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

Gordon Kindlmann
glk@uchicago.edu

Joint work with: Prof. John Reppy, Nicholas Seltzer, Charisee Chiw, and Lamont Samuels

THE UNIVERSITY OF CHICAGO
COMPUTER SCIENCE

Computation Institute

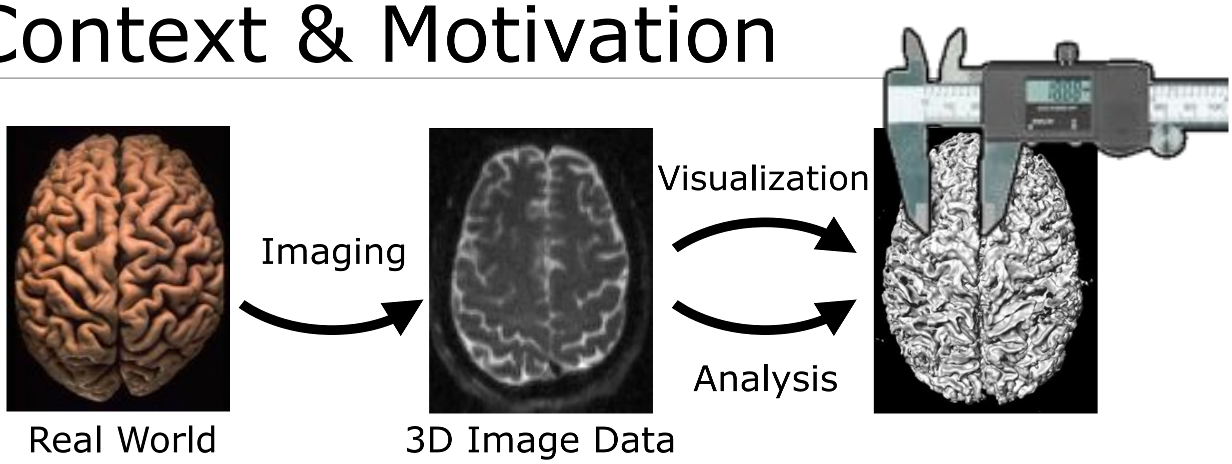
Outline

- Context & Motivation
- Language design
- Example programs
- Looking forward

Outline

- Context & Motivation
- Language design
- Example programs
- Looking forward

Context & Motivation



- Scientists study world by using software to show/extract structure from images
- Creating new visualization/analysis tools is important part of the scientific process
- But this is not easy ...

Creating these tools is hard

Increasing range of: Imaging **modalities** | Imaging **applications** | Vis & analysis **algorithms**

Want to rapidly implement variety of programs

Diderot helps rapidly develop portably parallel methods of image visualization and analysis

Example problems....

Want **portably** parallel implementations

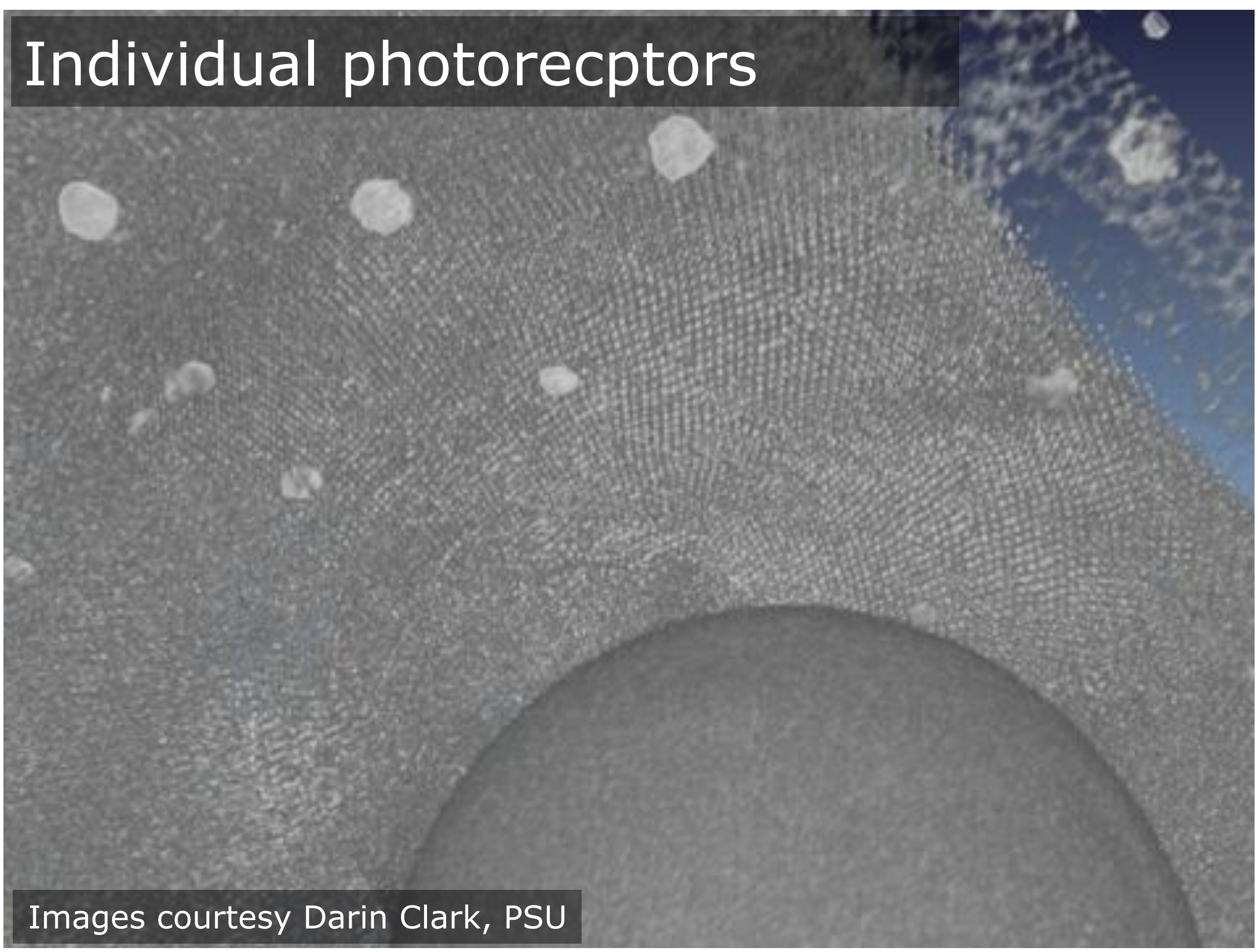
Increasing **data size** → Need **parallel** computing | Rapidly shifting parallel computing architectures

