Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis [Chiw-PLDI-2012] [Kindlmann-VIS-2015]

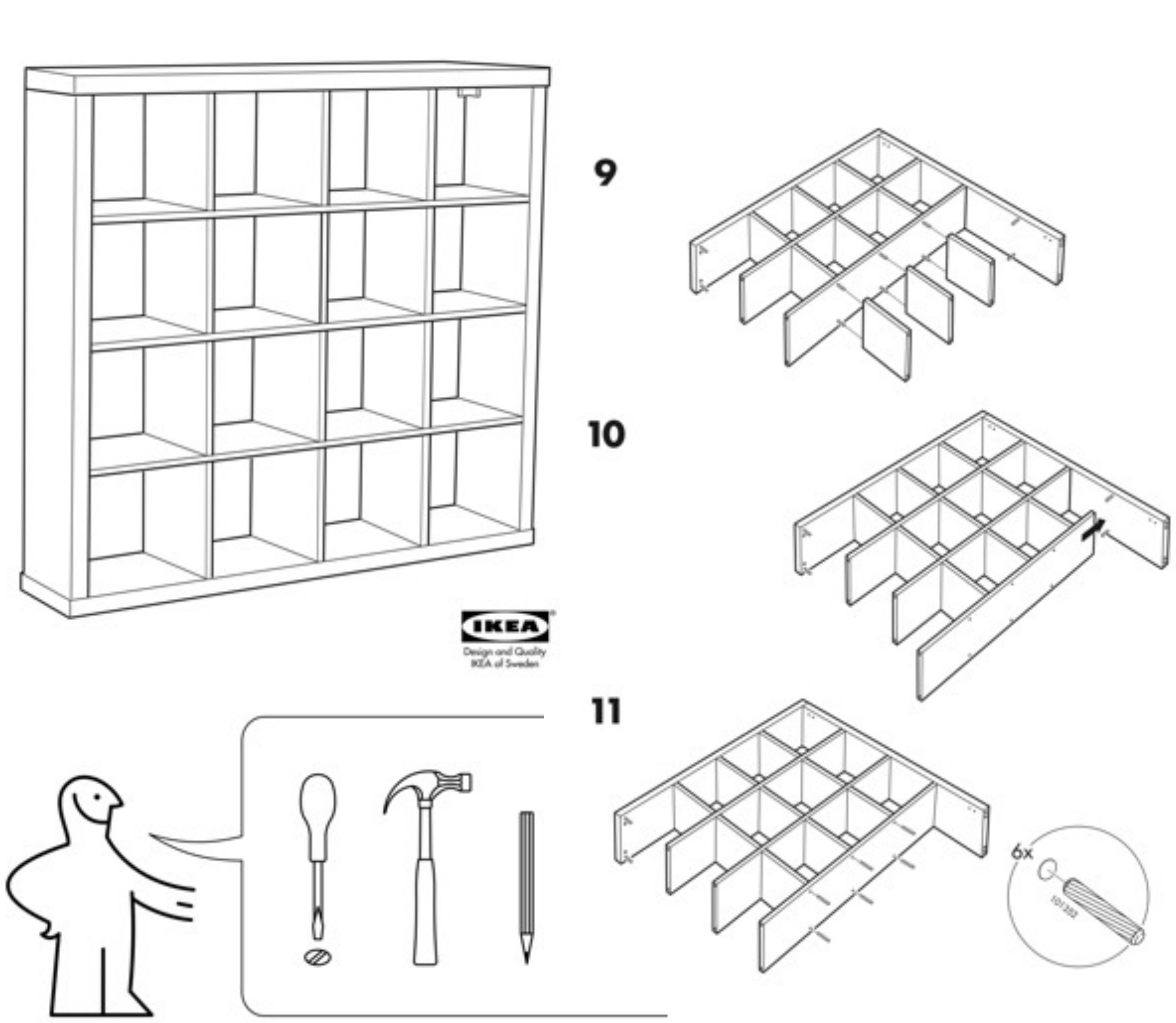
Please interrupt me with questions!

Gordon Kindlmann
GLK@uchicago.edu

20 Jan 2016 Geilo Winter School

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

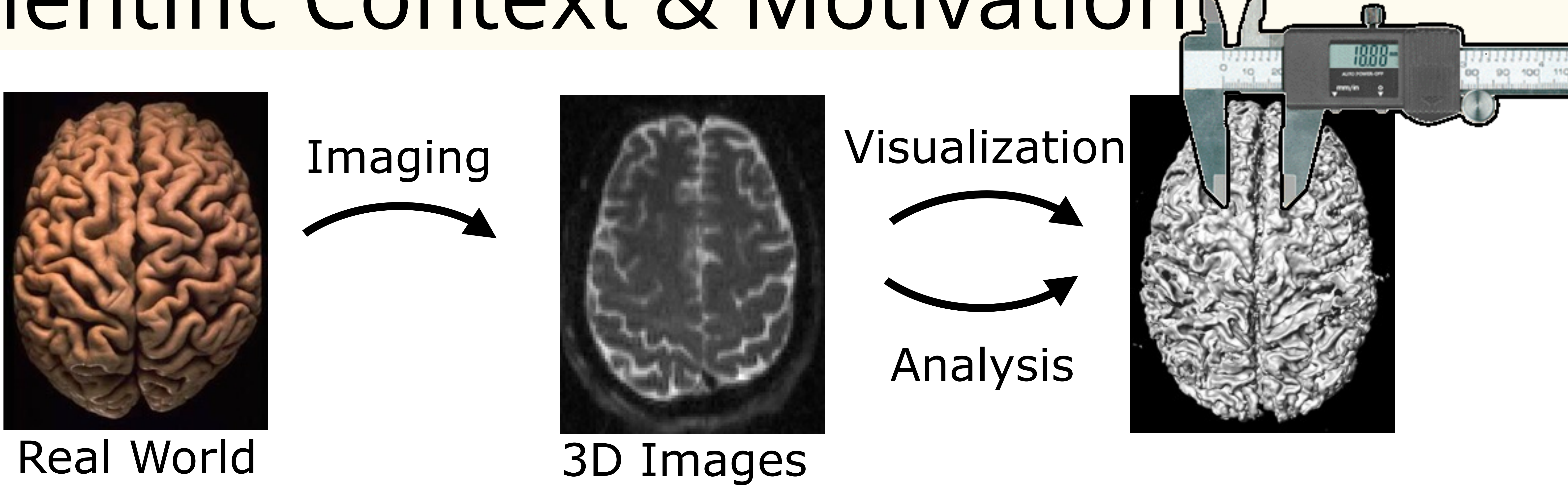
Warning: Diderot is not like Paraview



Paraview

Diderot

Scientific Context & Motivation



- Scientists need software to show and measure structure in large complex image datasets
- **Creating new visualization/analysis tools is an essential part of the scientific process**

