving with kernel derivatives h' on one axis. Diderot uses (3) to limit the convolution to the kernel support. A field `F` in Diderot is defined over world-space and evaluated with `F(p)` at world-space position vector `p`. The field implementation includes the homogeneous coordinate matrix M, transforming index-space location $\vec{x}$ to world-space $\vec{p}$, to represent the image orientation learned from the image data file. Field evaluation `F(p)` first computes $\vec{x} = M^{-1}\vec{p}$ (the inverse M^{-1} is pre-computed once) before applying (3). Further details of the convolution implementation are in [16].

Diderot supports differential and arithmetic operators on fields in a mathematically idiomatic way. Evaluation of the field derivative `∇F(x)` correctly returns the gradient in world space, according to the covariant transform of the index-space partial derivatives $\partial F/\partial x_i$. The derivative of a field `G` of order-n tensors ($n > 0$) is field `∇⊗G` of order-$(n+1)$ tensors. A compile-time type error results from differentiating a field beyond its continuity order, like the (second derivative) Hessian `∇⊗∇F` of a C^1 scalar field `field#1(3)[] F`.

While thinking mathematically about visualization algorithms on fields, it is natural for programmers to consider expressions like $f = \mathbf{u} \bullet \mathbf{v}$, with the understanding that variables $\mathbf{u}$, $\mathbf{v}$, and f could stand for constant values, or fields, or some mix of both. Diderot syntax supports this convention. As defined in Sec. A.10, arithmetic operators like addition, subtraction, scalar multiplication $*$, and inner product $\bullet$ can work on a mix of tensors and tensor fields. Two fields arising from image data with different orientations and resolutions may be added without restriction. Furthermore, as described in Sec. 5.2, differential operators like gradient ∇, Jacobian $\nabla\otimes$, curl $\nabla\times$, and divergence $\nabla\bullet$ may be applied to such arithmetically constructed fields, and the Diderot compiler determines how to compute their evaluation.

3.5 Bulk-Synchronous Parallel Strand Execution

The *bulk-synchronous parallel* (BSP) programming model of Valient [49, 44] provides the basis of Diderot's execution semantics, illustrated in Fig. 3. All active strands execute in parallel execution steps called *super-steps*, each with two phases: a strand update phase and an optional global computation phase. First, each active strand executes its **`update`** method once, to update its local state, including **`output`** variables. Strands do not directly communicate, but they may learn the state of nearby strands according to their position in world space, as used in the particle system code in Fig. 2. A strand makes a *spatial query* to get an iterable sequence of all strands within some world-space radius, providing (read-only) access to their state at the *beginning* of the super-step. This requires maintaining two copies of strand state, but we can optimize away the extra storage in many cases. Diderot is not intended for applications in which all pairs of strands need to directly communicate, and of the possible mechanisms for limiting strand communication we choose spatial proximity. The assumption is that each strand is responsible for analyzing or visualizing some local region of the data (as with particle systems), so any communication required between strands can likely be expressed in terms of region (or strand) neighborhoods.

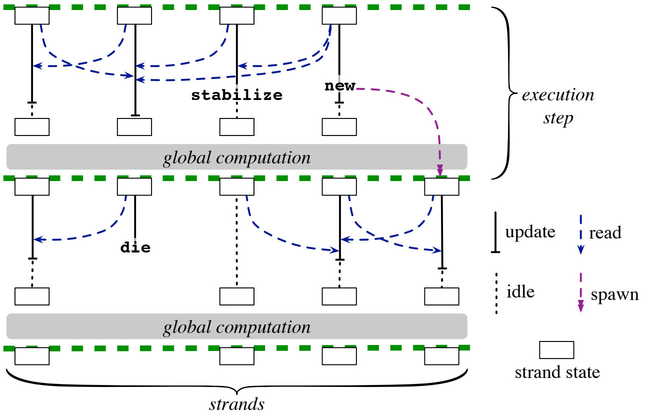


Fig. 3. The Diderot bulk synchronous parallel (BSP) execution model alternates between strand updates and global reductions.

Second, the global phase computes reductions over strands (such as finding average or extremal values of strand variables) to update global variables, or to terminate program execution by stabilizing all strands. At the start of the next super-step, strands created by **`new`** start executing, and all strands can read (but not write) values of globals set in the preceding global phase. Each strand is idle after it finishes executing its **`update`** method until the end of the strand update phase. Stable strands are idle for the entirety of their update. Dead strands are also idle during update, but do not produce any output. The program executes until all strands are either stable or dead. The BSP model is deterministic because the order of strand scheduling does not affect the computation. In practice, non-determinism may arise from the non-associativity of floating-point arithmetic in global reductions, but for debugging purposes this can be avoided at some performance cost.

3.6 Programs as libraries

One drawback of DSLs is their lack of infrastructure for I/O, graphics, networking, or other services required of complete applications. Such functionality requires significant implementation effort and may dilute the domain-specific focus of the language. One solution is to *embed* the DSL in a general-purpose host language, so the embedded DSL can use all the features of the host language. Delite takes this approach, for example [15, 10]. We did not embed Diderot because we did not want to commit to a single host language, and be constrained by its syntax and type system. Supporting idiomatic expression of mathematical operators required Unicode, and field properties like continuity order and tensor shape are not readily encoded in other languages.

To address the infrastructure problem, we compile Diderot programs to *libraries* with a C-language API. This allows programmers to develop applications in *any* language that can call C code. The API includes functions for setting inputs, reading outputs, and controlling program execution. A Diderot program can either be run to completion or for some number of super-steps. Snapshots of program state can be accessed to create animations of program execution. The compiler can additionally generate stand-alone executables (with command-line options to set inputs), which link against the C library created by the compiler. Appendix B demonstrates the API generated from a program for isocontour sampling (Fig. 1(d)).

4 DEMONSTRATIONS AND RESULTS

4.1 Volume rendering

Volume rendering of scalar datasets is a well-studied research area, with many efficient and configurable tools [11, 38, 34, 41]. We show volume renderings here that go well beyond the capabilities of existing tools, by exploiting Diderot's ability to operate on continuous fields as first-class mathematical objects. The quantities volume rendered in these examples (scalar field Canny edges, flow field vortex structure indicators, and tensor invariants) are all defined with a few lines of Diderot code, and the subsequent volume rendering code *remains the same*. Even though the code works from an underlying scalar-, vector-, or tensor-valued volume dataset, the rendering is always of a *scalar* field analytically derived from the data (not pre-computed on a grid) and numerically evaluated at each sample along each ray. The Diderot language permits a clean separation between the code that implements the core visualization algorithm, and the code that defines the quantity of interest to visualize.

```
 1  input real isoval;  input real thick;
 2  input vec3 camEye;  input vec3 camAt;   input vec3 camUp;
 3  input real camFOV;  input int iresU;    input int iresV;
 4  input real refStep; input real rayStep;
 5  input vec3 lightVsp; // light direction in view-space
 6  input real phongKa; input real phongKd;
 7
 8  field#4(3)[] V = bspln5 ⊛ image("ctscan-prefilt.nrrd");
 9  field#4(3)[] F = V - isoval; // isosurface is {x|F(x)=0}
10  function real mask(vec3 x) = 1.0;
11  field#0(1)[3] cmap = tent ⊛ clamp(image("bow.nrrd"));
12  function vec3 color(vec3 x) = cmap(V(x));
13  function real alpha(real v, real g)
14    = clamp(0, 1, 1.3*(1 - |v|/(g*thick)));
15
16  real camDist = |camAt - camEye|;
17  vec3 camN = normalize(camAt - camEye); // away
18  vec3 camU = normalize(camN × camUp);   // right
19  vec3 camV = camU × camN;               // up
20  real camVmax = tan(camFOV*π/360)*camDist;
21  real camUmax = camVmax*iresU/iresV;
22  vec3 light = transpose([camU,camV,camN])•normalize(lightVsp);
23
24  strand raycast (int ui, int vi) {
25    real rayU = lerp(-camUmax, camUmax, -0.5, ui, iresU-0.5);
26    real rayV = lerp(camVmax, -camVmax, -0.5, vi, iresV-0.5);
27    real rayN = camNear;
28    vec3 rayVec = camN + (rayU*camU + rayV*camV)/camDist;
29    real transp = 1;
30    vec3 rgb = [0, 0, 0];
31    output vec4 rgba = [0, 0, 0, 0];
32    update {
33      vec3 x = camEye + rayN*rayVec;
34      if (inside(x,V)) {
35        real val = F(x);
36        vec3 grad = -∇F(x);
37        real a = alpha(val, |grad|)*mask(x);
38        if (a > 0) {
39          a = 1 - pow(1-a, rayStep*|rayVec|/refStep);
40          real depth = lerp(1.1, 0.7, camNear, rayN, camFar);
41          real shade = max(0, normalize(grad)•light);
42          rgb += transp*a*depth*(phongKa + phongKd*shade)*color(x);
43          transp *= 1 - a;
44        }
45      }
46      if (transp < 0.01) {  // early ray termination
47        transp = 0;
48        stabilize;
49      }
50      if (rayN > camFar) stabilize;
51      rayN = rayN + rayStep;
52    }
53    stabilize {
54      real a = 1 - transp; // undo pre-multiplied alpha
55      if (a > 0) rgba = [rgb[0]/a, rgb[1]/a, rgb[2]/a, a];
56    }
57  }
58  initially [raycast(ui, vi) | vi in 0..iresV-1, ui in 0..iresU-1];
```

Fig. 4. Complete Diderot program for volume rendering an approximate isocontour at isovalue `isoval` with thickness `thick` in scalar field `F`.

Fig. 4 shows the complete listing of a Diderot program for ray-cast volume rendering. While the code is mostly self-explanatory, lines 8 to 14 merit further explanation, since they will be changed in subsequent examples. The CT scan volume dataset `ctscan.nrrd` used on line 8, and all subsequent volume datasets used in these volume rendering examples, are *pre-filtered* so that convolution with the 5-order B-spline `bspln5` (piece-wise quintic, C^4 continuous, 6-sample support) interpolates [48]. The scalar field in which to render the zero isocontour is (line 9) `F=V-isoval`. The per-sample color and opacity in the ray-casting are defined in terms of the `cmap` colormap (1-D field of 3-vectors, line 11) and the `alpha` bivariate opacity function (13) based on Levoy [30].

Fig. 5 shows a CT scan of a *Cebus apella* (capuchin) head, for which an image analysis goal involves recovering the bone surface. Diderot can visualize the bone surface by volume rendering. The left side of Fig. 5 shows isocontours rendered with the code in Fig. 4, at three isovalues that span the range of CT values for bone. At too low an isovalue (V=1300) some of the soft tissue is visible, but at a higher isovalue (V=2150) suitable for most of the skull, holes incorrectly appear at the frontal sinuses (above and between the eye sockets). The tooth surface is cleanest at yet another isovalue (V=3000). The colormap is approximately isoluminant.

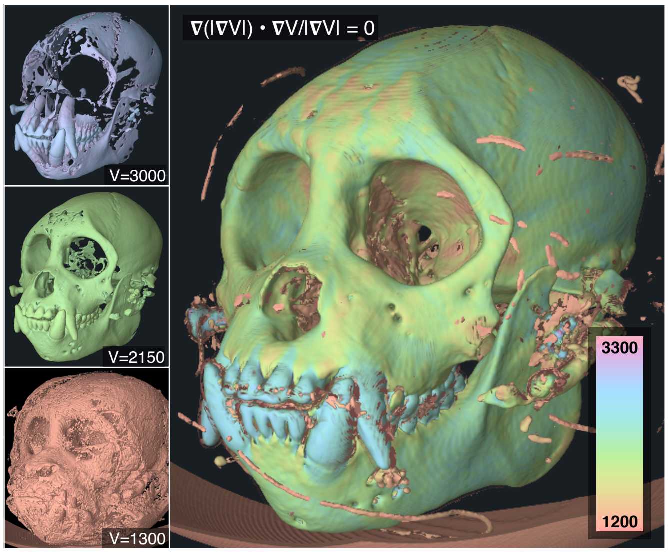


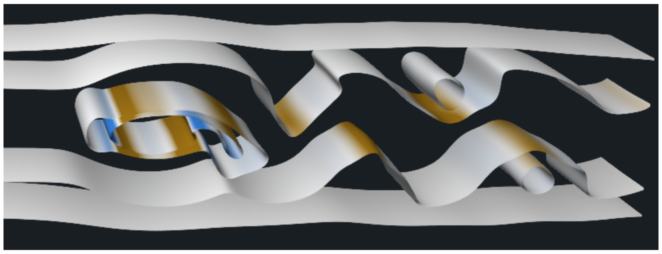
Fig. 5. Volume rendering of isocontours (left) and Canny edges (right) from a CT scan of *Cebus apella* (capuchin) head.