Genetics of Model Organisms w/ MicroCT

- US Argonne National Lab; Advanced Photon Source
- Multiple beamlines, one for microscopic CT

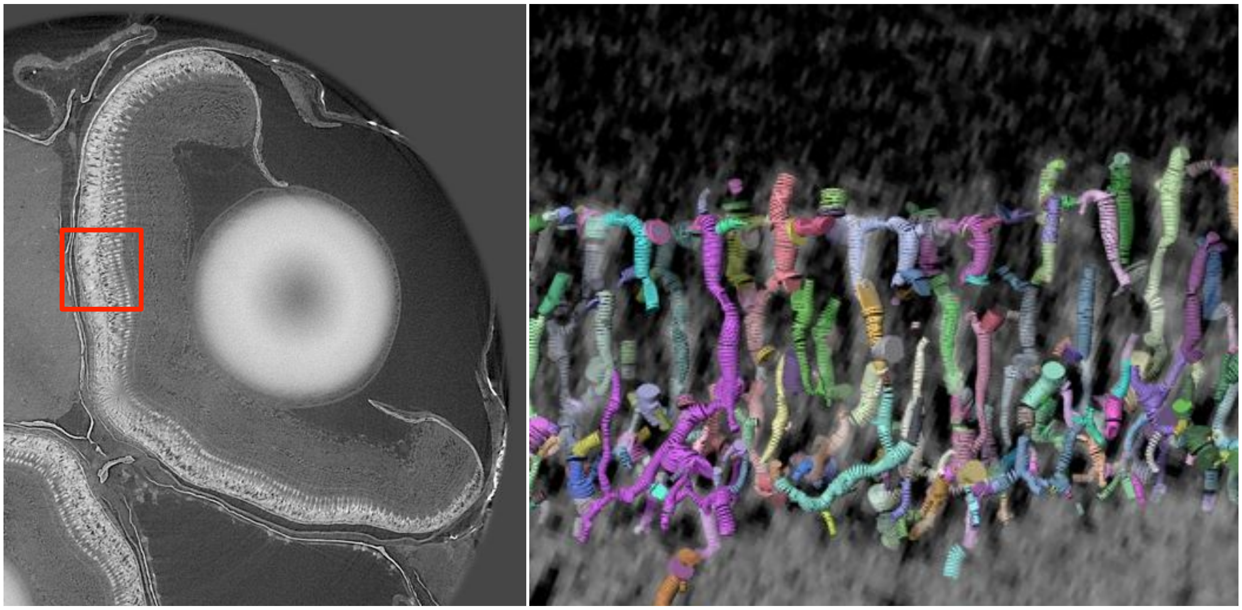
- ~5 micron resolution; output volumes 2000 x 2000 x 4000 (versus clinical CT ≈ 256 x 256 x 256)
- Zebrafish standard "model organism"
- Study with high-res whole-body microCT

Visualization: Volume Rendering
Images courtesy Darin Clark, PSU



Individual photorecptors
Images courtesy Darin Clark, PSU

Extraction of micro-anatomy



- Extraction of individual photoreceptors from microCT
  - Using Scale-Space Particles (Kindlmann et al. Vis'09)
- Wealth of anatomical features at larger scales
  - Not yet implemented!

Digital Light Sheet Microscopy

- Kevin White, Institute for Genomics and Systems Biology, U of Chicago

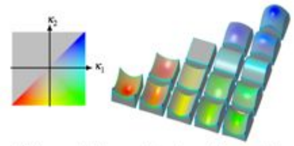
- Drosophila embryo-genesis imaged over 20 hours

- 5 terabytes/ day of image data

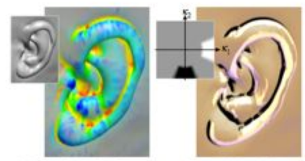
Example visualization method

- Curvature-based transfer functions (Vis'03)

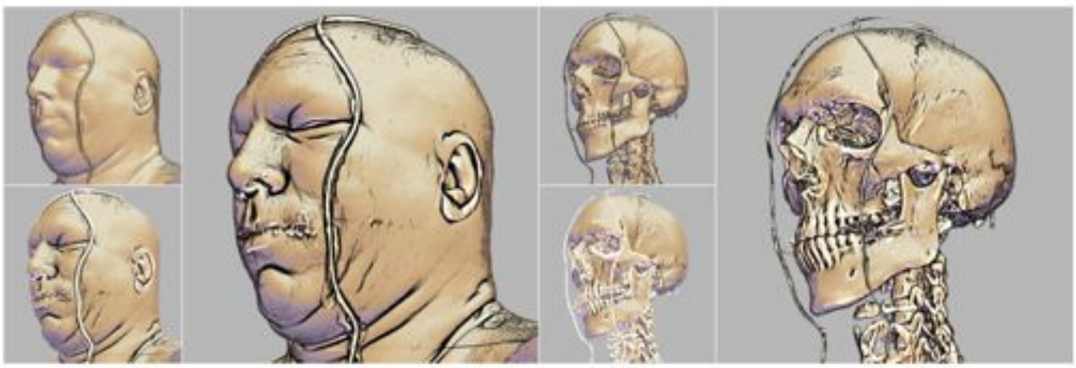
$$
\begin{aligned}
\nabla \mathbf{n}^T &= -\nabla\left(\frac{\mathbf{g}^T}{|\mathbf{g}|}\right) = -\left(\frac{\nabla \mathbf{g}^T}{|\mathbf{g}|} - \frac{\mathbf{g}\,\nabla^T|\mathbf{g}|}{|\mathbf{g}|^2}\right) \\
&= -\frac{1}{|\mathbf{g}|}\left(\mathbf{H} - \frac{\mathbf{g}\,\nabla^T(\mathbf{g}^T\mathbf{g})^{1/2}}{|\mathbf{g}|}\right) = -\frac{1}{|\mathbf{g}|}\left(\mathbf{H} - \frac{\mathbf{g}\,\nabla^T(\mathbf{g}^T\mathbf{g})}{2\,|\mathbf{g}|(\mathbf{g}^T\mathbf{g})^{1/2}}\right) \\
&= -\frac{1}{|\mathbf{g}|}\left(\mathbf{H} - \frac{\mathbf{g}\,(2\mathbf{g}^T\mathbf{H})}{2\,|\mathbf{g}|^2}\right) = -\frac{1}{|\mathbf{g}|}\left(\mathbf{I} - \frac{\mathbf{g}\mathbf{g}^T}{|\mathbf{g}|^2}\right)\mathbf{H} \\
&= -\frac{1}{|\mathbf{g}|}(\mathbf{I} - \mathbf{n}\mathbf{n}^T)\mathbf{H}\,.
\end{aligned}
$$



(a) Volume rendered diagram of (κ_1, κ_2) space. The colors in the (κ_1, κ_2) transfer function domain are mapped onto the patches with corresponding surface curvature.