Creating vis/analysis tools is hard to do

Increasing range of:

Imaging **modalities**

Imaging **applications**

Vis & analysis **algorithms**

Scientists need to **rapidly** implement variety of new programs

Goal: speed the development of portable parallel methods of 3D scientific visualization and analysis

Programmers want **portable** parallel languages

Increasing **data size**

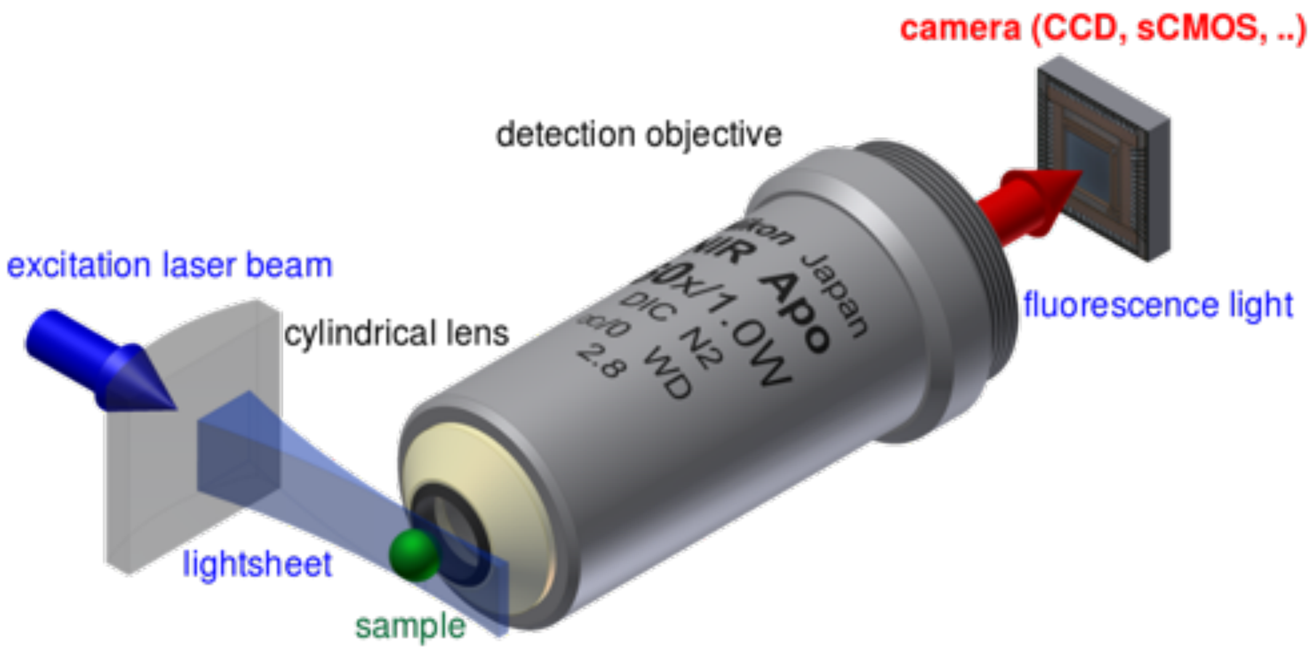
Need **parallel** computing

Rapidly shifting parallel computing architectures

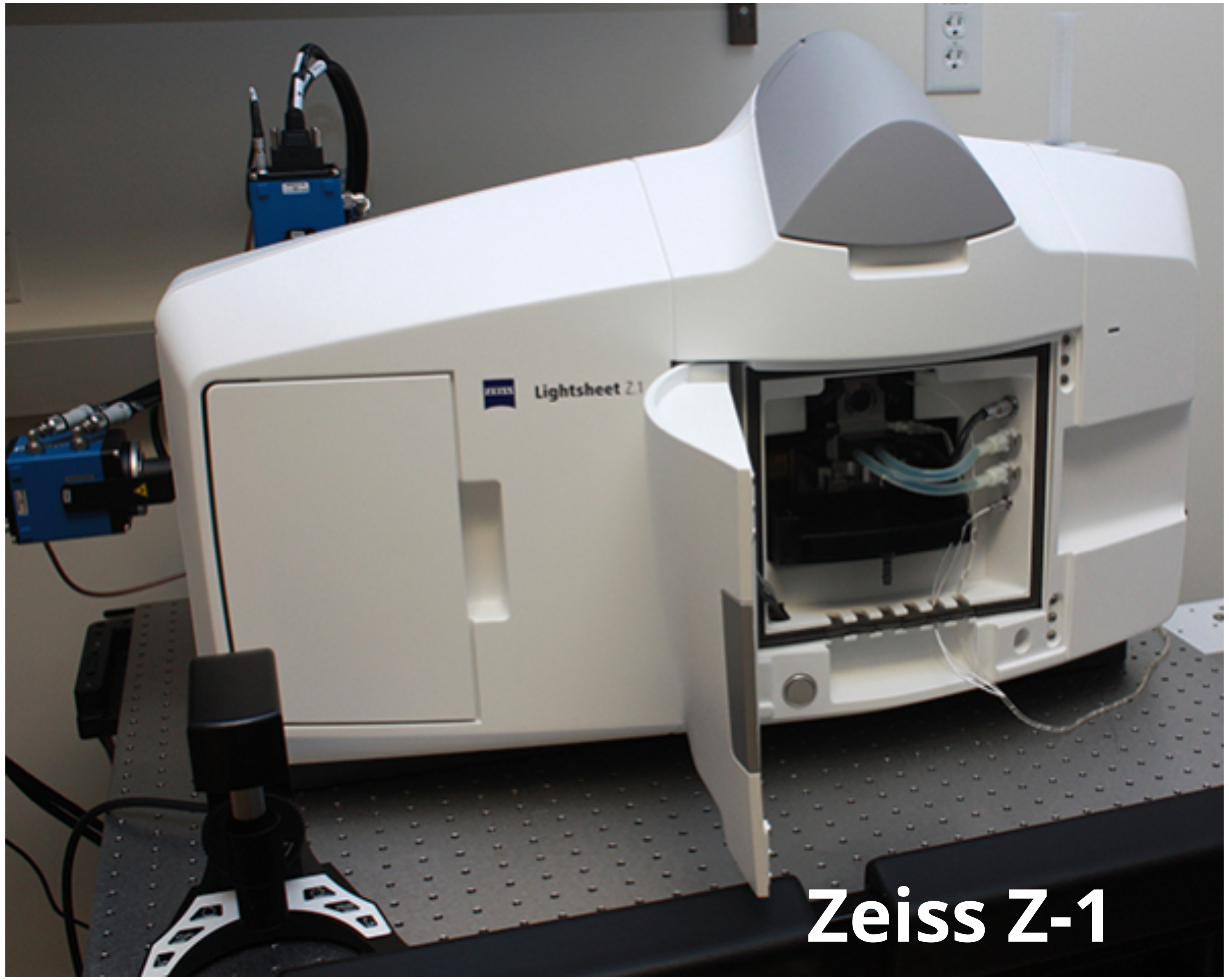
Torrent of new data from microscopy

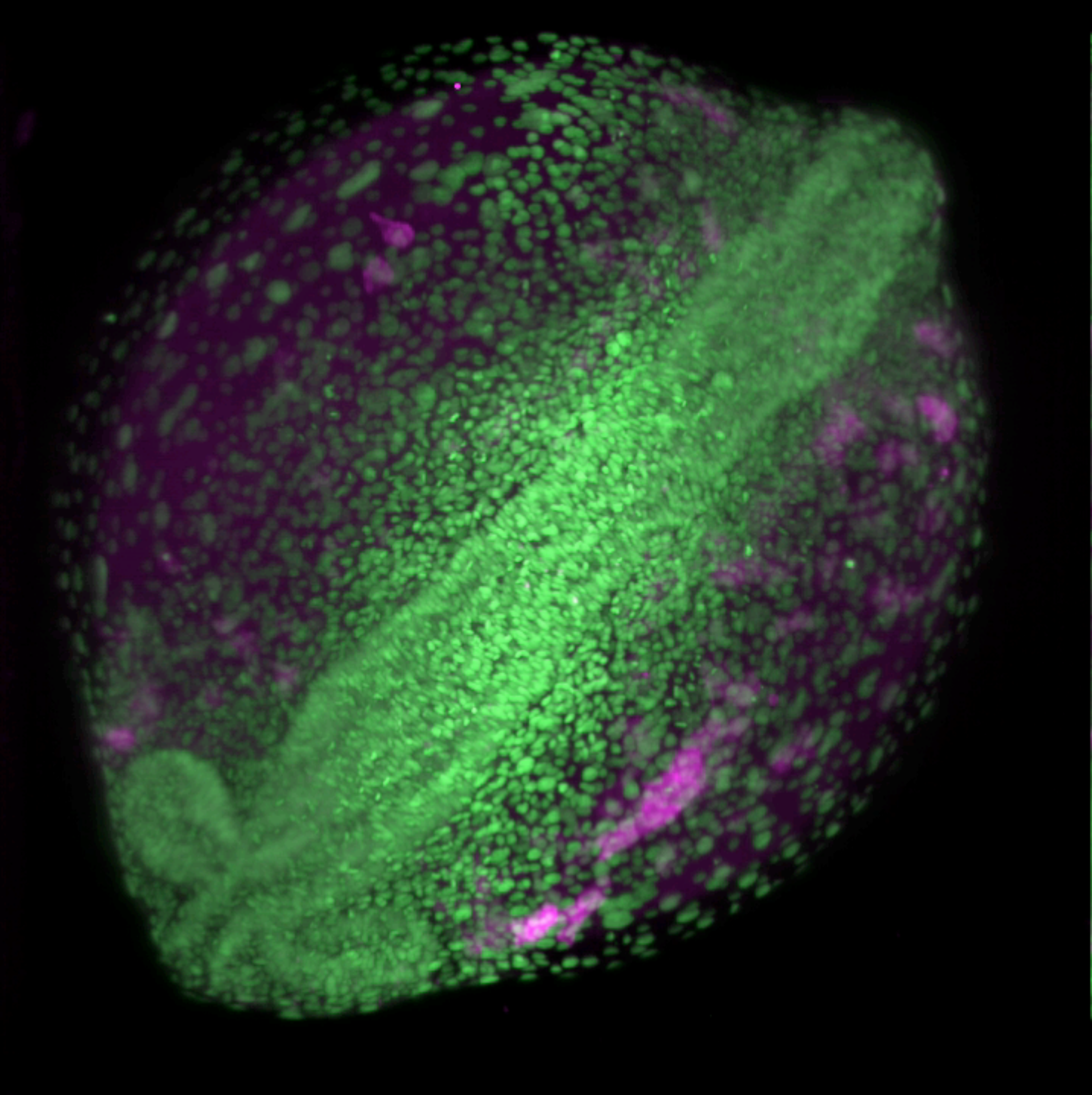
Digital Light Sheet Microscopy

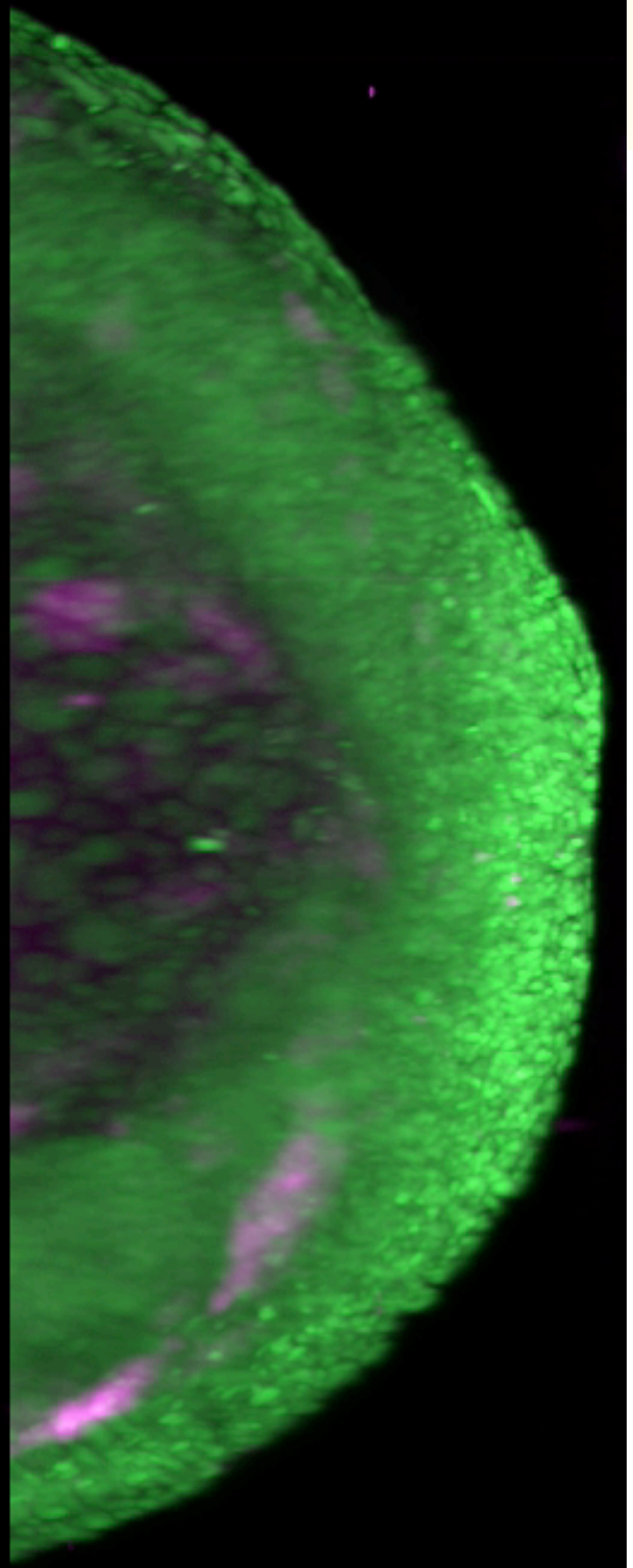
https://en.wikipedia.org/wiki/Light_sheet_fluorescence_microscopy