A classical principle of edge detection proposed by Canny is that edge points are where the image gradient magnitude is maximized with respect to motion along the (normalized) gradient direction [14]. Algorithmically, Canny edge detection also involves finding optimal smoothing, and hysteresis thresholding of edge components based on gradient magnitude. These are outside the scope of Diderot, but we can still capture Canny's principle of edge localization. In scalar volume data field $V(x)$, we seek locations where $|\nabla V(x)|$ is maximized with respect to motion along $\nabla V(x)/|\nabla V(x)|$, which means that $\nabla|\nabla V(x)|$ (the gradient of quantity being maximized) is orthogonal to $\nabla V(x)/|\nabla V(x)|$. In Diderot, we write:

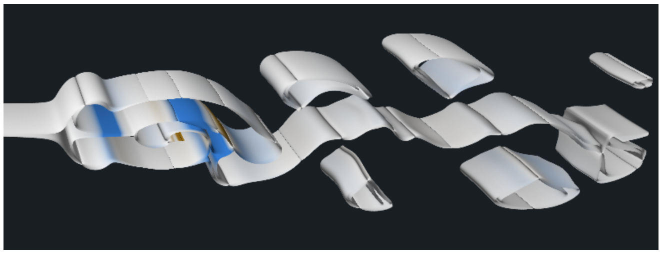
```
 9  field#2(3)[] F = -∇(|∇V|) • ∇V/|∇V|;
10  function real mask(vec3 x) =
        1.0 if (|∇V(x)| > gmin) else 0.0;
```

in place of lines 9 and 10 in Fig. 4. Strong edges (as opposed to minima of gradient magnitude) are selected by user-specified threshold `gmin`. The right of Fig. 5 shows the results, with *no other changes* to the code of Fig. 4. Note that the Canny edges are shaded correctly, based ∇`F`. The isocontour shading also used ∇`F` (line 41), but now ∇`F` is some expression involving *third* derivatives of the volume data `V`, which the Diderot compiler generates automatically, based on its understanding of vector calculus. The smoothness apparent in the bone surface is thanks to the C^4 continuous reconstruction of the data **`field`**`#4(3)[] V` (line 8). The Canny function **`field`**`#2(3)[] F` is C^2 continuous (based the second derivative of `V`), and its gradient ∇`F` will be C^1 continuous. With no prior segmentation or parameter tuning, Fig. 5 correctly shows the skull surface over the frontal sinuses, as well as the very thin bones of the orbital walls (within the eye socket), neither of which are visible with any isovalue reliably higher than all the soft tissue. Diderot has thus greatly simplified the successful visual exploration (by volume rendering) of a first-principles approach (Canny edges) to bone surface extraction in real-world three-dimensional image data.

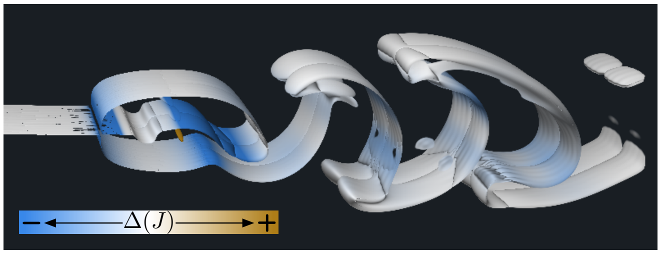
Flow visualization is another domain of scientific visualization that involves mathematically sophisticated consideration of vector fields and their derivatives. For example, one statement of vortex structure



(a) Flow magnitude $|\mathbf{v}| = 0.8$ and $|\mathbf{v}| = 0.4$.

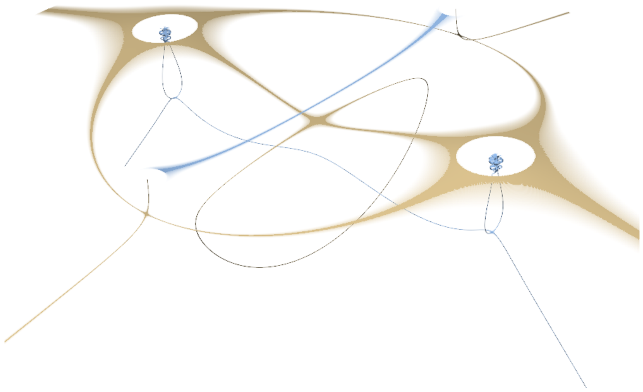


(b) Surfaces around extremum lines: $\frac{\mathbf{v}}{|\mathbf{v}|} \cdot \frac{\nabla|\mathbf{v}|}{|\nabla|\mathbf{v}||} = \pm 0.99$.

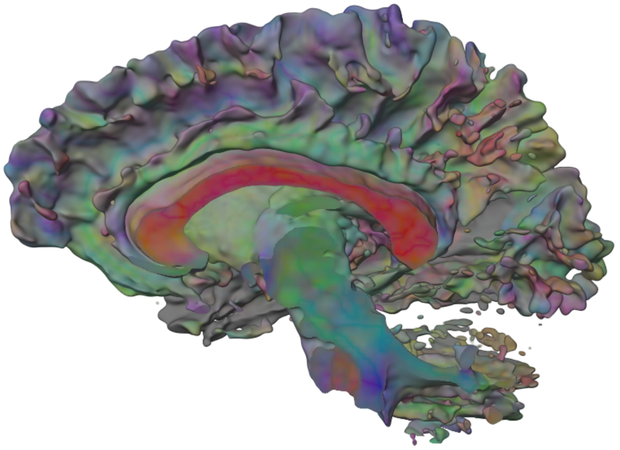


(c) Normalized helicity $\frac{\mathbf{v}}{|\mathbf{v}|} \cdot \frac{\nabla\times\mathbf{v}}{|\nabla\times\mathbf{v}|} = \pm 0.99$.

Fig. 6. Volume rendering of isocontours of vector field attributes, colored by the discriminant of the Jacobian $\Delta(J)$.



(a) Lines of degeneracy in a stress tensor field revealed by volume rendering isosurfaces of tensor mode +1 (yellow) and -1 (blue).



(b) Volume rendering a diffusion tensor field of the human brain at FA=0.3.

Fig. 7. Volume rendering stress tensor (a) and diffusion tensor (b) fields.

identifies them with locations where the direction of flow $\frac{\mathbf{v}}{|\mathbf{v}|}$ is aligned with that of its curl $\frac{\nabla\times\mathbf{v}}{|\nabla\times\mathbf{v}|}$ [20, 37]. This is equivalent to saying that the normalized helicity $\frac{\mathbf{v}}{|\mathbf{v}|} \cdot \frac{\nabla\times\mathbf{v}}{|\nabla\times\mathbf{v}|}$ is at an extremum (near +1 or -1). We show here how volume rendering in Diderot can directly visualize the mathematical elements of this type of flow field analysis.

The renderings in Fig. 6 use a single time-step from a Navier-Stokes simulation of flow (from left to right) past a square rod, creating a train of vortices [13, 51]. The Fig. 4 volume rendering code is re-used, but with `V` defining a 3-D vector field:

```
8   field#4(3)[3] V = bspln5 ⊛ image("flow.nrrd");
```

and with a diverging colormap of the Jacobian discriminant (in lieu of lines 11 and 12):

```
field#3(3)[3,3] J = ∇×V;
field#3(3)[] A = -trace(J);
field#3(3)[] B = (trace(J)*trace(J) - trace(J•J))/2;
field#3(3)[] C = -det(J);
field#3(3)[] Q = (A*A/3.0 - B)/3.0;
field#3(3)[] R = (-2.0*A*A*A/27.0 + A*B/3.0 - C)/2.0;
field#3(3)[] D = R*R - Q*Q*Q; // the discriminant
field#0(1)[3] dmap = clamp(tent⊛image("diverg.nrrd"));
function vec3 color(vec3 x) = dmap(D(x));
```

After forming a field of the Jacobian $\nabla\times\mathbf{v}$, fields `A`, `B`, and `C` are the coefficients of the characteristic polynomial $p(\lambda) = \lambda^3 + A\lambda^2 + B\lambda + C$, and the discriminant field `D` is the pre-cursor to analytically solving $p(\lambda) = 0$ for λ. There are two complex-conjugate eigenvalues (indicating rotational flow) when the discriminant `D(x)` is negative [37].

The only substantial difference in how the various renderings in Fig. 6 were computed was in the statement of the derived scalar field `F` being rendered. Fig. 6(a) shows simple isosurfaces of flow magnitude, created by using

```
9   field#4(3)[] F=|V|;
```

Fig. 6(b) visualizes isosurfaces (at ±0.99) of a quantity related to the extremal lines of the field: places where the vector magnitude is extremal with respect to motion perpendicular to the vector, or, where $\nabla|\mathbf{v}|$ is aligned with $\mathbf{v}$ [46, 37]:

```
9   field#3(3)[] F = (V/|V|) • (∇|V|/|(∇|V|)|);
```

Fig. 6(c) visualizes isosurfaces (at ±0.99) of normalized helicity, using

```
9   field#3(3)[] F = (V/|V|) • (∇×V/|∇×V|);
```

Note that in all cases, the Diderot compiler is determining how to analytically differentiate field `F` so that it can be shaded correctly. These renderings use two-sided lighting:

```
41        real shade = |normalize(grad)•light|;
```

It should be emphasized that although a scalar field is being volume rendered to make these images, *there is no pre-computed scalar dataset*: the scalar field is defined symbolically in terms of a vector field (reconstructed by convolution), and the Diderot compiler generates the necessary instructions to probe the scalar field at each ray sample position during rendering.

Two examples of volume rendering tensor fields further demonstrate the simplicity of creating mathematically and computationally sophisticated visualization tools with few lines of Diderot code. Research in tensor field topology often investigates lines of degeneracy: locations in a tensor field where two of the tensor eigenvalues are equal, or equivalently, places where the tensor *mode* is ±1 [19, 47].

The mode of tensor $\mathbf{D}$ is defined as

$$\mathrm{mode}(\mathbf{D}) = 3\sqrt{6}\,\det(\widetilde{\mathbf{D}}/|\widetilde{\mathbf{D}}|) \tag{4}$$

$$\widetilde{\mathbf{D}} = \mathbf{D} - \mathrm{tr}(\mathbf{D})\mathbf{I}/3 \tag{5}$$

$$|\widetilde{\mathbf{D}}| = \sqrt{\mathrm{tr}(\widetilde{\mathbf{D}}\widetilde{\mathbf{D}}^T)}. \tag{6}$$

This can be directly expressed in Diderot, starting with a stress tensor dataset (simulating a double point load), then defining fields of tensor deviatoric `E`$=\widetilde{\mathbf{D}}$ and mode `F`:

```
field#4(3)[3,3] V = bspln5 ⊛ image("stress.nrrd");
field#4(3)[3,3] E = V - trace(V)*identity[3]/3;
field#4(3)[] F = 3*sqrt(6)*det(E/|E|);
```

Fig. 7(a) shows `F` rendered with the same code as in Fig. 6, with yellow for mode $= +1$ and blue for mode $= -1$. The rendering shows the lines of degeneracy (the ridge and valley lines of tensor mode) [47].