(b) Left: Visualization of ear curvature using transfer function from (a); Right: ridge and valley emphasis implemented with inset transfer function, combined with Gooch shading



The C code to implement that

```
if (                                              ens)) {
  if
                                                  ine the sense of the
if (
  do
  T
  N
  D
  D
  D
  k1
  k2
}
```

```
#define DOT_4(a,b) ((a)[0]*(b)[0]+(a)[1]*(b)[1]+(a)[2]*(b)[2]+(a)[3]*(b)[3])
#define VL_4(i, axis) DOT_4(fw0 + (axis)*4, iv##axis + i*4)
#define D1_4(i, axis) DOT_4(fw1 + (axis)*4, iv##axis + i*4)
#define D2_4(i, axis) DOT_4(fw2 + (axis)*4, iv##axis + i*4)

  /* x0 */
  ivY[ 0] = VL_4( 0,X);
  ivY[ 1] = VL_4( 1,X);
  ivY[ 2] = VL_4( 2,X);
  ivY[ 3] = VL_4( 3,X);
  ivY[ 4] = VL_4( 4,X);
  ivY[ 5] = VL_4( 5,X);
  ivY[ 6] = VL_4( 6,X);
  ivY[ 7] = VL_4( 7,X);
  ivY[ 8] = VL_4( 8,X);
  ivY[ 9] = VL_4( 9,X);
  ivY[10] = VL_4(10,X);
  ivY[11] = VL_4(11,X);
  ivY[12] = VL_4(12,X);
  ivY[13] = VL_4(13,X);
  ivY[14] = VL_4(14,X);
  ivY[15] = VL_4(15,X);
  /* x0y0 */
  ivZ[ 0] = VL_4( 0,Y);
  ivZ[ 1] = VL_4( 1,Y);
  ivZ[ 2] = VL_4( 2,Y);
  ivZ[ 3] = VL_4( 3,Y);
  /* x0y0z0 */
  if (doV) {
    *val = VL_4( 0,Z);                          /* f */
  }

  if (!( doD1 || doD2 ))
    return;

  /* x0y0z1 */
  if (doD1) {
    gvec[2] = D1_4( 0,Z);                       /* g_z */
  }
  if (doD2) {
    /* x0y0z2 */
    hess[8] = D2_4( 0,Z);                       /* h_zz */
  }

  /* x0y1 */
  ivZ[ 0] = D1_4( 0,Y);
  ivZ[ 1] = D1_4( 1,Y);
  ivZ[ 2] = D1_4( 2,Y);
  ivZ[ 3] = D1_4( 3,Y);
  /* x0y1z0 */
  if (doD1) {
    gvec[1] = VL_4( 0,Z);                       /* g_y */
  }
  if (doD2) {
    /* x0y1z1 */
    hess[5] = hess[7] = D1_4( 0,Z);             /* h_yz */
    /* x0y2 */
    ivZ[ 0] = D2_4( 0,Y);
    ivZ[ 1] = D2_4( 1,Y);
    ivZ[ 2] = D2_4( 2,Y);
    ivZ[ 3] = D2_4( 3,Y);
    /* x0y2z0 */
    hess[4] = VL_4( 0,Z);                       /* h_yy */
  }
  /* x1 */
  ivY[ 0] = D1_4( 0,X);
  ivY[ 1] = D1_4( 1,X);
  ivY[ 2] = D1_4( 2,X);
  ivY[ 3] = D1_4( 3,X);
  ivY[ 4] = D1_4( 4,X);
  ivY[ 5] = D1_4( 5,X);
  ivY[ 6] = D1_4( 6,X);
  ivY[ 7] = D1_4( 7,X);
  ivY[ 8] = D1_4( 8,X);
  ivY[ 9] = D1_4( 9,X);
  ivY[10] = D1_4(10,X);
  ivY[11] = D1_4(11,X);
  ivY[12] = D1_4(12,X);
  ivY[13] = D1_4(13,X);
  ivY[14] = D1_4(14,X);
  ivY[15] = D1_4(15,X);
  /* x1y0 */
```

OpenCL code (for GPUs)

```
float4 computeGradient(image3d_t sampler, float4 gradPos, const float gradOffset)
{
   //central differences gradient
   return (float4)(
                  read_imagef(sampler, linearSampler, (float4)(gradPos.x+gradOffset, gradPos.y, gradPos.z, 0.f)).x-
read_imagef(sampler, linearSampler, (float4)(gradPos.x-gradOffset, gradPos.y, gradPos.z, 0.f)).x,
                  read_imagef(sampler, linearSampler, (float4)(gradPos.x, gradPos.y+gradOffset, gradPos.z, 0.f)).x-
read_imagef(sampler, linearSampler, (float4)(gradPos.x, gradPos.y-gradOffset, gradPos.z, 0.f)).x,
                  read_imagef(sampler, linearSampler, (float4)(gradPos.x, gradPos.y, gradPos.z+gradOffset, 0.f)).x-
read_imagef(sampler, linearSampler, (float4)(gradPos.x, gradPos.y, gradPos.z-gradOffset, 0.f)).x,
   0.f);
}

float2 computeCurvature(
            image3d_t sampler,
            float4 gradPos,
            const float gradOffset
)
{
   float4 gradient = computeGradient(
                                    sampler,
                                    gradPos,
                                    gradOffset);
   float4 gradient1 = computeGradient(
                              sampler,
                              gradPos+(float4)(gradOffset,0.f,0.f,0.f) ,
                              gradOffset);
   float4 gradient2 = computeGradient(
                                    sampler,
                                    gradPos-(float4)(gradOffset,0.f,0.f,0.f) ,
                                    gradOffset);
   float4 gradient3 = computeGradient(
                                    sampler,
                                    gradPos+(float4)(0.f,gradOffset,0.f,0.f) ,
                                    gradOffset);
   float4 gradient4 = computeGradient(
                                    sampler,
                                    gradPos-(float4)(0.f,gradOffset,0.f,0.f) ,
                                    gradOffset);
   float4 gradient5 = computeGradient(
                                    sampler,
                                    gradPos+(float4)(0.f,0.f,gradOffset,0.f) ,
                                    gradOffset);
   float4 gradient6 = computeGradient(
                                    sampler,
                                    gradPos-(float4)(0.f,0.f,gradOffset,0.f) ,
                                    gradOffset);

   gradient1 = fast_normalize(gradient1);
   gradient2 = fast_normalize(gradient2);
   gradient3 = fast_normalize(gradient3);
   gradient4 = fast_normalize(gradient4);
   gradient5 = fast_normalize(gradient5);
   gradient6 = fast_normalize(gradient6);
```