- Compared to confocal microscopy:
  - Fewer photons to get same image quality
  - Less phototoxicity, photobleaching
  - More data: **~5 gigabytes / minute**
    $\Rightarrow$ **~7 terabytes / day**



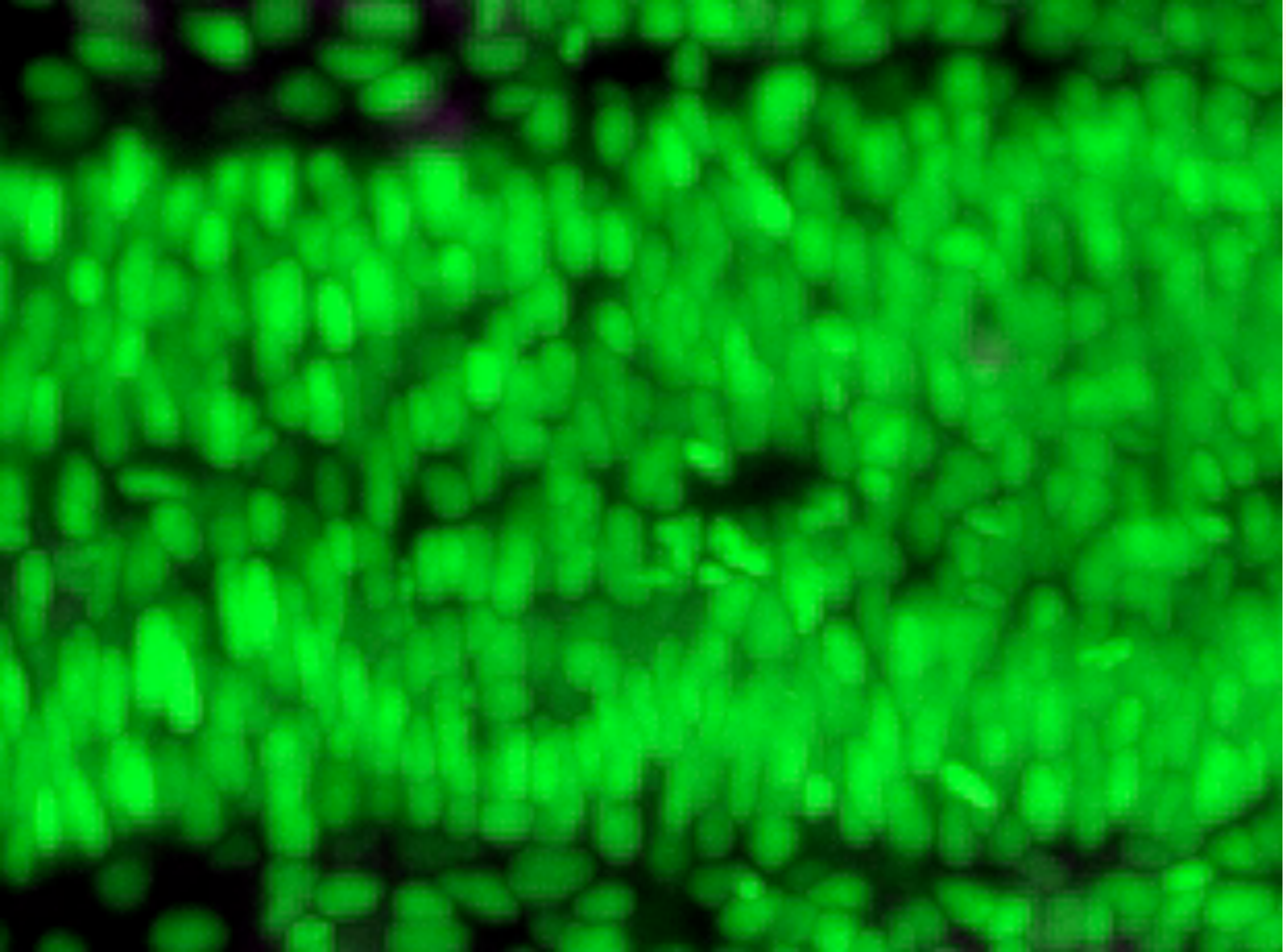




Example Data

Prof. Victoria Prince (Dept. Organismal Biology and Anatomy, University of Chicago), Anastasia Beiriger

- Where do pioneer neurons come from; where do they go; how do they find their way?
- Pioneer neurons are first to traverse path of nerve
- Facial branchiomotor neuron (FBMN) of zebrafish, pioneer migration starts ~16 hpf



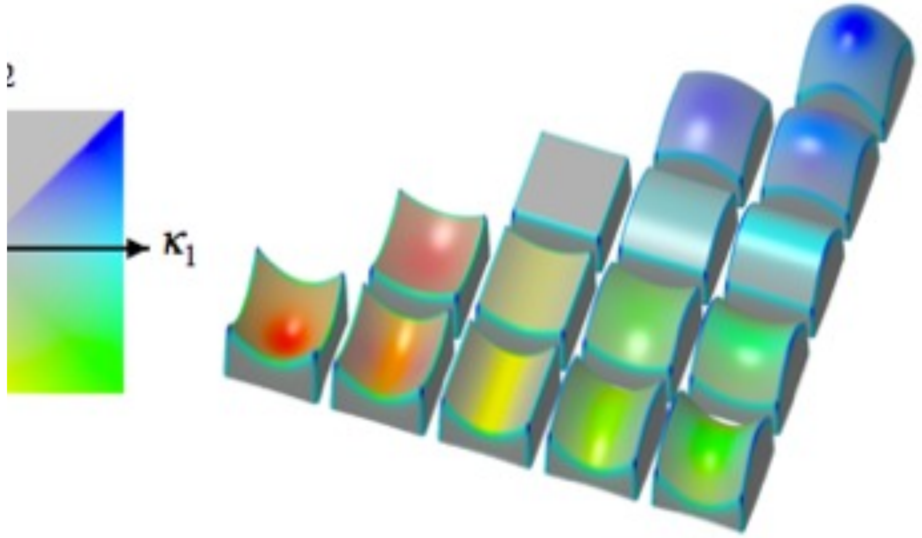
Example Data

Prof. Victoria Prince (Dept. Organismal Biology and Anatomy, University of Chicago), Anastasia Beiriger