Finally, Fig. 7(b) shows a volume rendering of half of a human brain diffusion tensor scan, showing an isosurface of *fractional anisotropy* (FA), which quantifies the amount of directional organization in the white matter as captured by the single tensor model of diffusion. Though FA is often defined in terms of eigenvalues, its original definition [5] involved just the tensor $\mathbf{D}$ and its deviatoric $\widetilde{\mathbf{D}}$ (5)

$$\mathrm{FA} = \sqrt{\frac{3}{2}}\frac{|\widetilde{\mathbf{D}}|}{|\mathbf{D}|}. \tag{7}$$

Diderot permits directly translating this definition into working code:

```
field#4(3)[3,3] V = bspln5 ⊛ image("dti.nrrd");
field#4(3)[3,3] E = V - trace(V)*identity[3]/3;
field#4(3)[] F = sqrt(3.0/2.0)*|E|/|V| - isoval;
```

The RGB coloring is provided by the traditional map of the principal eigenvector, modulated by linear anisotropy [52]. In Diderot:

```
function vec3 color(vec3 x) {
   real{3} ev = evals(V(x));
   vec3 dir = evecs(V(x)){0};
   real CL = (ev{0} - ev{1})/ev{0};
   return [|dir[0]|,|dir[1]|,|dir[2]|]*CL;
}
```

Note also that in both tensor invariant (mode or FA) volume renderings, the Diderot compiler did the "heavy lifting" of analytically deriving and numerically computing the spatial derivatives of the tensor invariants, expanded in terms of the spatial derivatives of the individual tensor components. Manually deriving and implementing these expressions is tedious and error-prone [47].

4.2 LIC and Streamlines

Fig. 8 lists a Diderot program for computing a line-integral convolution (LIC) visualization of a two-dimensional vector field, seen as the background image of Fig. 10. The program is a straight-forward implementation of the original LIC method [12], in which samples of a noise texture (field `R`) are averaged along a streamline through normalized vector field `nV` (line 6). The integration continues for `stepNum` steps or until the streamline leaves the domain. Field evaluations falling outside the domain are gracefully handled by the border functions `clamp` (line 5) and `wrap` (line 6). Upon stabilization, the output color value is computed by modulating the LIC contrast by the velocity at the start point `x0` (line 23) and colormapping by vorticity `∇×V(x0)` (line 24). Over-all image contrast is clamped to encompass roughly two standard deviations (line 25) in the convolution result, where `stdv` is computed (line 5) as the expected standard deviation of the average of `stepNum` samples in the unit standard deviation noise field. Two strands are used for each output pixel; one upstream (line 32 `si=0` and line 9 `h=-h0`) and one downstream (`si=1` and `h=h0`).

To emphasize (along with the particle system in Fig. 1(e)) that Diderot programs can generate geometry as well as values on grids, the program in Fig. 9 computes the geometry of the streamlines overlayed on the LIC result in Fig. 10. For each seedpoint `x0` in text file `seeds.txt` (line 1), `sline(x0)` outputs a sequence of **`vec2`** streamline vertex positions, starting with `x0` (line 9). As they are computed by midpoint method integration (line 12), points `x` along the streamline are appended to the output path (line 13). A small arrowhead, proportional to `V(x)` (line 18), is added in the **`stabilize`** method to the downstream end of the streamline polyline; this is converted upon rendering (not shown) to a filled triangle. In the program output, all polyline vertices are concatenated into a single array, and a second array stores the start indices for each streamline. The polylines were post-processed to produce the vector graphics seen in Fig. 10.

```
1  input int sizeX;  input int sizeY; // LIC image size
2  input real h0;       // step size of integration
3  input int stepNum;  // steps taken up or downstream
4  real stdv = sqrt(1.0/stepNum);
5  field#1(2)[2] V = bspln3 ⊛ clamp(image("flow.nrrd"));
6  field#1(2)[2] nV = normalize(V);
7  field#0(2)[] R = tent ⊛ wrap(image("rand.nrrd"));
8  field#0(1)[3] cmap = tent ⊛ clamp(image("cmap.nrrd"));
9  strand LIC (vec2 x0, real sign) {
10   real h = sign*h0;
11   vec2 x = x0;
12   int step = 0;
13   real sum = R(x0)/2;  // initialize convolution sum
14   output vec3 rgb = [0,0,0];
15   update {
16     x += h*nV(x + 0.5*h*nV(x)); // Midpoint method
17     if (step == stepNum || !inside(x, V))
18       stabilize;
19     sum += R(x);
20     step += 1;
21   }
22   stabilize {
23     sum *= sqrt(|V(x0)|)/stepNum;
24     rgb = cmap(∇×V(x0)) // colormap of vorticity
25            * clamp(0,1, lerp(0,1, -stdv, sum, stdv));
26   }
27 }
28 initially [ LIC([lerp(0, 6.78, -0.5, xi, sizeX-0.5),
29                  lerp(0, 3.72, -0.5, yi, sizeY-0.5)],
30                  lerp(-1, 1, 0, si, 1))
31             | yi in 0..(sizeY-1), xi in 0..(sizeX-1),
32               si in 0..1 ];
```

Fig. 8. Complete Diderot program for LIC, creating the background raster image in Fig. 10.

5 IMPLEMENTATION

5.1 Strand communication and global reduction

Strands may communicate with neighboring strands according to their proximity in world space. We accelerate the spatial queries with a k-d tree [6, 24], a binary tree that at each internal node splits world space along one axis. The split axis alternates (for two dimensions) or cycles (three dimensions) through the axes as one descends the tree. Strands are assigned to nodes in the tree by comparing their coordinate on the splitting axis to the splitting value.

Spatial queries pose an implementation challenge for parallel targets. To account for strand motion, and the creation of new strands, the tree must be maintained at the end of each super-step, prior to executing the next strand updates. Maintaining the spatial data structure must be parallelized to avoid sequential bottlenecks. We use a parallel version of the median-of-medians algorithm [7] to select the splitting value for the world-space coordinate at each level of the tree. The GPU presents additional challenges for the dynamic memory management needed to support strand motion and creation. OpenCL requires that GPU memory be preallocated on the host side before running the GPU computation, but an execution step might exceed the allocation. For example, if the host side preallocated memory for 30 strands, used to store the states of 20 active or stable strands, then the current execution step cannot successfully create more then 10 new strands. Fortunately, Diderot's execution model preserves the starting state of strands in a given execution step, so we can abort the step if we run out of memory and restart it with a larger preallocation.

```
vec2{} x0s = load("seeds.txt"); // list of seedpoints
real h = 0.02;
int stepNum = 200;
field#1(2)[2] V = bspln3 ⊛ image("flow.nrrd");
real arrow = 0.1;    // scale from |V(x)| to arrow size
strand sline(vec2 x0) {
  int step = 0;
  vec2 x = x0;
  output vec2{} p = {x0}; // start streamline at seed
  update {
    if (inside(x, V)) {
      x += h*V(x + 0.5*h*V(x));  // Midpoint method
      p = p @ x;     // append new point to streamline
    }
    step += 1;
    if (step == stepNum) {
      // finish streamline with triangular arrow head
      vec2 a = arrow*V(x);     // length of arrow head
      vec2 b = 0.4*[-a[1],a[0]]; // perpendicular to a
      p = p@(x-b); p = p@(x+a); p = p@(x+b); p = p@x;
      stabilize;
    }
  }
}
initially [ sline(i, x0s{i}) | i in 0..length(x0s)-1 ];
```

Fig. 9. Diderot code for streamlines (with arrowheads) in a vector field. This computed the streamlines drawn over the LIC result in Fig. 10.

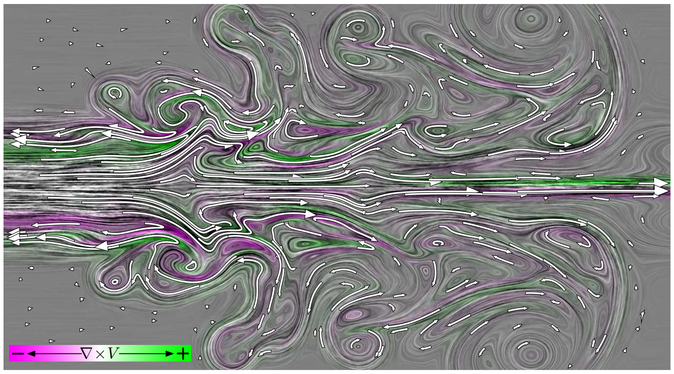


Fig. 10. Two-dimensional turbulent flow visualization computed in Diderot, with a LIC image (Fig. 8) underneath streamlines (Fig. 9). In the LIC image, contrast indicates velocity, and color indicates vorticity.

Strands may indirectly share information via global reductions (the second part of the super-step), which can modify the global variables that strands may read in their subsequent update. We accelerate the reductions by regrouping them to avoid needlessly repeating work. For example, if a mean and product reduction can be performed together then the Diderot compiler will group them into the same execution phase rather than executing them individually. The regrouping is straightforward to implement in the sequential version; the main challenge is fusing reduction phases to reduce overhead. For the parallel version, we integrate the reductions into the barrier synchronization that ends the super-step.

5.2 EIN Intermediate Representation

Relative to [16], Diderot has a new intermediate representation (IR), called *EIN*. EIN is inspired by Einstein index notation or the summation convention, a concise notation for tensor calculus [23]. Others have previously extended Einstein notation, based on close study of its ambiguities and limitations [26, 1, 4, 18, 43, 45, 21]. Part of the ambiguity is related to implicit summation. EIN uses explicit summation, and adds representation of operations such as convolution, image indexing, kernel differentiation, and trigonometry.

The EIN IR is embedded Diderot's static single assignment (SSA) representation, with EIN assignment nodes of the form

$$t = \lambda params\langle e\rangle_{\alpha}(args)$$

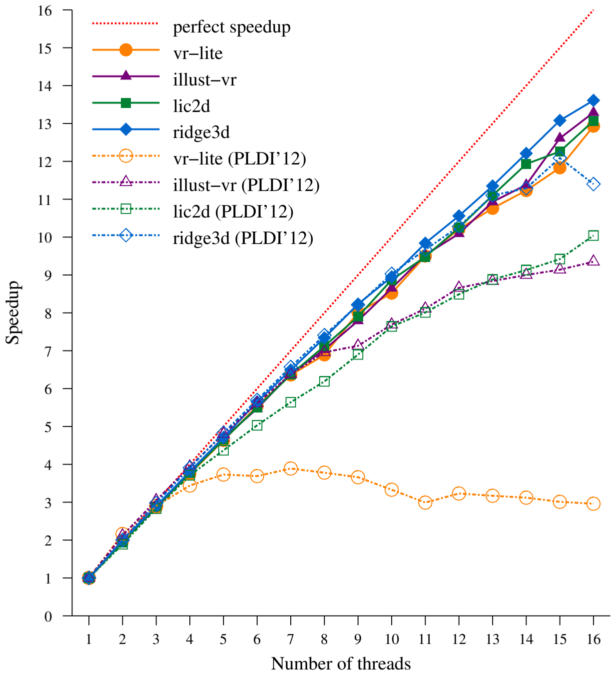


Fig. 11. A comparison of parallel speedup on the benchmark programs from [16] on a 16-core Intel Xeon E5-2687W.

where t is the variable being assigned, $\lambda params\langle e\rangle_{\alpha}$ is an EIN operator with formal parameters *params* and body $\langle e\rangle_{\alpha}$, and *args* are the arguments to the EIN operator. Once a surface language operation is mapped to an EIN operator, the compiler can handle the computations generically, by systematically applying EIN operators to one another, normalizing, and optimizing. The compiler can then break the EIN operator apart into simple and direct scalar and vector operations to generate code.

A family of operations (like tensor plus tensor, field plus tensor, field plus field) that previously [16] required several type-specific operations and case analysis can be captured with a single EIN operator. All tensor operators are lifted to tensor fields as consequence of their expression in EIN. The previous compiler generated code for a fixed set of tensor types, and a fixed set of operations between them, which impeded adding new operators to Diderot. EIN represents tensors and tensor fields of arbitrary shape. The new compiler simplifies complicated and large EIN operators into operations on scalars and vectors. It needs only to generate code for scalar and vector operations, avoiding the explosion of shape-specific operators in the compiler.