Courtesy Klaus Engel, Siemens

```
   float l = fast_length(gradient);
   if (l == 0.f)
      return (float2)(0.f,0.f);
   float4 n = -gradient/l;
   float P[3][3];
   P[0][0] = 1.f-n.x*n.x;
   P[0][1] = n.x*n.y;
   P[0][2] = n.x*n.z;
   P[1][0] = n.y*n.x;
   P[1][1] = 1.f-n.y*n.y;
   P[1][2] = n.y*n.z;
   P[2][0] = n.z*n.x;
   P[2][1] = n.z*n.y;
   P[2][2] = 1.f-n.z*n.z;

   float hessian[3][3];
   hessian[0][0] = gradient1.x - gradient2.x;
   hessian[0][1] = gradient1.y - gradient2.y;
   hessian[0][2] = gradient1.z - gradient2.z;
   hessian[1][0] = gradient3.x - gradient4.x;
   hessian[1][1] = gradient3.y - gradient4.y;
   hessian[1][2] = gradient3.z - gradient4.z;
   hessian[2][0] = gradient5.x - gradient6.x;
   hessian[2][1] = gradient5.y - gradient6.y;
   hessian[2][2] = gradient5.z - gradient6.z;

   float T[3][3];
   float G[3][3];// = -P*hessian*P/l;

   T[0][0] = -P[0][0]*hessian[0][0] - P[1][0]*hessian[0][1] - P[2][0]*hessian[0][2];
   T[1][0] = -P[0][0]*hessian[1][0] - P[1][0]*hessian[1][1] - P[2][0]*hessian[1][2];
   T[2][0] = -P[0][0]*hessian[2][0] - P[1][0]*hessian[2][1] - P[2][0]*hessian[2][2];
   T[0][1] = -P[0][1]*hessian[0][0] - P[1][1]*hessian[0][1] - P[2][1]*hessian[0][2];
   T[1][1] = -P[0][1]*hessian[1][0] - P[1][1]*hessian[1][1] - P[2][1]*hessian[1][2];
   T[2][1] = -P[0][1]*hessian[2][0] - P[1][1]*hessian[2][1] - P[2][1]*hessian[2][2];
   T[0][2] = -P[0][2]*hessian[0][0] - P[1][2]*hessian[0][1] - P[2][2]*hessian[0][2];
   T[1][2] = -P[0][2]*hessian[1][0] - P[1][2]*hessian[1][1] - P[2][2]*hessian[1][2];
   T[2][2] = -P[0][2]*hessian[2][0] - P[1][2]*hessian[2][1] - P[2][2]*hessian[2][2];

   G[0][0] = (T[0][0]*P[0][0] + T[1][0]*P[0][1] + T[2][0]*P[0][2])/l;
   G[1][0] = (T[0][0]*P[1][0] + T[1][0]*P[1][1] + T[2][0]*P[1][2])/l;
   G[2][0] = (T[0][0]*P[2][0] + T[1][0]*P[2][1] + T[2][0]*P[2][2])/l;
   G[0][1] = (T[0][1]*P[0][0] + T[1][1]*P[0][1] + T[2][1]*P[0][2])/l;
   G[1][1] = (T[0][1]*P[1][0] + T[1][1]*P[1][1] + T[2][1]*P[1][2])/l;
   G[2][1] = (T[0][1]*P[2][0] + T[1][1]*P[2][1] + T[2][1]*P[2][2])/l;
   G[0][2] = (T[0][2]*P[0][0] + T[1][2]*P[0][1] + T[2][2]*P[0][2])/l;
   G[1][2] = (T[0][2]*P[1][0] + T[1][2]*P[1][1] + T[2][2]*P[1][2])/l;
   G[2][2] = (T[0][2]*P[2][0] + T[1][2]*P[2][1] + T[2][2]*P[2][2])/l;

   float t = G[0][0]+G[1][1]+G[2][2];
   float f = sqrt(G[0][0]*G[0][0]+G[0][1]*G[0][1]+G[0][2]*G[0][2]+
                              G[1][0]*G[1][0]+G[1][1]*G[1][1]+G[1][2]*G[1][2]+
                              G[2][0]*G[2][0]+G[2][1]*G[2][1]+G[2][2]*G[2][2]);

   float k1 = (t + sqrt(2.f*f*f-t*t))/2.f;
   float k2 = (t - sqrt(2.f*f*f-t*t))/2.f;

   return (float2)(k1,k2);
}
```

Time to think about new languages?

From Abelson & Sussman & Sussman *Structure and Interpretation of Computer Programs* (1985):

"First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, **programs must be written for people to read, and only incidentally for machines to execute**."

Time to think about new languages?

From Knuth *Literate Programming* (1992):

"Let us change our traditional attitude to the construction of programs: instead of imagining that our main task is to instruct a computer what to do, **let us concentrate rather on explaining to humans what we want the computer to do**."

Today as well, people are deciding we need to build new languages...

DSLs and related work

- "DSL" = Domain-specific language
  - C/C++: fast & general (not easy)
  - Python, other HLLs: easy & general (not fast)
  - DSLs: easy & fast (not general)
- M. Bostock, V. Ogievetsky & J. Heer: *Protovis* & *D^3* for web-based info-vis (2009, 2011)
- K. J. Brown et al.: *Delite* framework for portably parallel DSLs (2011)
  - *OptiML* for machine learning
  - *Liszt* for solving PDEs on meshes
- J. Ragan-Kelley et al.: *Halide* for computational photography image processing (2012)

Outline

- Context & Motivation
- **Language design**
- Example programs
- Looking forward