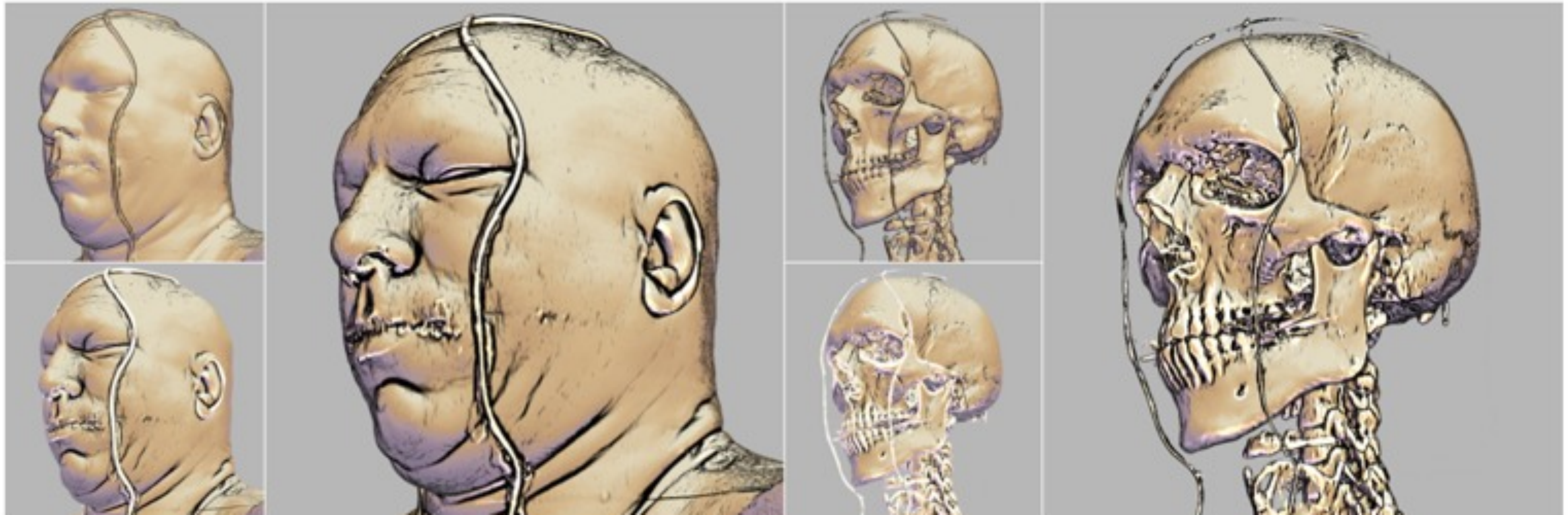
- Where do pioneer neurons come from; where do they go; how do they find their way?
- Pioneer neurons are first to traverse path of nerve
- Facial branchiomotor neuron (FBMN) of zebrafish, pioneer migration starts ~16 hpf

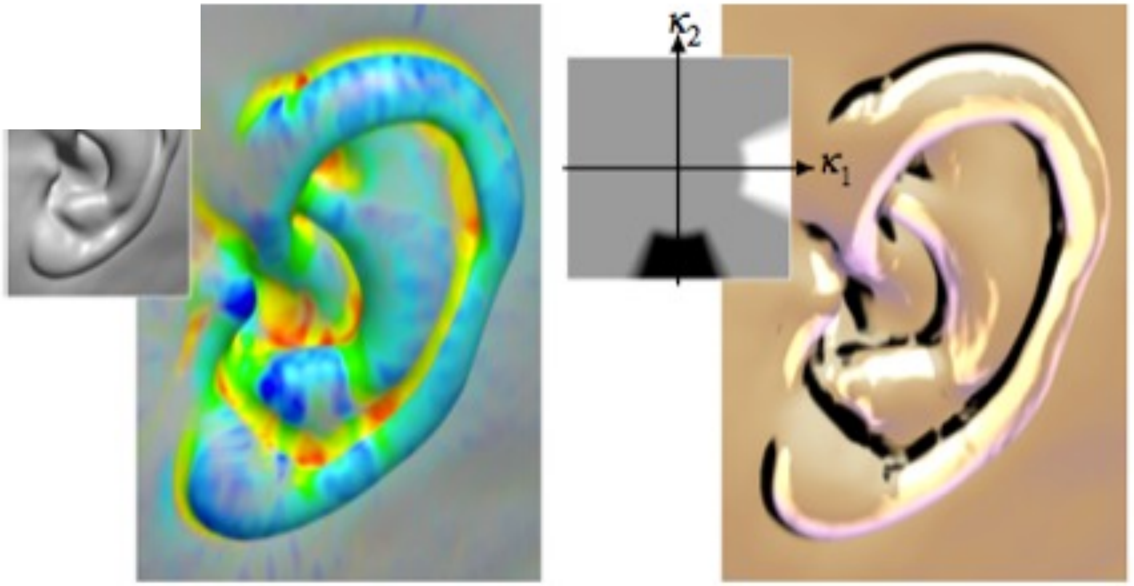
Example visualization method [Kindlmann-VIS-2003]

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



e rendered diagram of (κ_1, κ_2) space. The colors in the ansfer function domain are mapped onto the patches with ling 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

```
#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);

  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);
}
```

Courtesy Klaus Engel, Siemens

Time to think about new languages?