The compiler uses a rewrite system to optimize and lower the EIN representation. The Diderot programmer can then define fields with expressions like $\nabla(F \bullet G)$ without worrying about how to expand and simplify this in terms of derivatives of F and G, and their individual scalar components. Rewrite rules include derivative identities such as $\nabla(F+G) \Longrightarrow (\nabla F + \nabla G)$, the product rule $\nabla(f*g) \Longrightarrow g\nabla f + f\nabla g$, the quotient rule, and the chain rule. EIN also includes the permutation tensor $\mathcal{E}_{ijk}$ and the Kronecker delta δ_{ij}. The compiler understands that $\mathcal{E}_{ijk}\partial_{jk} \Longrightarrow 0$, where ∂_{jk} indicates partial derivatives along axes j and k, which produces well-known identities like $\nabla \times \nabla\varphi \Longrightarrow 0$ and $\nabla \bullet (\nabla \times F) \Longrightarrow 0$. The compiler also understands that $\mathcal{E}_{ijk}\mathcal{E}_{ilm} \Longrightarrow \delta_{jl}\delta_{km} - \delta_{jm}\delta_{lk}$ and $\delta_{ij}T_j \Longrightarrow T_i$, which produces identities such as $(a \times b) \times c \Longrightarrow b(a \bullet c) - a(b \bullet c)$. Being able to automatically find identities as part of optimization provides the benefits of domain-specific optimizations without having to encode a lot of domain-specific knowledge in the compiler.

6 PERFORMANCE EVALUATION

While space does not permit an in-depth performance analysis, we revisit previously published benchmarks [16] to demonstrate significant performance improvements provided by the new compiler. These

Program	Teem	Diderot (PLDI '12)					Diderot (this paper)					
		Seq.	1P	6P	12P	16P	Seq.	1P	6P	12P	16P	OpenCL
vr-lite	19.93	8.63	9.51	2.57	2.94	3.20	7.46	7.52	1.36	0.74	0.59	1.43
illust-vr	86.16	44.30	48.55	8.65	5.61	5.19	38.12	38.28	7.00	3.79	2.88	4.32
lic2d	3.03	1.59	1.64	0.33	0.19	0.16	1.56	1.51	0.28	0.15	0.12	1.09
ridge3d	7.92	5.96	6.36	1.12	0.62	0.56	5.22	5.26	0.93	0.50	0.39	1.77

Table 1. Average performance results over 10 runs (times in seconds).

results address our second design goal (Sec. 3.2) of combining good parallel performance with our high-level programming model.

Our test machine is a dual Intel Xeon E5-2687W system (16 cores) running Ubuntu 12.04 Linux. The OpenCL measurements were taken on the same system using a NVIDIA Tesla K20c with NVIDIA's CUDA 6.0 driver. All code was compiled with optimization level `-O3`. For each benchmark, we report the average wall-clock time for the computational part from 10 runs on a lightly-loaded machine.

The benchmarks presented here are from our earlier work, and were originally chosen to represent typical workloads:

- **textbf**: a simple volume renderer with Phong shading, like Fig. 4.
- **illust-vr**: A more complex volume renderer with curvature-based illustrative rendering [29], including tensor calculations that are awkward to express in other languages.
- **lic2d**: Line integral convolution visualization of a synthetic 2D vector field [12], like in Fig. 8 but simpler.
- **ridge3d**: An initial uniform distribution of points within a portion of CT scan of a lung is moved iteratively towards the centers of blood vessels, using Newton optimization to compute ridge lines [22], requiring eigenvalues and eigenvectors of the Hessian.

The most striking improvement of our current system over our previously reported work is in the performance scaling on SMP hardware. Fig. 11 shows the speedup curves for our benchmarks using both the current version of Diderot (solid lines) and the prevoius version (dashed lines). We now get excellent speedup for all four benchmarks at 16 cores, whereas the previous version scales poorly and does significantly worse on three benchmarks. Furthermore, the baseline performance of the previous version is worse ,often significantly, so we are seeing better scaling even when compared to a better baseline.

Tbl. 1 presents selected numeric results from our experiments. For each benchmark, we report several sets of numbers: the execution time for the Teem version (hand-coded C), sequential and parallel (1, 6, 12, and 16-core) execution times for the PLDI'12 version of the system, sequential and parallel (1, 6, 12, and 16-core) execution times for the current version of the system, and execution times for the OpenCL version of the program. The sequential performance of our compiler has improved — 15% faster for three of the four benchmarks.

The performance of the OpenCL target is benchmark dependent. For a program like illust-vr, which has significant arithmetic intensity, the GPU is comparable to 9-core performance. On the other hand, it performs much worse on lic2d, which is more memory-bound. Appendix C compares Diderot with hand-written OpenCL code for two kinds of volume rendering. These limited results suggest that nontrivial programs are more easily expressed in Diderot than in OpenCL, with a performance penalty that is acceptable when the coding skill and time of researchers is the actual bottleneck. Nonetheless, there is room for significant improvement in our OpenCL implementation.

7 DISCUSSION AND ONGOING WORK

This paper presents a significant step towards creating a powerful and portable parallel DSL for scientific visualization and image analysis. In particular, supporting fields as a fundamental abstraction in Diderot frees programmers from worrying about the computational details of convolving on the underlying sampling grid and the mathematical details of differentiating arithmetically constructed fields.

The utility of Diderot would be enhanced by further work on its mathematical basis. We hope to expand the kinds of fields it can handle. Sec. 4.1, for example, used the same volume rendering code to visualize a variety of fields (all based on convolution on regular grids). The same rendering code could in principle be re-used to visualize fields arising from higher-order basis functions on unstructured meshes, point clouds with radial basis function, or analytic closed-form expressions. More ambitious would be a way of defining the `update` method not with imperative program statements, but with a declarative statement of mathematical intent. The volume rendering code of Fig. 4 would be simplified by stating its goal as solving a volume rendering integral, if the sampling and numerical integration schemes could be automatically generated by the compiler. Diderot cannot currently warn the user, for example, when too large a ray sampling distance will cause undersampling artifacts.

Diderot will also benefit from work in less mathematical areas. Most important are improvements to our implementation on GPUs. These include developing a better scheduler for strands on the GPU; our current scheduler does well with irregular workloads, but its baseline performance is not very good. We are also looking to implement a virtual memory strategy akin to that already developed for GPU-based rendering [25]; such a scheme will allow us to scale to data sets that are larger than the available memory on GPUs. Further work on code generation may lead to other GPU performance improvements. Our compiler currently generates OpenCL code, but generating CUDA code should allow better performance on NVIDIA hardware. Targeting OpenCL or CUDA, instead of assembly code or low-level GPU code (e.g., SPIR or PTX), greatly simplifies our Diderot compiler implementation, while benefiting from the optimizing compilers specialized for the target hardware. Finally, automatically generating GUIs for the input and output variables of a Diderot program would accelerate the exploration of the parameter space of different algorithms.

The source code for the Diderot compiler is available[2] and a release is upcoming, which will merge features currently split across different branches. Scripts for regenerating representative images from freely available datasets will be made available from our web page.

ACKNOWLEDGMENTS

We gratefully acknowledge the anonymous reviewers for their constructive comments. We also thank the providers of data seen in the figures. Fig. 1: University of Utah SCI group, NIH NIGMS grant P41GM103545. Fig. 5: Callum Ross, University of Chicago. Fig. 6: Resampling by Tino Weinkauf of Navier-Stokes simulation by S. Camarri, M.-V. Salvetti, M. Buffoni, and A. Iollo [28]. Fig. 7(a): Xavier Tricoche, Purdue University. Fig. 7(b): Centre for Functional MRI of the Brain, John Radcliffe Hospital, Oxford University. Fig. 10: Wolfgang Kollmann, UC Davis. Portions of this research were supported by National Science Foundation award CCF-1446412. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

REFERENCES

[1] K. Ahlander. Einstein summation for multi-dimensional arrays. *Computers and Mathematics with Applications*, 44:1007–1017, Oct.–Nov. 2002.

[2] J. Ahrens, B. Geveci, and C. Law. Paraview: An end-user tool for large data visualization. In *Visualization Handbook*, pages 717–731. Academic Press, Inc., Orlando, FL, USA, 2005.

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

---

[2] http://diderot-language.cs.uchicago.edu

[4] A. H. Barr. The Einstein summation notation: Introduction and extensions. In *ACM SIGGRAPH 1989 Course Notes #30: Topics in Physically Based Modeling*, pages J1–J12, 1989.

[5] P. J. Basser and C. Pierpaoli. Microstructural and physiological features of tissues elucidated by quantitative-diffusion-tensor MRI. *J. Mag. Res., B*, 111:209–219, 1996.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[7] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comp. Syst. Sci.*, 7(4):448–461, Aug. 1973.

[8] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Vis. Comp. Graph. (Proc. InfoVis)*, 15(6):1121–1128, 2009.

[9] M. Bostock, V. Ogievetsky, and J. Heer. $\mathbb{D}^3$: Data-driven documents. *IEEE Trans. Vis. Comp. Graph. (Proc. InfoVis)*, 17(12):2301–2309, 2011.

[10] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT '11*, Oct. 2011.

[11] S. Bruckner and E. Gröller. Volumeshop: An interactive system for direct volume illustration. In *Proc. IEEE Visualization*, pages 671–678, Oct. 2005.

[12] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *Proc. SIGGRAPH 93*, pages 263–270, Aug. 1993.

[13] S. Camarri, M.-V. Salvetti, M. Buffoni, and A. Iollo. Simulation of the three-dimensional flow around a square cylinder between parallel walls at moderate Reynolds numbers. In *XVII Congresso di Meccanica Teorica ed Applicata*, 2005.

[14] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–714, 1986.

[15] H. Chafi, Z. Devito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *OOPSLA '10*, pages 835–847, Oct. 2010. Part of the Onward! 2010 Conference.

[16] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *PLDI '12*, pages 111–120. ACM, June 2012.

[17] 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 Trans. Vis. Comp. Graph. (Proc. SciVis)*, 20(12):2407–2416, Dec. 2014.

[18] T. L. Chow. *Mathematical Methods for Physicists : A Concise Introduction*. Cambridge University Press, Cambridge, 2000.

[19] J. C. Criscione, J. D. Humphrey, A. S. Douglas, and W. C. Hunter. An invariant basis for natural strain which yields orthogonal stress response terms in isotropic hyperelasticity. *J. Mech. Phys. Sol.*, 48:2445–2465, 2000.

[20] D. Degani, Y. Levy, and A. Seginer. Graphical visualization of vortical flows by means of helicity. *AIAA Journal*, 28:1347–1352, Aug. 1990.

[21] K. Dullemond and K. Peeters. *Introduction to Tensor Calculus*. Kees Dullemond and Kasper Peeters, 1991.

[22] D. Eberly. *Ridges in Image and Data Analysis*. Kluwer Academic Publishers, Boston, MA, 1996.

[23] A. Einstein. The foundation of the general theory of relativity. In A. J. Kox, M. J. Klein, and R. Schulmann, editors, *The Collected papers of Albert Einstein*, volume 6, pages 146–200. Princeton University Press, Princeton NJ, 1996.

[24] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, Sept. 1977.

[25] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Trans. Vis. Comp. Graph. (Proc. SciVis)*, 18(12):2285–2294, 2012.

[26] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. E. Bernholdt, S. Hirata, C.-C. Lam, R. M. Pitzer, J.Ramanuja, and P.Sadayappan. Automated operation minimization of tensor contraction expressions in electronic structure calculations. *Proc. ICCS 2005 5th International Conference on Computational Science*, pages 155–164, 2005.

[27] G. A. Holzapfel. *Nonlinear Solid Mechanics*. John Wiley and Sons, Ltd, England, 2000.

[28] International CFD database. `http://cfd.cineca.it/`. Further info at `https://people.mpi-inf.mpg.de/~weinkauf/notes/squarecylinder.html`.

[29] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proc. IEEE Visualization*, pages 67–74, Oct. 2003.

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

[31] 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.

[32] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[33] M. D. Meyer, P. Georgel, and R. T. Whitaker. Robust particle systems for curvature dependent sampling of implicit surfaces. In *SMI '05*, pages 124–133, June 2005.