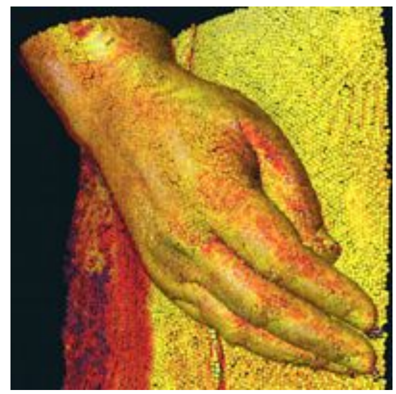
Diderot

http://diderot-language.cs.uchicago.edu

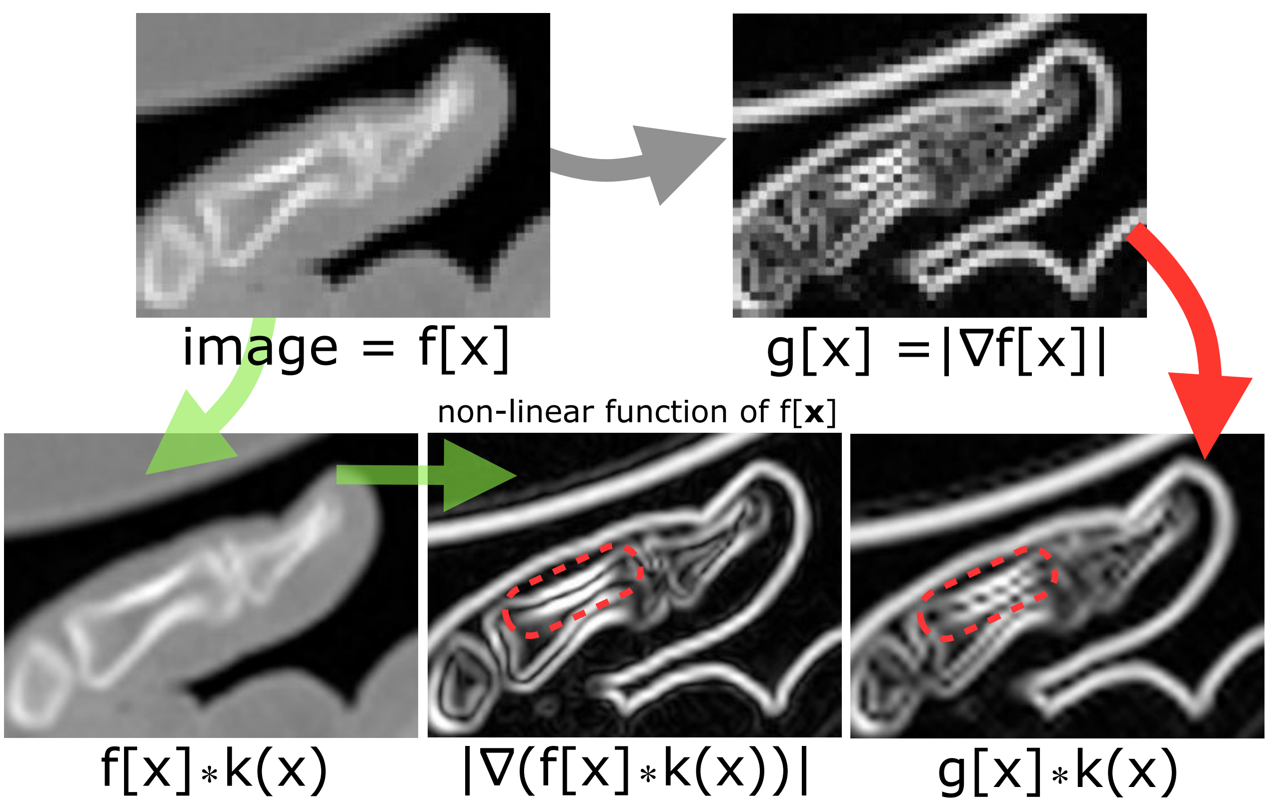
- Domain-Specific Language for portably parallel analysis and visualization of continuous fields (scalar/vector/tensor)
- Gain **programmer efficiency** and **parallel performance** at cost of algorithmic **generality**
- **Portably** parallel: compiles to multi-core CPUs (pthreads), GPUs (OpenCL)
- High-level notation supports rapid development and mathematically legible code (“from whiteboard to executable”)

What is Diderot best at?

- Algorithms with large number of (mostly) independent computations based on local properties of continuous fields, e.g.
  - Direct Volume Rendering
  - Streamlines, Fiber Tractography
  - Particle Systems for Image Feature Sampling

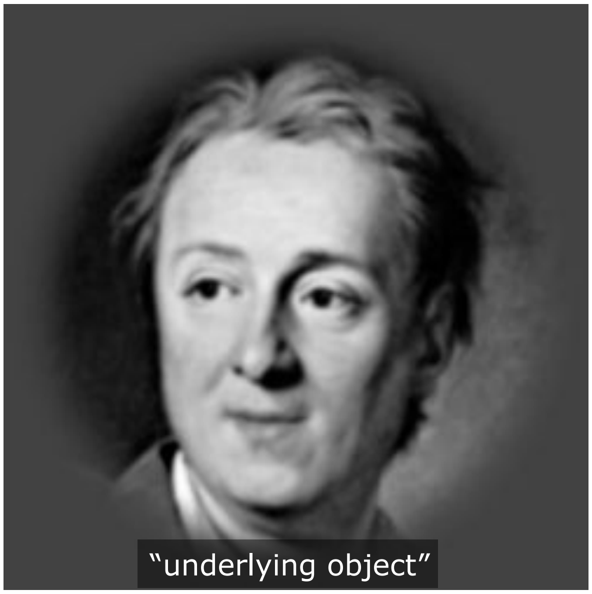


Continuous fields ≠ discrete images
(Matlab, Numpy good for discrete images)



Objects versus **images**

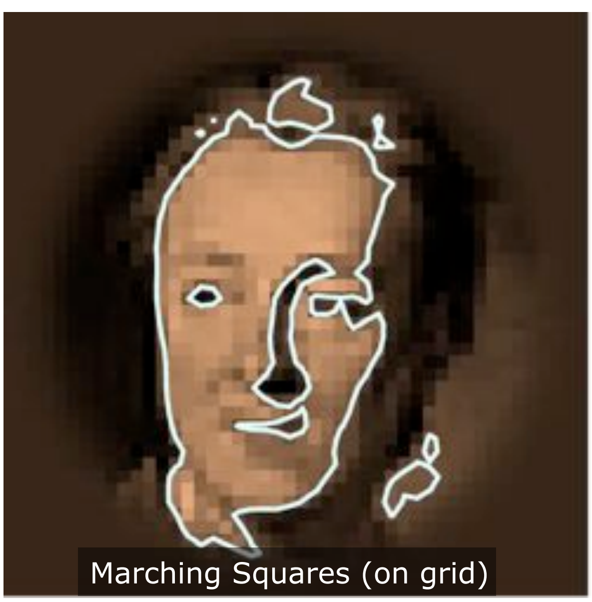
- Measurements of objects produce **images**