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

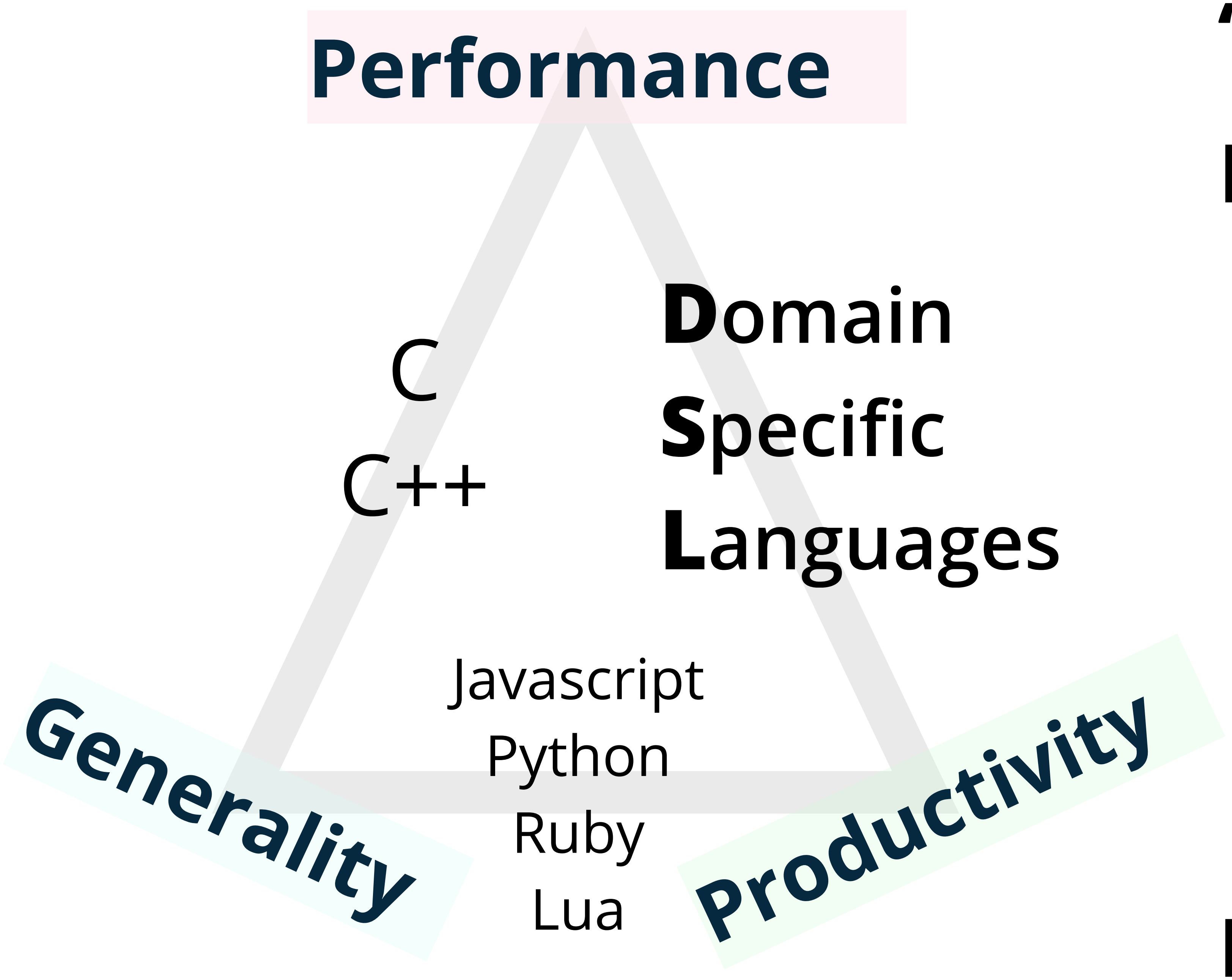
“First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, **programs must be written for people to read, and only incidentally for machines to execute**.”

Time to think about new languages?

From Donald 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**.”

Triangle of language strengths (courtesy Pat Hanrahan)



“Why not write a library?”

DSL advantages:

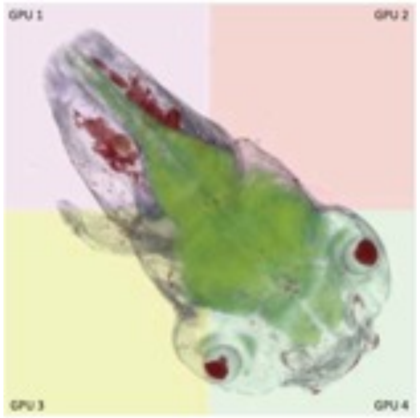
1. Code can be concise, idiomatic (types, syntax, operations)
2. Compiler analysis, optimizations
3. Express parallel execution apart from OS, hardware (CPU/GPU)

Expert C/C++ coders like libraries

Goal: Open up Sci Vis research to a larger user community

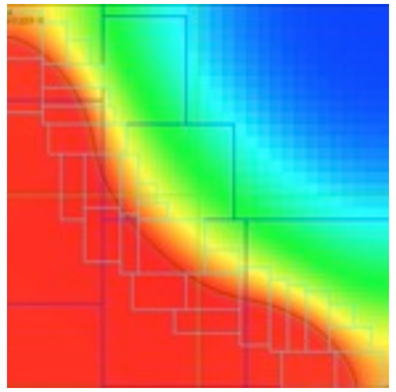
Related DSL research

- **Vivaldi** [Choi-VIS-2014]: Volume rendering, processing in Python-like DSL on distributed GPU clusters



- **ViSlang** [Rautek-VIS-2014]: Slangs (procedural, declarative, functional) interactively combined

- **Scout** [McCormick-VIS-2004] [McCormick-JPC-2007] [Jablin-IPDPS-2011] [McCormick-WOLFHPC-2014]: compile data- or task-parallel programs on grids, using LLVM toolchain



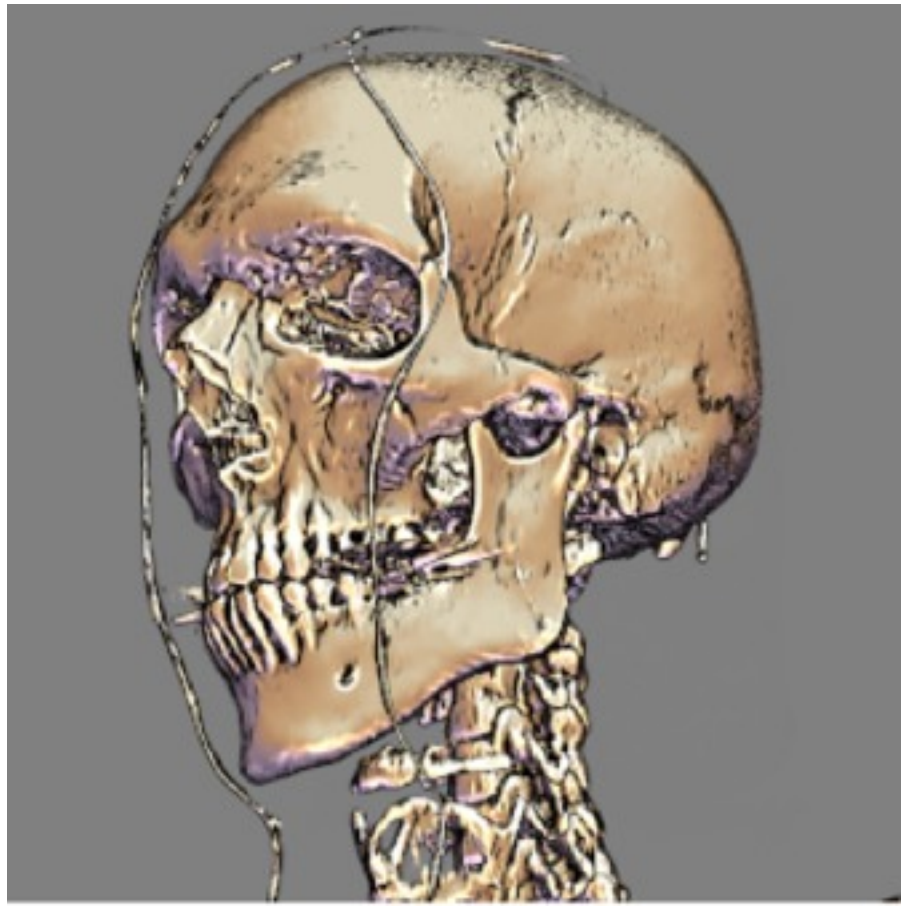
- Other DSLs discussed in paper
- Diderot's strength: **idiomatic mathematical abstractions**