[34] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs. Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Comp. Graph. App.*, 29(6):6–13, 2009.

[35] D. P. Mitchell and A. N. Netravali. Reconstruction filters in computer graphics. *Computer Graphics (SIGGRAPH)*, 22(4):221–228, Aug. 1988.

[36] T. Möller, R. Machiraju, K. Mueller, and R. Yagel. Evaluation and design of filters using a Taylor series expansion. *IEEE Trans. Vis. Comp. Graph.*, 3(2):184–199, 1997.

[37] R. Peikert and M. Roth. The "parallel vectors" operator - a vector field visualization primitive. In *Proc. IEEE Visualization '99*, pages 263–270, 1999.

[38] J. Plate, T. Holtkaemper, and B. Froehlich. A flexible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Trans. Vis. Comp. Graph. (Proc. Vis.)*, 13(6):1584–1591, Nov. 2007.

[39] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(4):32:1–32:12, July 2012.

[40] P. Rautek, S. Bruckner, M. E. Gröller, and M. Hadwiger. ViSlang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Trans. Vis. Comp. Graph. (Proc. SciVis)*, 20(12):2388–2396, Dec. 2014.

[41] C. Rieder, S. Palmer, F. Link, and H. K. Hahn. A shader framework for rapid prototyping of GPU-based volume rendering. *Computer Graphics Forum (Proc. EuroVis'11)*, 30(3):1031–1040, June 2011.

[42] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Kitware, Inc., Clifton Park, New York, 3rd edition, 2004.

[43] J. Simmonds. *A Brief on Tensor Analysis*. Springer-Verlag, New York, 1982.

[44] D. Skillicorn, J. M. Hill, and W. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[45] I. S. Sokolinkoff. *Tensor Analysis*. John Wiley and Sons, New York, 1960.

[46] C.-K. Tang and G. Medioni. Extremal feature extraction from 3-D vector and noisy scalar fields. In *Proc. Visualization '98*, pages 95–102, Oct. 1998.

[47] X. Tricoche, G. Kindlmann, and C.-F. Westin. Invariant crease lines for topological and structural analysis of tensor fields. *IEEE Trans. Vis. Comp. Graph. (Proc. Vis.)*, 14(6):1627–1634, Nov.–Dec. 2008.

[48] M. Unser, A. Aldroubi, and M. Eden. B-Spline signal processing: Part II–Efficient design and applications. *IEEE Trans. Sig. Proc.*, 41(2):834–848, Feb. 1993.

[49] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[50] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[51] W. von Funck, T. Weinkauf, H. Theisel, and H.-P. Seidel. Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE Trans. Vis. Comp. Graph. (Proc. Vis.)*, 14(6):1396–1403, Nov.–Dec. 2008.

[52] C.-F. Westin, S. E. Maier, B. Khidhir, P. Everett, F. A. Jolesz, and R. Kikinis. Image processing for diffusion tensor magnetic resonance imaging. In *Proc. MICCAI*, volume 1679 of *LNCS*, pages 441–452, 1999.

[53] T. S. Yoo, editor. *Insight into Images: Principles and Practice for Segmentation, Registration, and Image Analysis*. A K Peters/CRC Press, 2004.

A THE DIDEROT LANGUAGE

This appendix provides a concise, but complete tour of the Diderot language.

A.1 Types

Diderot is a strongly-typed language with a novel, dependent-type system that statically tracks important mathematical properties of the program. The following table summarizes the types of the language:

τ	::=	**`bool`**	booleans
		`int`	integers
		`string`	strings
		`tensor``[`σ`]`	tensor with shape σ
		S	reference to a strand S
		τ`{`d`}`	known-size sequence ($1 \leq d$)
		τ`{}`	unknown/dynamic length sequence
		`image``(`d`)[`σ`]`	dimension-d image ($1 \leq d \leq 3$)
		`kernel``#`k	reconstruction kernel of continuity k ($0 \leq k$)
		`field``#`k`(`d`)[`σ`]`	tensor field with continuity k, dimension d, and shape σ
σ	::=	$d_1, \ldots, d_n$	tensor shape ($0 \leq d$)

Diderot also defines synonyms for common tensor shapes, such as **`real`** for **`tensor`**`[]` and **`vec3`** for **`tensor`**`[3]`.

A.2 Top-level definitions

A Diderot program consists of a sequence of top-level definitions. At a minimum, the top-level definitions include a strand definition and a global **`initially`** block, but they may also include function definitions, global variable declarations, input declarations, and a global update block.

A.3 Global variables

Global variables, which include input variables, provide a mechanism to communicate information to all of the strands in the program. They are typically declared and initialized at the beginning of the program and may be updated in the global update block. Globals may be annotated as **`input`** variables, in which case they can be initialized from outside the Diderot program (either from the command-line in a standalone executable or using a library call).

A.4 Functions

Diderot allows the definition of user-defined functions at top level. There are two syntactic forms. For a function whose definition can be given as an expression, we write

```
function type Name ( parameters ) = expression;
```

whereas if the function implementation requires local variables and/or statements, we write

```
function type Name ( parameters ) {
  statements
}
```

A.5 Strand definitions

Every Diderot program has a single strand definition, which has the form

```
strand Name ( parameters ) {
  state variables
  update { statements }
  stabilize { statements }
}
```

The (optional) strand parameters are used to pass instance-specific data to the strand initialization. The strand state variables hold the strand's state; some variables may annotated as **`output`** variables, which means that there final values are included in the result of the program. The strand state variables are initialized with the strand is created and are modified by the strand methods. Strands have one or two methods: the required **`update`** method that is invoked at each step and an optional **`stabilize`** method that is invoked when the strand stabilizes.

A.6 Global update block

Diderot programs may optionally include a global **`update`** block that is run at the end of each execution step. The code in this block may perform global reductions over the strands in the program and is also allowed to update global variables.

A.7 Global initially block

The global **`initially`** block is used to create the initial set of strands. Diderot uses a comprehension syntax, similar those of Haskell or Python, to define the initial set of strands. For example, the following code specifies a grid of initial ray positions:

```
initially [
    RayCast(ui, vi)
  | vi in 0 .. imgResV-1, ui in 0 .. imgResU-1
];
```

When the strands are initialized as a grid, it implies that the strands will all stabilize (*i.e.*, they do not die). The grid structure is then preserved in the output.

Diderot also allows one to specify an initial *collection* of strands by using "`{ }`" as the brackets around the comprehension (instead of "`[ ]`"). In this case, the program's output will be a one-dimension array of values for each **`output`** variable in a stable strand.

A.8 Statements

Statements in Diderot are a subset of C statements and include local-variable definitions, assignments, conditionals, and blocks that define a nested scope. In addition, there are several statement forms that are special to Diderot. The **`new`** statement is used to dynamically create a new strand. The **`stabilize`** and **`die`** statements terminate the calling strand; in the case of **`stabilize`**, the strand's state is preserved and may contribute to the remaining computation and to the result. Lastly, Diderot has a looping statement form (**`foreach`**) that provides iteration over dynamic sequences.

A.9 Expressions

Diderot's expression syntax is fairly standard, with infix binary operators, unary operators, function application (including field application), *etc.* Conditional expressions follow the Python syntax. We use Unicode characters for operators and literals to support traditional mathematical notation. Tensor values can be constructed from lower-order tensors using the notation `[`$e_1, \ldots, e_n$`]`. We also provide the special tensor literals **`identity`**`[`d`]` for the $d \times d$ identity matrix, **`zeros`**`[`σ`]` for the zero tensor of the given shape, and **`nan`**`[`σ`]` for the tensor of the given shape initialized to NaN.

A.10 Operators and built-in functions

Diderot provides a rich set of unary and binary operations, as well as various built-in functions. These can be be overloaded (*e.g.*, + works on integers, tensors, fields, *etc.*) and polymorphic (*e.g.*, + operates on tensors of any shape). A major use of overloading is to support *lifted* versions of tensor operations. We first present the overloaded operators and functions, along with the types at which they are defined. In this description, we use the type synonyms **`real`**, **`vec2`**, and **`vec3`**.

A.10.1 Overloaded operators and functions

Comparison operators: `<`,`<=`,`>`,`>=`

```
int * int → bool
real * real → bool
```

Equality operators: `==`, `!=`

int ⋆ **int** → **bool**
real ⋆ **real** → **bool**
bool ⋆ **bool** → **bool**
string ⋆ **string** → **string**

Minimum and maximum (including reductions): `min`, `max`

int ⋆ **int** → **int**
real ⋆ **real** → **real**
real{} → **real**

Unary negation: `-`

int → **int**
tensor[σ] → **tensor**[σ]
field#k(d)[σ] → **field**#k(d)[σ]

Logical negation: `not`

bool → **bool**

Addition operators: `+`, `-`

int ⋆ **int** → **int**
tensor[σ] ⋆ **tensor**[σ] → **tensor**[σ]
field#k(d)[σ] ⋆ **field**#k(d)[σ] → **field**#k(d)[σ]
tensor[σ] ⋆ **field**#k(d)[σ] → **field**#k(d)[σ]
field#k(d)[σ] ⋆ **tensor**[σ] → **field**#k(d)[σ]

Scalar multiplication: ⋆

int ⋆ **int** → **int**
real ⋆ **real** → **real**
real ⋆ **tensor**[σ] → **tensor**[σ]
tensor[σ] ⋆ **real** → **tensor**[σ]
real ⋆ **field**#k(d)[σ] → **field**#k(d)[σ]
tensor[σ] ⋆ **field**#k(d)[] -> **field**#k(d)[σ]
field#k(d)[] ⋆ **tensor**[σ] -> **field**#k(d)[σ]
field#k(d)[σ] ⋆ **real** → **field**#k(d)[σ]
field#k(d)[] ⋆ **field**#k(d)[] → **field**#k(d)[]
field#k(d)[] ⋆ **field**#k(d)[σ] → **field**#k(d)[σ]
field#k(d)[σ] ⋆ **field**#k(d)[] → **field**#k(d)[σ]

Scalar division: `/`

int ⋆ **int** → **int**
real ⋆ **real** → **real**
tensor[σ] ⋆ **real** → **real**
field#k(d)[σ] ⋆ **real** → **field**#k(d)[σ]
field#k(d)[] ⋆ **field**#k(d)[] → **field**#k(d)[]
field#k(d)[σ] ⋆ **field**#k(d)[] → **field**#k(d)[σ]

Scalar exponentiation: `^`

real ⋆ **int** → **real**
real ⋆ **real** → **real**

Inner product: •

tensor[σ_1,d] ⋆ **tensor**[d,σ_2] → **tensor**[σ_1,σ_2]
tensor[σ_1,d'] ⋆ **field**#k(d)[d',σ_2]
  → **field**#k(d)[σ_1,σ_2]
field#k(d)[σ_1,d'] ⋆ **tensor**[d',σ_2]
  → **field**#k(d)[σ_1,σ_2]
field#k(d)[σ_1,d'] ⋆ **field**#k(d)[d',σ_2]
  → **field**#k(d)[σ_1,σ_2]

Double-dot product: `:`

tensor[σ_1,d_1,d_2] ⋆ **tensor**[d_1,d_2,σ_2]
  → **tensor**[σ_1,σ_2]
tensor[σ_1,d_1,d_2] ⋆ **field**#k(d)[d_1,d_2,σ_2]
  → **field**#k(d)[σ_1,σ_2]
field#k(d)[σ_1,d_1,d_2] ⋆ **tensor**[d_1,d_2,σ_2]
  → **field**#k(d)[σ_1,σ_2]
field#k(d)[σ_1,d_1,d_2] ⋆ **field**#k(d)[d_1,d_2,σ_2]
  → **field**#k(d)[σ_1,σ_2]

Cross product: ×

vec2 ⋆ **vec2** → **real**
vec3 ⋆ **vec3** → **vec3**
field#k(d)[2] ⋆ **field**#k(d)[2] → **real**
field#k(d)[3] ⋆ **field**#k(d)[3] → **field**#k(d)[3]