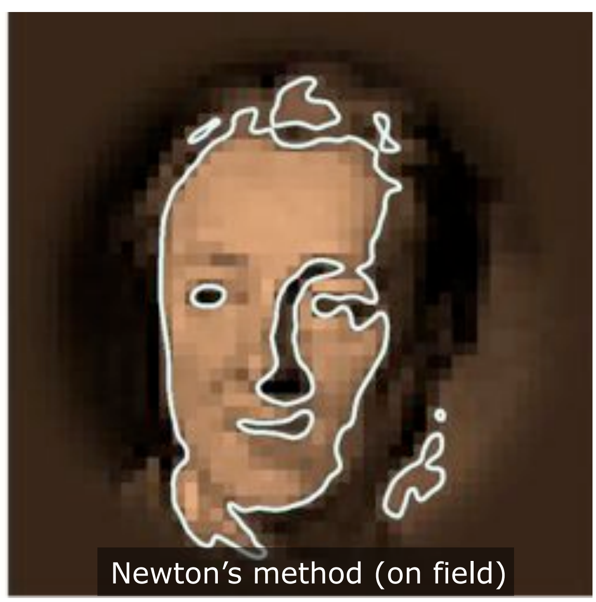


- Goal of scientific vis & analysis is to make statements about the underlying **objects**

Objects versus **images**

- Grid orientation/spacing is property of **image**





- Continuous fields (in Diderot) help get away from grid details towards object properties

Objects versus **images**

Previous work from 1928:

La trahison des images, Magritte

Minimal example

Square roots of numbers 1..1000 by Heron's method

```
// Global definitions
input int N = 1000;
input real eps = 0.000001;
// Strand definition
strand sqroot(real val) {
   output real root = val;
   update {
      root = (root + val/root)/2.0;
      if (|root^2 - val|/val < eps) {
         stabilize;
      }
   }
}
// Strand initialization
initially [ sqroot(real(i)) | i in 1..N ];
```

Globals are immutable; used for program inputs

Strands are bulk synchronous

Input parameters for initialization

Strand state, including output

Update method implements algorithm

Initialization of collection of strands with comprehension notation

Diderot program structure

- Computation decomposed into collection of mostly autonomous *strands*
- Each strand has state and an **`update`** method
- **`update`** implements one iteration of algorithm
  - strands can **`stabilize`**, **`die`**, **`new`**
- Abstractions:
  - Fields: convolution & differentiation of discrete data
  - Parallel computation (CPU vs GPU)
  - Strand communication

Execution model

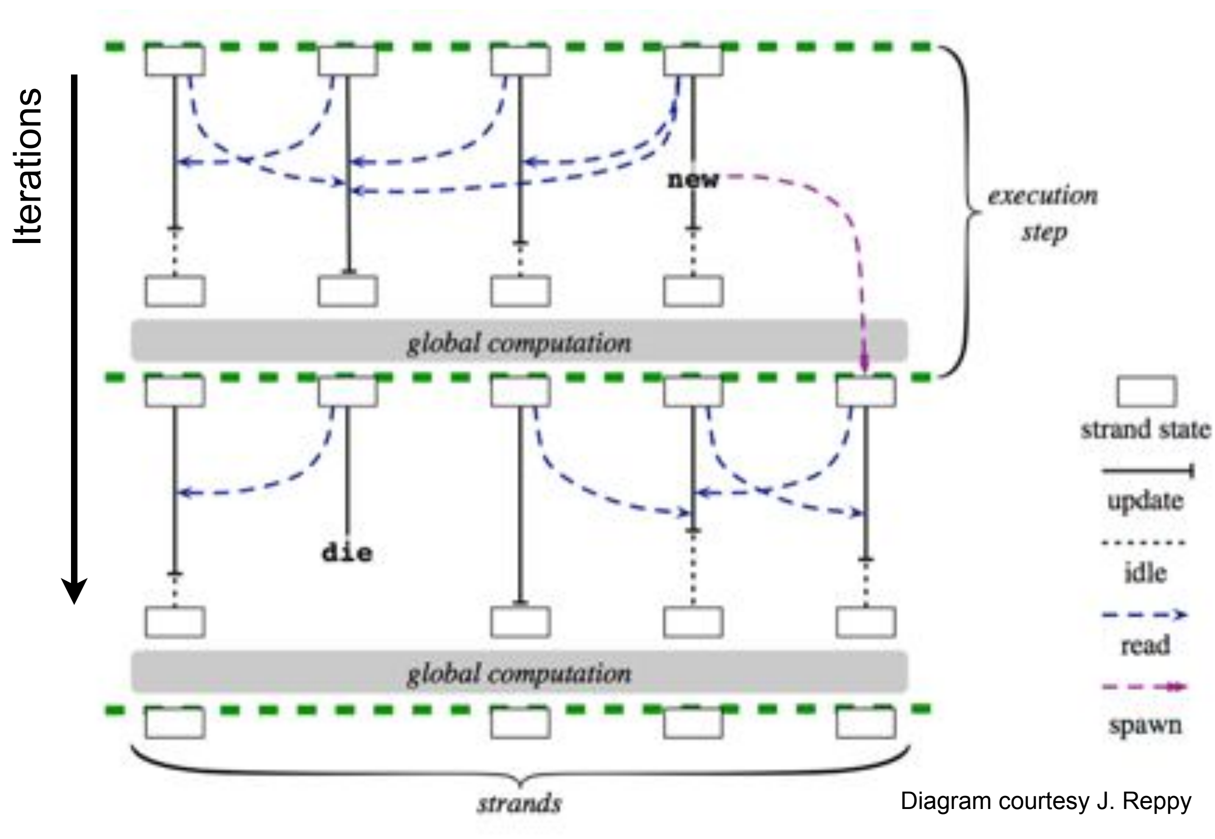
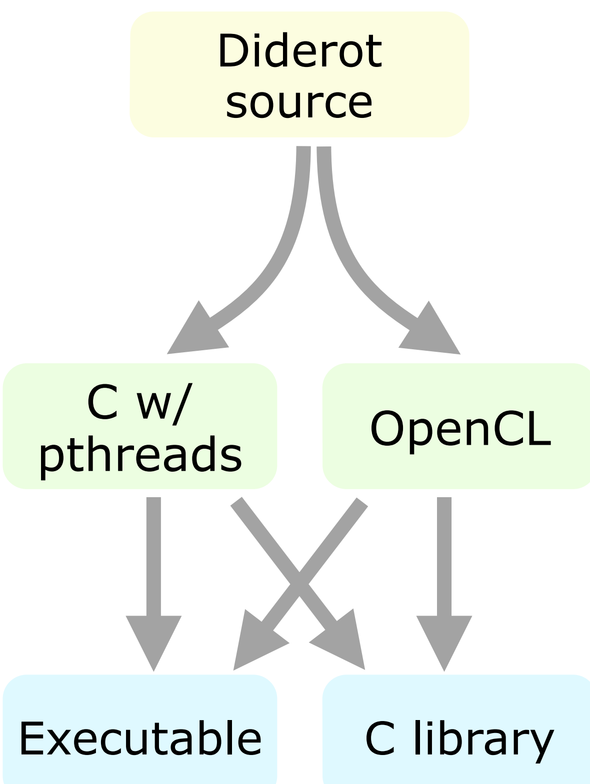