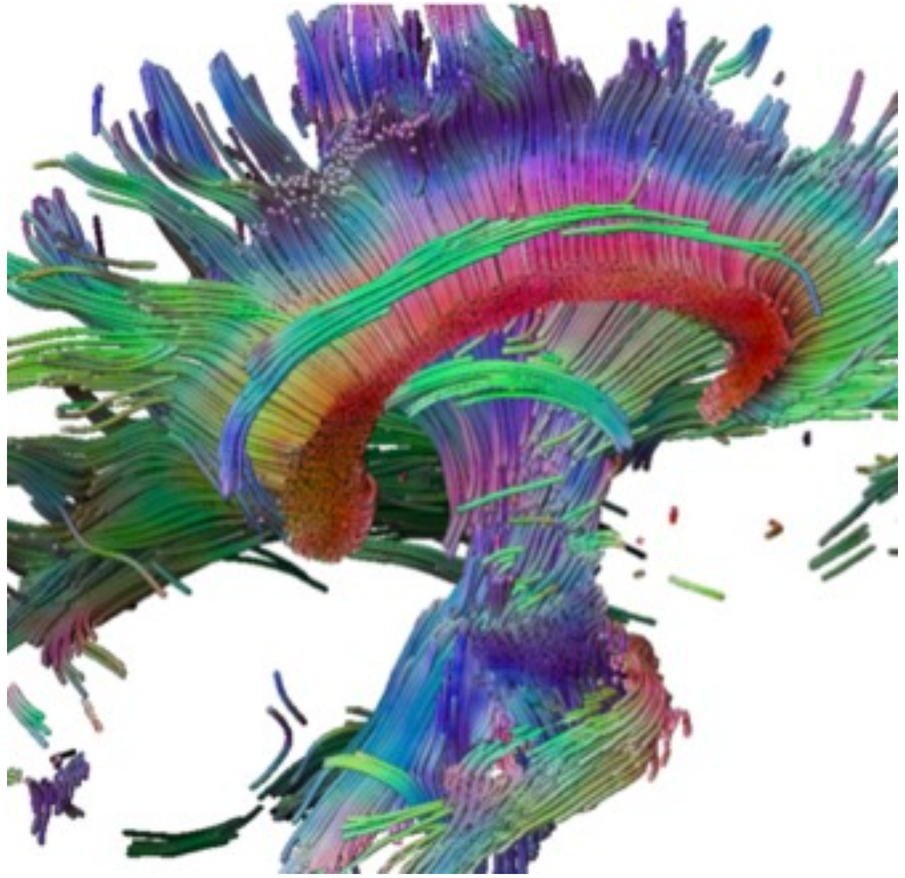
What is Diderot best at?

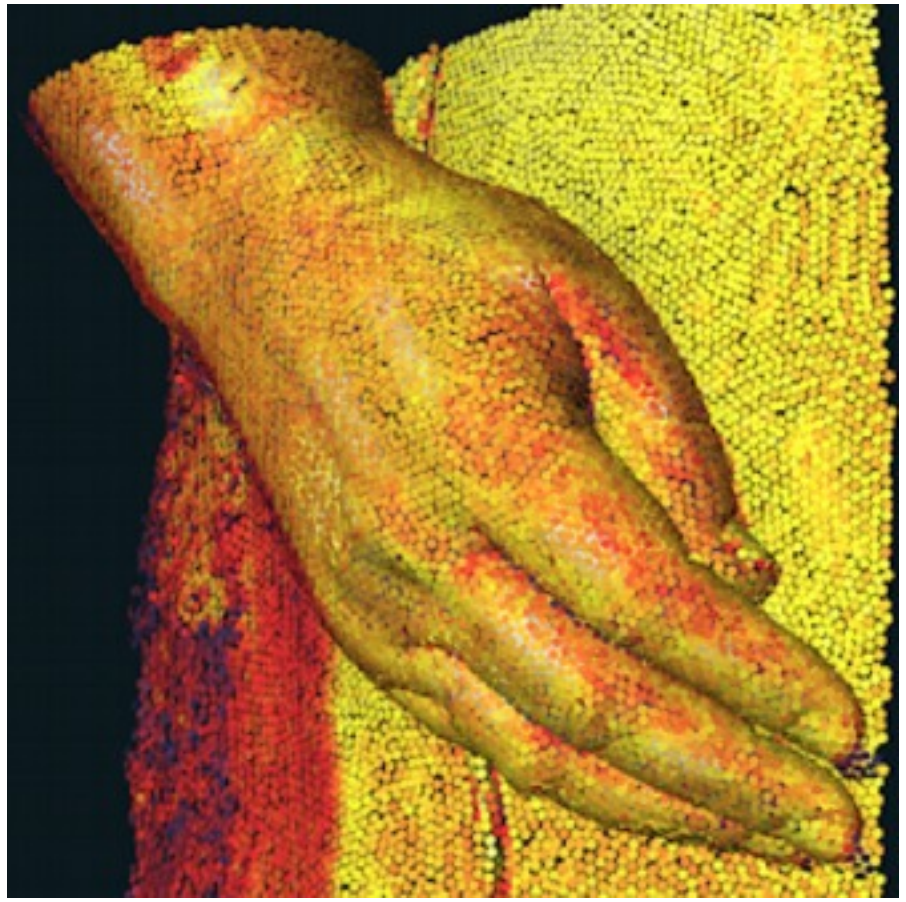
Algorithms with large number of (mostly) independent computations based on local properties of continuous fields:

- Direct Volume Rendering
- Streamlines, Fiber Tractography
- Particle Systems for Image Feature Sampling