Outer product: ⊗

tensor[d_1] ⋆ **tensor**[d_2] → **tensor**[d_1,d_2]
field#k(d)[d_1] ⋆ **field**#k(d)[d_2] → **field**#k(d)[d_1,d_2]

Convolution: ⊛

image(d)[σ] ⋆ **kernel**#k → **field**#k(d)[σ]
kernel#k ⋆ **image**(d)[σ] → **field**#k(d)[σ]

Gradient: ∇

field#$k+1$(d)[] → **field**#k(d)[d]

Tensor derivative: ∇⊗

field#$k+1$(d)[σ] → **field**#k(d)[sh,d]

Curl: ∇×

field#$k+1$(2)[2] → **field**#k(2)[]
field#$k+1$(3)[3] → **field**#k(3)[3]

Divergence: ∇•

field#$k+1$(d_1)[sh,d_2] → **field**#k(d)

Norm: `|·|`

tensor[σ] → **real**
field#k(d)[σ] → **field**#k(d)[]

Normalization: **`normalize`**

tensor[σ] → **tensor**[σ]
field#k(d_1)[σ] → **field**#k(d_1)[σ]

Trace of a matrix: `trace`

tensor[d,d] → **real**
field#k(d_1)[sh,d_2,d_2] → **field**#k(d_1)[σ]

Transpose of a matrix: `transpose`

tensor[d_1,d_2] → **tensor**[d_2,d_1]
field#k(d)[d_1,d_2] → **field**#k(d)[d_2,d_1]

Eigenvalues: `evals`

tensor[2,2] → **real**{2}
tensor[3,3] → **real**{3}

Eigenvectors: `evecs`

tensor[2,2] → **vec2**{2}
tensor[3,3] → **vec3**{3}

Determinate: `det`

tensor[2,2] → **real**
tensor[3,3] → **real**
field#k(d)[2,2] → **field**#k(d)[]
field#k(d)[3,3] → **field**#k(d)[]

Clamping: `clamp`

real ⋆ **real** ⋆ **real** → **real**
tensor[d] ⋆ **tensor**[d] ⋆ **tensor**[d] → **tensor**[d]

Lerp: `lerp`

tensor[σ] $\star$ **tensor**[σ] $\star$ **real** $\rightarrow$ **tensor**[σ]
tensor[σ] $\star$ **tensor**[σ] $\star$ **real** $\star$ **real** $\star$ **real**
$\rightarrow$ **tensor**[σ]

Trigonometry: `arccos`, `arcsin`, `cos`, `sin`

real $\rightarrow$ **real**
field#k(d)[] $\rightarrow$ **field**#k(d)[]

Sequence concatenation: `@`

τ $\star$ τ\{\} $\rightarrow$ τ\{\}
τ\{\} $\star$ τ $\rightarrow$ τ\{\}
τ\{\} $\star$ τ\{\} $\rightarrow$ τ\{\}

A.10.2 Other operators and functions

Pointwise vector multiplication: `modulate`

tensor[d] $\star$ **tensor**[d] $\rightarrow$ **tensor**[d]

Testing the domain of a field: **`inside`**

tensor[d] $\star$ **field**#k(d)[σ] $\rightarrow$ **bool**

Sequence length: `length`

τ\{\} $\rightarrow$ **int**

Boolean-sequence reductions: `all`, `exists`

bool\{\} $\rightarrow$ **bool**

Scalar-sequence reductions: `mean`, `product`, `sum`, `variance`

real\{\} $\rightarrow$ **real**

Loading images: **`image`**

string $\rightarrow$ **image**(d)[σ]

Image dimensions: **`load`**

image(d)[σ] $\rightarrow$ **int**\{d\}

Loading sequences: **`load`**

string $\rightarrow$ τ[σ]\{\}

B EMBEDDING DIDEROT INTO AN APPLICATION

In addition to compiling to a self-contained executable, a Diderot program may be compiled to a C library, which allows Diderot programs to be embedded in any language that supports calling C code. We demonstrate this here with the program that produced the dense isocontour sampling seen in Fig. 1(d). The complete Diderot code is below.

```
1  field#1(2)[] F = c4hexic ⊛ image("hand.nrrd");
2  input vec2 cent;
3  input real hght;
4  input int size0;
5  input int size1;
6  input int stepsMax = 10;
7  input real epsilon = 0.0001;
8  real wdth = hght*size0/size1;
9  vec2 spc = [wdth/(size0-1), hght/(size1-1)];
10 vec2 dir0 = [spc[0], 0.0];
11 vec2 dir1 = [0.0, spc[1]];
12 vec2 orig = cent - (dir0*(size0-1) + dir1*(size1-1))/2;
13
14 strand isofind(vec2 pos0) {
15   output vec2 pos = pos0;
16   int steps = 0;
17   update {
18     // Stop after too many steps or leaving field
19     if (steps > stepsMax || !inside(pos, F))
20       die;
21     // one Newton-Raphson iteration
22     vec2 delta = -normalize(∇F(pos)) * F(pos)/|∇F(pos)|;
23     pos += delta;
24     if (|delta| < epsilon)
25       stabilize;
26     steps += 1;
27   }
28 }
29
30 initially { isofind(orig + ui*dir0 + vi*dir1) |
31             vi in 0..(size1-1), ui in 0..(size0-1) };
```

The stand-alone executable produced by compiling the program above can be run with:

```
isofind -cent 61 40.7 -hght 53 -size0 50 -size1 30 -n 8
```

The executable has command-line options to set all the input variables; values must be given for those lacking defaults. Inputs `stepsMax` and `epsilon` do have defaults (lines 6 and 7). The final "`-n 8`" option sets the number of cores for parallel execution, available when the program was compiled for the SMP target.

When compiling a Diderot program to a library, the compiler produces functions for initializing the program, setting the values of input variables, stepping through program execution, and getting program output. To avoid symbol name clashes, one specifies to the Diderot compiler a string, for example "`ISO_`", which the compiler will use as a prefix for all symbols in the generated C library API. The C code below calls into the API generated from the Diderot program above, to replicate the effects of the invoking the stand-alone executable.

```
1  int main (int argc, const char **argv) {
2    ...
3
4    // Create execution context
5    ISO_World_t *wrld = ISO_New ();
6
7  #ifdef ISO_TARGET_PARALLEL
8    ISO_SetNumWorkers (wrld, 8);
9  #endif
10
11   // initialize the Diderot program
12   ISO_Init (wrld);
13
14   // initialize inputs
15   float cent[2] = {61, 40.7};
16   ISO_InVarSet_cent (wrld, cent);
17   ISO_InVarSet_hght (wrld, 53);
18   ISO_InVarSet_size0 (wrld, 50);
19   ISO_InVarSet_size1 (wrld, 30);
20
21   // global initialization
22   ISO_Initially (wrld);
23
24   // wrapper for output data
25   Nrrd *nout = nrrdNew();
26
27   // Main loop
28   while (ISO_NumActive(wrld) > 0) {
29     // get and render the state
30     if (ISO_Snapshot_pos(wrld, nout)) {
31       ⋯ handle error ⋯
32     }
33     Draw (nout);
34     // step the computation
35     ISO_Run (wrld, 1);
36   }
37
38   // get and render final state
39   if (ISO_OutputGet_pos(wrld, nout)) {
40     ⋯ handle error ⋯
41   }
42   Draw (nout);
43
44   // shutdown the world
45   ISO_Shutdown (wrld);
46   ...
47 }
```

Lines 4–25 start the Diderot program. Line 5 allocates a new execution context, called a *world*. This context contains all runtime state of the program. Line 8 requests 8 cores for parallel execution, but only when the program was compiled for the SMP target. Line 12 allocates the computational resources needed to support the program's execution (such as memory and threads). Lines 15 through 19 initialize those program input variables lacking defaults (the same ones set via command-line options above). Line 23 initializes the program, including allocating the initial set of strands. Line 25 allocates a `Nrrd` struct to contain the program output.

The `while` loop (lines 28–36) runs the program until all of the strands have stabilized or died. For each iteration of the loop, we grab (line 30) a snapshot of the program state (the union of all active and stable strands) and display it (line 33) with some `Draw` function, the details of which are not important here. For this program, which computes a *collection* of strands (due to the **`initially`** `{ ... }` in lines 30 and 31 of Diderot code above), the output allocated in `nout` by `ISO_Snapshot_pos` will be a two-dimensional 2–by–*N* array, where *N* is the number of output values, and the (faster) first axis is length 2 because the output variable `pos` is a **`vec2`**. Had this program initialized on *grid* of strands (with **`initially`** `[ ... ]`) the output would be a higher dimensional array, with one axis for each dimension of the grid, and an additional axis if the output variable was not **`real`**.

Each `while` loop iteration executes (line 35) one super-step of the Diderot program. It is also possible to run all strands to completion (without being able to access snapshots of intermediate state) with `ISO_Run(wrld,0)`. The `while` loop terminates when there are zero active strands (line 28). The final output of the program execution is grabbed by line 39 and displayed in line 42. The resources allocated by program initialization and execution are released by line 45.

C COMPARISON WITH HAND-CODED OPENCL

We show here comparisons between Diderot and hand-coded OpenCL code, for two simple volume renders: a maximum intensity projection (MIP), and isosurface rendered colormapped by the second derivative. These examples are somewhat favorable to OpenCL due to their simplicity compared to the previous examples in the paper, but the properties of the code, and changes required for a more sophisticated algorithm, nonetheless demonstrate benefits of Diderot.

```
 1  // coefficients of piece-wise cubic bspln3
 2  __constant float h[4][4] = {
 3    { 1.33333f,   2.0f,  1.0f,  0.166667f },  // -2 .. -1
 4    { 0.666667f,  0.0f, -1.0f, -0.5f },       // -1 .. 0
 5    { 0.666667f,  0.0f, -1.0f,  0.5f },       //  0 .. 1
 6    { 1.33333f,  -2.0f,  1.0f, -0.166667f },  //  1 .. 2
 7  };
 8  __constant int hs = 2; // half of support=4
 9
10  #define FLOAT4(p) (float4)((p)[0], (p)[1], (p)[2], (p)[3])
11
12  static inline float probeVal (global ushort * volData,
13                                int sx, int sy, float4 volPos) {
14    float4 nf, t;
15
16    float4 d = (float4) (h[3][0],h[2][0],h[1][0],h[0][0]);
17    float4 c = (float4) (h[3][1],h[2][1],h[1][1],h[0][1]);
18    float4 b = (float4) (h[3][2],h[2][2],h[1][2],h[0][2]);
19    float4 a = (float4) (h[3][3],h[2][3],h[1][3],h[0][3]);
20
21    float4 f = modf(volPos,&nf);
22    int4 n = convert_int4(nf);
23
24    t = (float4) (f.x + 1, f.x, f.x -1, f.x - 2);
25    float4 hx = d + t * (c + t * (b + t*a));
26    t = (float4) (f.y + 1, f.y, f.y - 1, f.y - 2);
27    float4 hy = d + t * (c + t * (b + t*a));
28    t = (float4) (f.z + 1, f.z, f.z - 1, f.z -2);
29    float4 hz = d + t * (c + t * (b + t*a));
30
31    float vx[4], vy[4];   // 4 == kernel support
32
33    for (int k = 1-hs; k <= hs; k++) {    // slowest
34      for (int j = 1-hs; j <= hs; j++) { // medium
35        int index = n.x - 1 + sx*(n.y + j + sy*(n.z + k));
36        float4 v = FLOAT4(volData + index);
37        vx[j+hs-1] = dot(v, hx);
38      }
39      vy[k+hs-1] = dot(FLOAT4(vx),hy);
40    }
41    return dot(FLOAT4(vy), hz);
42  }
43
44  __constant float4 eye =
45            (float4)(-240.538f, 396.133f, 64.6804f, 1.0f);
46  __constant float4 orig =
47            (float4)(-47.5009f, 48.5999f, 71.5282f, 1.0f);
48  __constant float4 du =
49            (float4)(0.170942f, 0.146426f, -0.0851239f, 0.0f);
50  __constant float4 dv =
51            (float4)(0.049961f, 0.0720342f, 0.224239f, 0.0f);
52  __constant float raystep = 0.3f;
53  __constant float lenmax = 130;
54
55  __kernel void kern (global ushort * volData,
56                      float16 w2iPos,
57                      float16 i2wGrad, // (unused)
58                      global int * volSize,
59                      int imgSizeU, int imgSizeV,
60                      global float * out) {
61    int ui = get_global_id(0), vi = get_global_id(1);
62    int sx = volSize[0];
63    int sy = volSize[1];
64    int sz = volSize[2];
65
66    float4 pos0 = orig + ui*du + vi*dv;
67    float3 raydir = normalize(pos0 - eye).s012;
68    float mip = 0;
69
70    for (float rlen = 0; rlen < lenmax; rlen += raystep) {
71      float4 wpos = pos0 + rlen*raydir;
72      // Transform position from world to index
73      float4 ipos = (float4)(dot(wpos, w2iPos.s0123),
74                             dot(wpos, w2iPos.s4567),
75                             dot(wpos, w2iPos.s89ab),
76                             dot(wpos, w2iPos.scdef));
77      if ((ipos.x > 1) && (ipos.x < sx - 2)
78      &&  (ipos.y > 1) && (ipos.y < sy - 2)
79      &&  (ipos.z > 1) && (ipos.z < sz - 2)) {
80        float val = probeVal(volData, sx, sy, ipos);
81        mip = fmax(val, mip);
82      }
83    }
84
85    out[ui + imgSizeU*vi] = mip;
86  }
```