Diagram courtesy J. Reppy

Example: sampling isosurfaces

```
input real isoval = 0.4;
field#1(2)[] F = ctmr ⊛ image("ddro.nrrd") - isoval;
int grid = 150;
int stepsMax = 10;
real epsilon = 0.000001;
strand FindZero(vec2 x0) {
  output vec2 x = x0;
  int steps = 0;
  update {
    if (!inside(x, F) || steps > stepsMax)
      die;    // Stop outside domain or after many steps
    if (|∇F(x)| == 0)
      die;    // Can't proceed with zero derivative
    // the Newton-Raphson step
    vec2 dx = normalize(∇F(x)) * F(x)/|∇F(x)|;
    x -= dx;
    if (|dx| < epsilon)
      stabilize; // Converged when step small enough
    steps += 1;
  }
}
initially { FindZero([lerp(0, 1, -0.5, ui, grid-0.5),
                      lerp(0, 1, -0.5, vi, grid-0.5)])
            | vi in 0..(grid-1), ui in 0..(grid-1) };
```

Compilation



- Compiler written in SML/NJ
- Three stages of intermediate representation

- Use **cc** to create executable (with command-line interface) or C library (with API)

Some technical details

- Type system can capture abstractions

```
field#2(3)[3] F =  bspln3 ⊛ load("vecs.nrrd");
field#1(3)[3,3] G = ∇F;
```

  - ⊛=separable convolution; #K: order of continuity
  - $[d_1, d_2, \ldots]$: shape of individual tensor samples
- Expose optimization opportunities from whole-program analysis and vector calc
  - many common sub-expressions for separable convolution: F(x), ∇F(x), and ∇⊗∇F(x)
  - vector, tensor-valued expression simplification:

```
n = u × v; P = identity[3,3] - n⊗n;
x = Pu   ⇒   x = u
```

Some technical details

- Tensor expression simplification/ optimization based on Einstein Notation

Optimizing tensor operations

Slide courtesy J. Reppy

Consider the expression trace(a⊗b).

This Diderot expression is represented in the compiler as

$$\textbf{let } M = (\lambda(u,v).\langle u_i v_j \rangle_{ij})(a,b)$$
$$\textbf{let } t = (\lambda X.\langle X_{kk} \rangle)(M)$$
$$\textbf{in } t$$

substitution of the definition of M for X yields

$$\textbf{let } t = (\lambda(u,v).\langle u_k v_k \rangle)(a,b)$$
$$\textbf{in } t$$

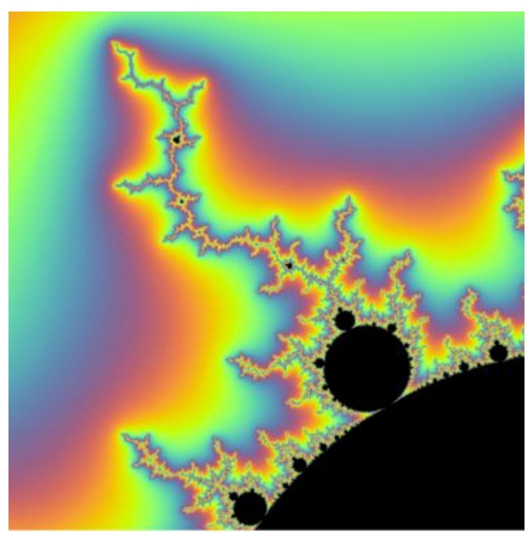
Replaces a rewrite rule: $\text{Trace}(\text{Outer}(u,v)) \Rightarrow \text{Dot}(u,v)$.

Outline

- Context & Motivation
- Language design
- **Example programs**
- Looking forward

Mandelbrot set

```
// Global definitions
input int reso = 2000;
input real escape = 4.0;
input int maxiter = 1000;
input vec2 center = [0,0];
input real fov = 2;
field#0(1)[3] cmap = tent ⊛ image("colormap.nrrd");
// Strand definition
strand mandel(vec2 c) {
   vec2 z = c;
   int iter = 0;
   output vec3 rgb = [0, 0, 0];
   update {
      // z = z^2 + c
      z = [z[0]^2 - z[1]^2, 2*z[0]*z[1]] + c;
      if (|z| > escape) {
         // point escaped; color based on iter and |z|
         real time = iter - log2(log(|z|)/log(escape));
         rgb = cmap(fmod(log(time), 1));
         stabilize;
      }
      iter += 1;
      if (iter > maxiter) {
         rgb = [0, 0, 0];
         stabilize;
      }
   }
}
```



```
// Strand initialization
initially [ mandel([lerp(center[0]-fov, center[0]+fov,
                         1, realIdx, reso),
                    lerp(center[1]-fov, center[1]+fov,
                         1, compIdx, reso)])
            | compIdx in 1..reso, realIdx in 1..reso ];
```

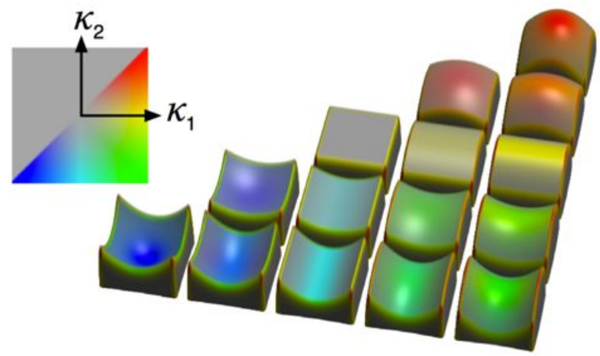
Example: curvature measurement

```
// volume dataset
field#2(3)[] F = bspln3 ⊛ load("quads.img");

// RGB colormap of (kappa1,kapp2)
field#0(2)[3] RGB = tent ⊛ load("rgb.img");

...

vec3 grad = -∇F(pos);
vec3 norm = normalize(grad);
// begin curvature computation
tensor[3,3] H = ∇⊗∇F(pos);
tensor[3,3] P = identity[3] - norm⊗norm;
tensor[3,3] G = -(P·H·P)/|grad|;
real disc = max(0.0, sqrt(2.0*|G|^2 - trace(G)^2));
real k1 = (trace(G) + disc)/2.0;
real k2 = (trace(G) - disc)/2.0;
// find material RGBA
vec3 matRGB = RGB([clamp(-1.0, 1.0, 6.0*k1),
                   clamp(-1.0, 1.0, 6.0*k2)]);
```