Above is hand-written OpenCL code for a simple ray-casting maximum intensity projection volume renderer. The code begins (lines 2-7) with a hard-coded definition of cubic B-spline kernel, in terms of its four piecewise cubic segments over $[-2,2]$. The `probeVal` function (lines 12-42) implements convolution-based reconstruction with this kernel. The evaluation of the B-spline kernel at the grid sample locations is four-way parallelized in the setting of `hx`, `hy`, and `hz` (lines 25, 27, 29). The Diderot compiler generates the same parallelization for kernels with support four. After defining simple camera and ray-casting parameters (lines 44-53), the remainder of the code defines the computational kernel `kern`, which computes the maximum intensity projection for a single ray, indexed by `ui` and `vi`. Rays are traversed in world-space, and converted (lines 73-76) to index space, to determine (lines 77-79) if the sample location `ipos` falls within the volume data, before evaluating the field with `probeVal`. There are about 400 additional lines of hand-written C driver code (not shown) for compiling and running the kernel, and for saving its output.

```
 1  vec3 eye = [-240.538, 396.133, 64.6804];
 2  vec3 orig = [-47.5009, 48.5999, 71.5282];
 3  vec3 du = [0.170942, 0.146426, -0.0851239];
 4  vec3 dv = [0.049961, 0.0720342, 0.224239];
 5  real raystep = 0.3;
 6  real lenmax = 130;
 7
 8  field#2(3)[] F = bspln3 ⊛ image ("vfrhand.nrrd");
 9  input int imgSizeU;
10  input int imgSizeV;
11
12  strand raycast (int vi, int ui) {
13      vec3 pos0 = orig + ui*du + vi*dv;
14      vec3 raydir = normalize(pos0 - eye);
15      real rlen = 0;
16      output real mip = 0;
17
18      update {
19          vec3 pos = pos0 + rlen*raydir;
20          if (inside(pos, F)) {
21              mip = max(F(pos), mip);
22          }
23          rlen += raystep;
24          if (rlen >= lenmax) {
25              stabilize;
26          }
27      }
28  }
29  initially [ raycast(vi, ui) | vi in 0..imgSizeV-1,
30                                ui in 0..imgSizeU-1 ];
```

Above is the same MIP renderer written in Diderot. The main difference is that the Diderot code is expressed in terms of world space, and that the details of convolution are hidden behind "⊛" and "`F(pos)`". The code for **`update`** (lines 12-28) is similar to the OpenCL code. This is expected: Diderot is something like a shader language for tensors and fields. However, to change the kernel in Diderot, only the kernel name needs changing on line 8. In OpenCL, the kernel coefficients would be different, and the details of the OpenCL `probeVal` function would be significantly different for a kernel with smaller or larger support, or higher or lower polynomial order. The Diderot compiler automatically generates all the necessary code for convolution in OpenCL, as well as the driver code needed to compile and execute the computational kernels. Both programs produce the same rendering, seen in Fig. 12.

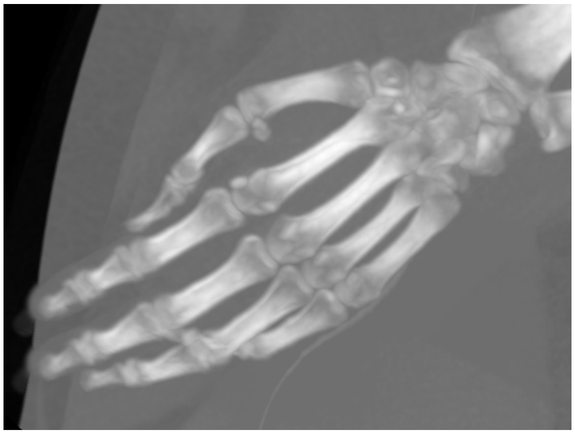


Fig. 12. 640×480 maximum intensity projection of right hand of Visible Human female CT scan.

The Diderot-generated OpenCL code is not has fast as the hand-written code, although it is more flexible. On the same platform as used for benchmarking (Sec. 6), the OpenCL code took on average **0.53 seconds** to run (excluding compilation and disk I/O), while the Diderot code took on average **1.46 seconds**. As noted in Sec. 6, our OpenCL implementation does not currently perform as well as might be hoped; work on it continues. On the other hand, as noted in Sec. 1, in creating Diderot we are not seeking to optimize execution time so much as human implementation time, with the long-term goal of assisting the rapid development and application of novel visualization algorithms.

The next rendering example illustrates this. Below is Diderot code for a kind of shaded pseudo-isosurface volume rendering. The rendering algorithm was designed to help assess whether an isosurface coincides with a material edge, considered as the zero-crossing of the second-directional derivative along the gradient direction. This is complementary to the Canny edge criterion visualized in Sec. 4.1, and requires only second (not third) derivatives.

```
 1 vec3 eye = [-240.538, 396.133, 64.6804];
 2 vec3 orig = [-47.5009, 48.5999, 71.5282];
 3 vec3 du = [0.170942, 0.146426, -0.0851239];
 4 vec3 dv = [0.049961, 0.0720342, 0.224239];
 5 real raystep = 0.1;
 6 real lenmax = 130;
 7
 8 real valIso = 1440;
 9 field#2(3)[] F = bspln3 ⊛ image ("vfrhand.nrrd") - valIso;
10 input int imgSizeU;
11 input int imgSizeV;
12 real valTol = 150;
13 real isoThick = 3;
14 vec3 light = normalize([-1,2,-1]);
15 vec3 gray = 0.7*[1,1,1];
16
17 function real atxf (real vv, real gg) {
18   return clamp(0, 1, 1 - |vv|/(gg*isoThick));
19 }
20
21 strand raycast (int vi, int ui) {
22    vec3 pos0 = orig + ui*du + vi*dv;
23    vec3 raydir = normalize(pos0 - eye);
24    real rlen = 0;
25    output vec4 rgba = [0,0,0,0];
26    vec3 rgb = [0,0,0];
27    real transp = 1;
28
29    update {
30       vec3 pos = pos0 + rlen*raydir;
31       if (inside (pos, F) && |F(pos)| <= valTol) {
32          real alpha = atxf(F(pos), |∇F(pos)|);
33          if (alpha > 0) {
34             vec3 norm = -normalize(∇F(pos));
35             real lit = lerp(0, 1, -1, norm•light, 1)^2;
36             real sdd = norm • ∇⊗∇F(pos) • norm/600;
37             vec3 col = lerp(gray, [1.0, 0.0, 0.7], -sdd)
38                        if (sdd < 0) else
39                        lerp(gray, [0.3, 1.0, 0.3], sdd);
40             rgb += transp*alpha*lit*col;
41             transp *= 1 - alpha;
42          }
43       }
44       rlen += raystep;
45       if (rlen >= lenmax) {
46          stabilize;
47       }
48    }
49    stabilize {
50       real a = 1 - transp;
51       if (a > 0) {
52          rgba = [rgb[0]/a, rgb[1]/a, rgb[2]/a, a];
53       }
54    }
55 }
56 initially [ raycast(vi, ui) | vi in 0..imgSizeV-1,
57                               ui in 0..imgSizeU-1 ];
```

The key lines are 36-39: the Hessian `∇⊗∇F(pos)` is contracted (by •) on both sides by the unit-length isosurface normal `norm = -normalize(∇F(pos))` to find the second directional derivative `sdd` of `F` along `norm`. Depending on the sign of `sdd`, a per-sample color `col` is found by lerping between gray and magenta (negative) or green (positive). Assuming materials of interest are more dense than the background, if the isovalue is too low, the isosurface is outside the material boundary and hence in a region of positive second derivative. Conversely, isovalues too high will put the isosurface within a negative second derivative.

This is precisely what is seen in the resulting rendering, in Fig. 13. The isosurface is outside the dense cortical bone surface in the middle of the phalanges, but inside the thinner bone surface at the joints. Four lines 36-39 of code make an otherwise generic isosurface volume rendering into an informative illustration of the mathematical properties of isosurfaces and the bone surface. The Diderot programmer is unburdened by the complexity associated with correctly computing `∇⊗∇F(pos)` in world space. For this more complex rendering task, however, the performance hit from Diderot (relative to hand-written OpenCL code) is **smaller** than in the MIP example. The hand-written OpenCL code took on average **1.84** seconds, while the Diderot code took on average **3.30** seconds.

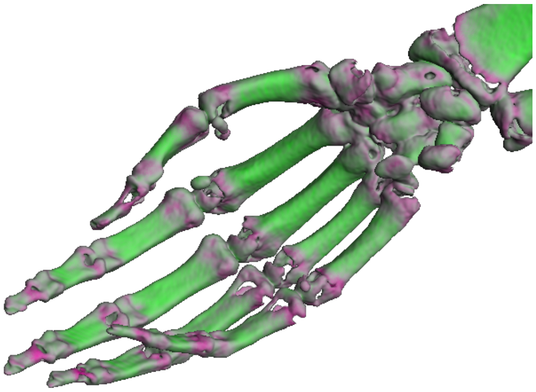


Fig. 13. 640×480 rendering of isosurface at 1440 of right hand of Visible Human female CT scan, colored by second directional derivative (green: positive, gray: zero, magenta: negative).