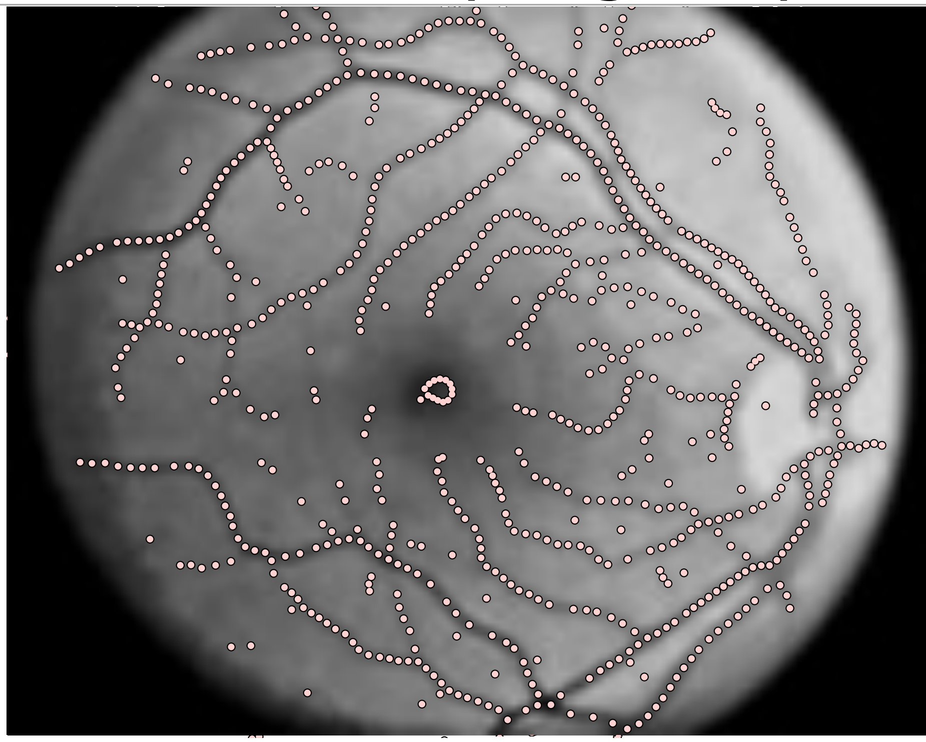
Direct (coordinate-free) notation encourages and basis-independent code (eventually, dimension-independent code)

Finding valley lines

```
strand valleyline(vec3 initpos) {
   output vec3 x = initpos;
   update {
      vec3{3} ev = evecs(∇⊗∇F(x));
      vec3 dir = normalize((ev{3}⊗ev{3}
                            + ev{2}⊗ev{2})•∇F(x));
      real fdd = ∇F(x)•dir;
      real sdd = dir•∇⊗∇F(x)•dir;
      vec3 delta = dir*fdd/sdd; // Newton Optimization
      if (|delta| < epsilon) {
         stabilize;
      }
      x -= delta;
   }
}
```

Lung airways (chest CT)

Blood vessel sampling w/ particles



Neighboring particles repel each other with potential function (using strand communication)

Diffusion Tensor LIC

Diffusion Tensor LIC code

```
int sizeX = 776;
int sizeY = 664;
real hh = 0.03;         // step size of integration
int stepNum = 110;      // steps taken up or downstream
real stdv = 2*sqrt(1.0/stepNum);
real anisoMin = 0.01;  // stop on streamline path
field#0(2)[] R = tent ⊛ image("rand.nrrd");
field#0(2)[3,3] T = tent ⊛ image("ten3d.nrrd");
function real cl(vec2 x) {  // Westin '99
  real{3} lam = evals(T(x));
  return (lam{0} - lam{1})/lam{0};
}
function real contrast(real ani)
  = clamp(0,1, lerp(0,1, anisoMin, ani, 1));
function vec2 proj(vec3 v) = [v[0],v[2]];
function vec2 dir(vec2 ref, vec2 x) {
    vec2 ev = proj(evecs(T(x)){0});
    return ev if (ev ref > 0) else -ev;
}
```

```
strand hlic(vec2 x0, real sign) {
  vec2 prev = hh*sign*proj(evecs(T(x0)){0});
  vec2 x = x0;
  output vec3 rgb = [0,0,0];
  real sum = 0;
  vec2 step = [0,0];
  int stepIdx = 0;
  update {
    step = hh*dir(prev, x + 0.5*hh*dir(prev, x));
    x += step;
    if (stepIdx == stepNum || !inside(x, R)
        || cl(x) < anisoMin)
      stabilize;
    sum += R(x);
    stepIdx += 1;
    prev = step;
  }
  stabilize {
    sum *= contrast(cl(x0))/stepNum;
    real gray = clamp(0,1,lerp(0, 1, -stdv, sum, stdv));
    vec3 v = evecs(T(x0)){0};
    rgb = gray*lerp([1,1,1],[|v[0]|,|v[1]|,|v[2]|],cl(x0));
  }
}
initially [ hlic([lerp(-48, 48, -0.5, xi, sizeX-0.5),
                  lerp(-41, 41, -0.5, yi, sizeY-0.5)],
                 lerp(-1, 1, 0, si, 1))
            | yi in 0..(sizeY-1), xi in 0..(sizeX-1),
              si in 0..1 ];
```

Outline

- Context & Motivation
- Language design
- Example programs
- Looking forward

Much to do ...

- More natural field definitions:
  - **lifting**: field#1(3)[3] V = ctmr ⊛ image(vec.nrrd) → field#1(3)[] M = |V|
  - **composition**: V = W ∘ F
  - Differentiation possible, not implemented
  - Fields from point clouds, FEMs
- Many possibilities for GUI / IDE:
  - Sliders for all `input` variables
  - Simplify unicode input
  - Nicer compiler error messages
- More parallel targets: MPI, CUDA
- Virtual memory for big datasets

Summary

- Harder to extract knowledge from scientific imaging datasets
- Parallel computing platforms getting more complicated
- Diderot is (ambitious) work-in-progress to help build new scientific tools
- Is open-source (and undocumented!)
  - http://diderot-language.cs.uchicago.edu/