The OpenCL code is below. The computational kernel (starting line 194) is about as legible as the Diderot **`update`** code, though the double contraction of the Hessian (lines 232-234) is not as clear. Also, whereas the Diderot compiler can easily optimize repetitions of "`∇F(pos)`" with common subexpression elimination, the OpenCL programmer is more likely to call functions like `probeGrad` once (line 224) and save the results in a suggestively named variable. Given the mathematical limitations of OpenCL, is to hard to correctly implement `probeGrad` (lines 44-86) for reconstructing the world-space gradient, and `probeHess` (lines 88-166) for the Hessian, especially the conversion of the index-space Hessian to world-space (lines 144-163). Even with these utility functions written for scalar data, however, the OpenCL programmer is no closer to having code for evaluating $\nabla(|\nabla F|)$ let alone $\nabla(\nabla(|\nabla F|) \bullet \nabla F / |\nabla F|)$ used for rendering Canny edges, which Diderot generated automatically. Additional OpenCL code would be needed for doing convolution-based reconstruction of vector- and tensor-valued data. Handling derivatives in vector and tensor fields (including conversions from index to world space) would be challenging even for the most meticulous programmer. We suggest that this is exactly the kind of tedious error-prone work that should be automated by the compiler of a high-level language like Diderot.

```
 1 // coefficients of piece-wise cubic bspln3
 2 __constant float h[4][4] = {
 3   { 1.33333f,   2.0f,  1.0f,  0.166667f },  // -2 .. -1
 4   { 0.666667f,  0.0f, -1.0f, -0.5f },       // -1 .. 0
 5   { 0.666667f,  0.0f, -1.0f,  0.5f },       //  0 .. 1
 6   { 1.33333f,  -2.0f,  1.0f, -0.166667f },  //  1 .. 2
 7 };
 8 __constant int hs = 2; // half of support=4
 9
10 #define FLOAT4(p) (float4)((p)[0], (p)[1], (p)[2], (p)[3])
11
12 static inline float probeVal (global ushort * volData,
13                               int sx, int sy, float4 volPos) {
14   float4 nf, t;
15
16   float4 d = (float4) (h[3][0],h[2][0],h[1][0],h[0][0]);
17   float4 c = (float4) (h[3][1],h[2][1],h[1][1],h[0][1]);
18   float4 b = (float4) (h[3][2],h[2][2],h[1][2],h[0][2]);
19   float4 a = (float4) (h[3][3],h[2][3],h[1][3],h[0][3]);
20
21   float4 f = modf(volPos,&nf);
22   int4 n = convert_int4(nf);
23
24   t = (float4) (f.x + 1, f.x, f.x -1, f.x - 2);
```

```
  float4 hx = d + t * (c + t * (b + t*a));
  t = (float4) (f.y + 1, f.y, f.y - 1, f.y - 2);
  float4 hy = d + t * (c + t * (b + t*a));
  t = (float4) (f.z + 1, f.z, f.z - 1, f.z -2);
  float4 hz = d + t * (c + t * (b + t*a));

  float vx[4], vy[4];   // 4 == kernel support

  for (int k = 1-hs; k <= hs; k++) {   // slowest
    for (int j = 1-hs; j <= hs; j++) { // medium
      int index = n.x - 1 + sx*(n.y + j + sy*(n.z + k));
      float4 v = FLOAT4(volData + index);
      vx[j+hs-1] = dot(v, hx);
    }
    vy[k+hs-1] = dot(FLOAT4(vx),hy);
  }
  return dot(FLOAT4(vy), hz);
}

static inline float3 probeGrad (float16 i2wGrad,
                                global ushort * volData,
                                int sx, int sy, float4 volPos) {
  float4 nf, t;

  float4 d = (float4) (h[3][0],h[2][0],h[1][0],h[0][0]);
  float4 c = (float4) (h[3][1],h[2][1],h[1][1],h[0][1]);
  float4 b = (float4) (h[3][2],h[2][2],h[1][2],h[0][2]);
  float4 a = (float4) (h[3][3],h[2][3],h[1][3],h[0][3]);

  float4 f = modf(volPos,&nf);
  int4 n = convert_int4(nf);

  t = (float4) (f.x + 1, f.x, f.x -1, f.x - 2);
  float4 h0x = d + t*(c + t*(b + t*a));
  float4 h1x = c + t*(2*b + t*3*a);
  t = (float4) (f.y + 1, f.y, f.y - 1, f.y - 2);
  float4 h0y = d + t*(c + t*(b + t*a));
  float4 h1y = c + t*(2*b + t*3*a);
  t = (float4) (f.z + 1, f.z, f.z - 1, f.z -2);
  float4 h0z = d + t*(c + t*(b + t*a));
  float4 h1z = c + t*(2*b + t*3*a);

  float v0x[4], v1x[4], v0x0y[4], v1x0y[4], v0x1y[4];

  for (int k = 1-hs; k <= hs; k++) {   // slowest
    for (int j = 1-hs; j <= hs; j++) { // medium
      int index = n.x - 1 + sx*(n.y + j + sy*(n.z + k));
      float4 v = FLOAT4(volData + index);
      v0x[j+hs-1] = dot(v, h0x);
      v1x[j+hs-1] = dot(v, h1x);
    }
    v0x0y[k+hs-1] = dot(FLOAT4(v0x), h0y);
    v0x1y[k+hs-1] = dot(FLOAT4(v0x), h1y);
    v1x0y[k+hs-1] = dot(FLOAT4(v1x), h0y);
  }
  float3 igrad = (float3)(dot(FLOAT4(v1x0y), h0z),
                          dot(FLOAT4(v0x1y), h0z),
                          dot(FLOAT4(v0x0y), h1z));
  return (float3)(dot(igrad, i2wGrad.s012),
                  dot(igrad, i2wGrad.s456),
                  dot(igrad, i2wGrad.s89a));
}

static inline float16 probeHess (float16 i2wGrad,
                                 global ushort * volData,
                                 int sx, int sy, float4 volPos) {
  float4 nf, t;

  float4 d = (float4) (h[3][0],h[2][0],h[1][0],h[0][0]);
  float4 c = (float4) (h[3][1],h[2][1],h[1][1],h[0][1]);
  float4 b = (float4) (h[3][2],h[2][2],h[1][2],h[0][2]);
  float4 a = (float4) (h[3][3],h[2][3],h[1][3],h[0][3]);

  float4 f = modf(volPos,&nf);
  int4 n = convert_int4(nf);
  float16 ret;

  t = (float4) (f.x + 1, f.x, f.x -1, f.x - 2);
  float4 h0x = d + t*(c + t*(b + t*a));
  float4 h1x = c + t*(2*b + t*3*a);
  float4 h2x = 2*b + t*6*a;
  t = (float4) (f.y + 1, f.y, f.y - 1, f.y - 2);
  float4 h0y = d + t*(c + t*(b + t*a));
  float4 h1y = c + t*(2*b + t*3*a);
  float4 h2y = 2*b + t*6*a;
  t = (float4) (f.z + 1, f.z, f.z - 1, f.z -2);
  float4 h0z = d + t*(c + t*(b + t*a));
  float4 h1z = c + t*(2*b + t*3*a);
  float4 h2z = 2*b + t*6*a;

  float v0x[4], v1x[4], v2x[4],
    v0x0y[4], v0x1y[4], v0x2y[4],
    v1x0y[4], v1x1y[4], v2x0y[4];

  for (int k = 1-hs; k <= hs; k++) {   // slowest
    for (int j = 1-hs; j <= hs; j++) { // medium
      int index = n.x - 1 + sx*(n.y + j + sy*(n.z + k));
      float4 v = FLOAT4(volData + index);
      v0x[j+hs-1] = dot(v, h0x);
      v1x[j+hs-1] = dot(v, h1x);
      v2x[j+hs-1] = dot(v, h2x);
    }
    v0x0y[k+hs-1] = dot(FLOAT4(v0x), h0y);
```

```
    v0x1y[k+hs-1] = dot(FLOAT4(v0x), h1y);
    v0x2y[k+hs-1] = dot(FLOAT4(v0x), h2y);
    v1x0y[k+hs-1] = dot(FLOAT4(v1x), h0y);
    v1x1y[k+hs-1] = dot(FLOAT4(v1x), h1y);
    v2x0y[k+hs-1] = dot(FLOAT4(v2x), h0y);
  }
  // hessian in index space
  float3 ihes0 = (float3)(dot(FLOAT4(v2x0y), h0z),
                          dot(FLOAT4(v1x1y), h0z),
                          dot(FLOAT4(v1x0y), h1z));
  float3 ihes1 = (float3)(ihes0.s1,
                          dot(FLOAT4(v0x2y), h0z),
                          dot(FLOAT4(v0x1y), h1z));
  float3 ihes2 = (float3)(ihes0.s2,
                          ihes1.s2,
                          dot(FLOAT4(v0x0y), h2z));
  // convert hessian to world space, 2nd index
  float3 thes0 = (float3)(dot(ihes0, i2wGrad.s012),
                          dot(ihes1, i2wGrad.s012),
                          dot(ihes2, i2wGrad.s012));
  float3 thes1 = (float3)(dot(ihes0, i2wGrad.s456),
                          dot(ihes1, i2wGrad.s456),
                          dot(ihes2, i2wGrad.s456));
  float3 thes2 = (float3)(dot(ihes0, i2wGrad.s89a),
                          dot(ihes1, i2wGrad.s89a),
                          dot(ihes2, i2wGrad.s89a));
  // convert hessian to world space, 1st index
  ret.s0123 = (float4)(dot(thes0, i2wGrad.s012),
                       dot(thes1, i2wGrad.s012),
                       dot(thes2, i2wGrad.s012), 0.0f);
  ret.s4568 = (float4)(dot(thes0, i2wGrad.s456),
                       dot(thes1, i2wGrad.s456),
                       dot(thes2, i2wGrad.s456), 0.0f);
  ret.s89ab = (float4)(dot(thes0, i2wGrad.s89a),
                       dot(thes1, i2wGrad.s89a),
                       dot(thes2, i2wGrad.s89a), 0.0f);
  ret.scdef = (float4)(0.0f, 0.0f, 0.0f, 1.0f);
  return ret;
}

__constant float4 eye =
          (float4)(-240.538f, 396.133f, 64.6804f, 1.0f);
__constant float4 orig =
          (float4)(-47.5009f, 48.5999f, 71.5282f, 1.0f);
__constant float4 du =
          (float4)(0.170942f, 0.146426f, -0.0851239f, 0.0f);
__constant float4 dv =
          (float4)(0.049961f, 0.0720342f, 0.224239f, 0.0f);
__constant float raystep = 0.1f;
__constant float lenmax = 130;
__constant float valIso = 1440;
__constant float valTol = 150;
__constant float isoThick = 3;
__constant float3 light = (float3)(-0.4082f,0.8165f,-0.4082f);
__constant float3 gray = (float3)(0.7);

static inline float atxf (float vv, float gg) {
  float av = fabs(vv - valIso);
  return clamp(1 - av/(gg*isoThick), 0.0f, 1.0f);
}

__kernel void kern (global ushort * volData,
                    float16 w2iPos,
                    float16 i2wGrad,
                    global int * volSize,
                    int imgSizeU, int imgSizeV,
                    global float * out) {
  int ui = get_global_id(0), vi = get_global_id(1);
  int sx = volSize[0];
  int sy = volSize[1];
  int sz = volSize[2];

  float4 pos0 = orig + ui*du + vi*dv;
  float3 raydir = normalize(pos0 - eye).s012;
  float gmsum = 0;

  float transp = 1.0;
  float3 rgb = (float3)(0.0f, 0.0f, 0.0f);
  for (float rlen = 0; rlen < lenmax; rlen += raystep) {
    float4 wpos = pos0 + rlen*raydir;
    // Transform position from world to index
    float4 ipos = (float4)(dot(wpos, w2iPos.s0123),
                           dot(wpos, w2iPos.s4567),
                           dot(wpos, w2iPos.s89ab),
                           dot(wpos, w2iPos.scdef));
    if (!((ipos.x > 1) && (ipos.x < sx - 2)
      &&  (ipos.y > 1) && (ipos.y < sy - 2)
      &&  (ipos.z > 1) && (ipos.z < sz - 2)))
      continue;
    float val = probeVal(volData, sx, sy, ipos);
    if (fabs(val - valIso) > valTol)
      continue;
    float3 grad = probeGrad(i2wGrad, volData, sx, sy, ipos);
    float alpha = atxf(val, length(grad));
    if (0 == alpha)
      continue;
    float3 norm = normalize(-grad);
    float lit = (dot(norm,light) + 1)/2;
    lit *= lit;
    float16 hess = probeHess(i2wGrad, volData, sx, sy, ipos);
    float sdd = dot(norm, (float3)(dot(norm, hess.s012),
                                   dot(norm, hess.s456),
                                   dot(norm, hess.s89a)))/600;
```

```
    float3 col = ((sdd < 0)
                  ? mix(gray, (float3)(1.0, 0.0, 0.7), -sdd)
                  : mix(gray, (float3)(0.3, 1.0, 0.3), sdd));
    rgb += transp*alpha*lit*col;
    transp *= 1 - alpha;
  }
  global float *opix = out + 4*(ui + imgSizeU*vi);
  float aa = 1 - transp;
  if (aa > 0) {
    opix[0] = rgb.s0/aa;
    opix[1] = rgb.s1/aa;
    opix[2] = rgb.s2/aa;
    opix[3] = aa;
  } else {
    opix[0] = 0;
    opix[1] = 0;
    opix[2] = 0;
    opix[3] = 0;
  }
}
```