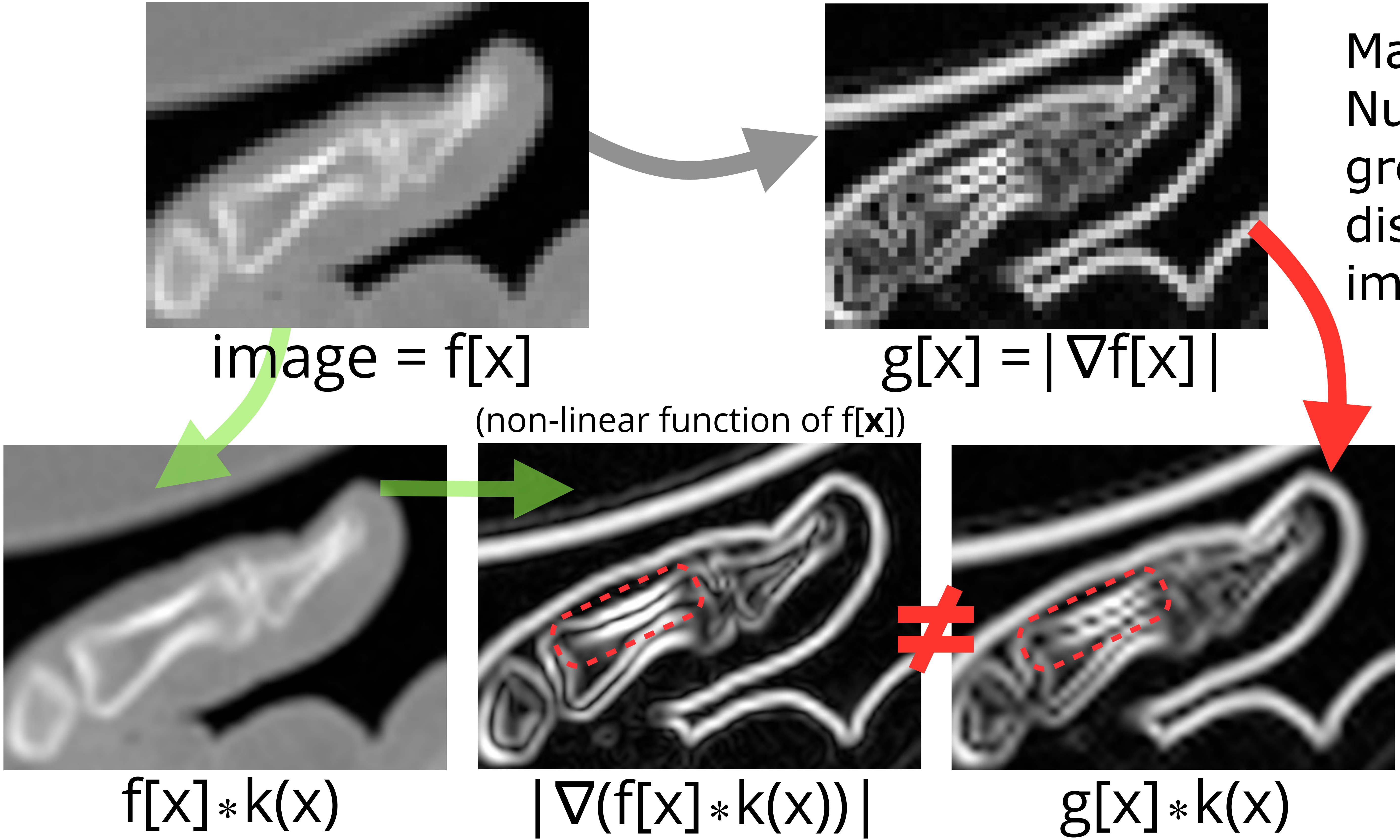
i.e.: an important part of traditional "sci vis" research







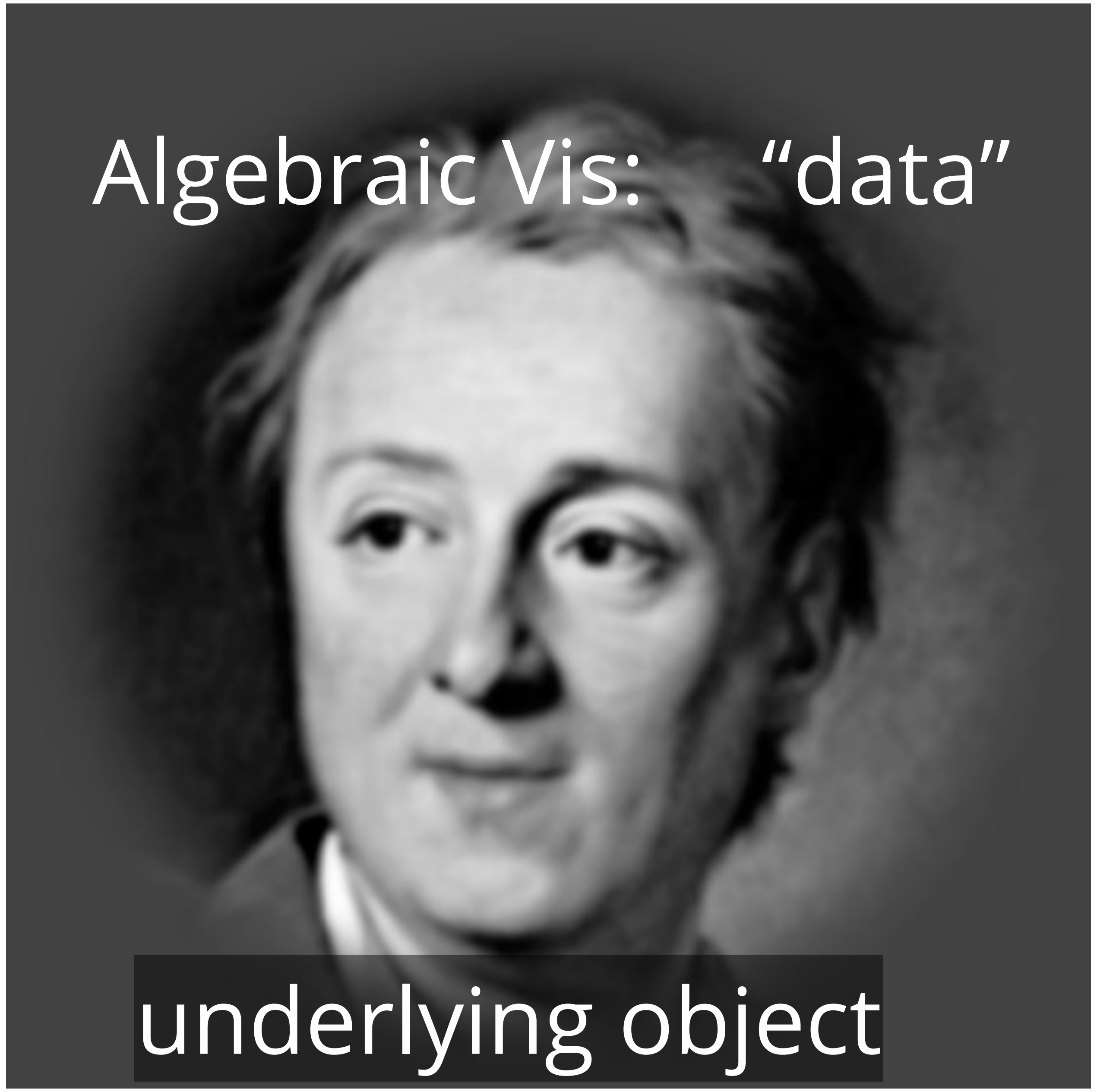
Continuous fields ≠ discrete images

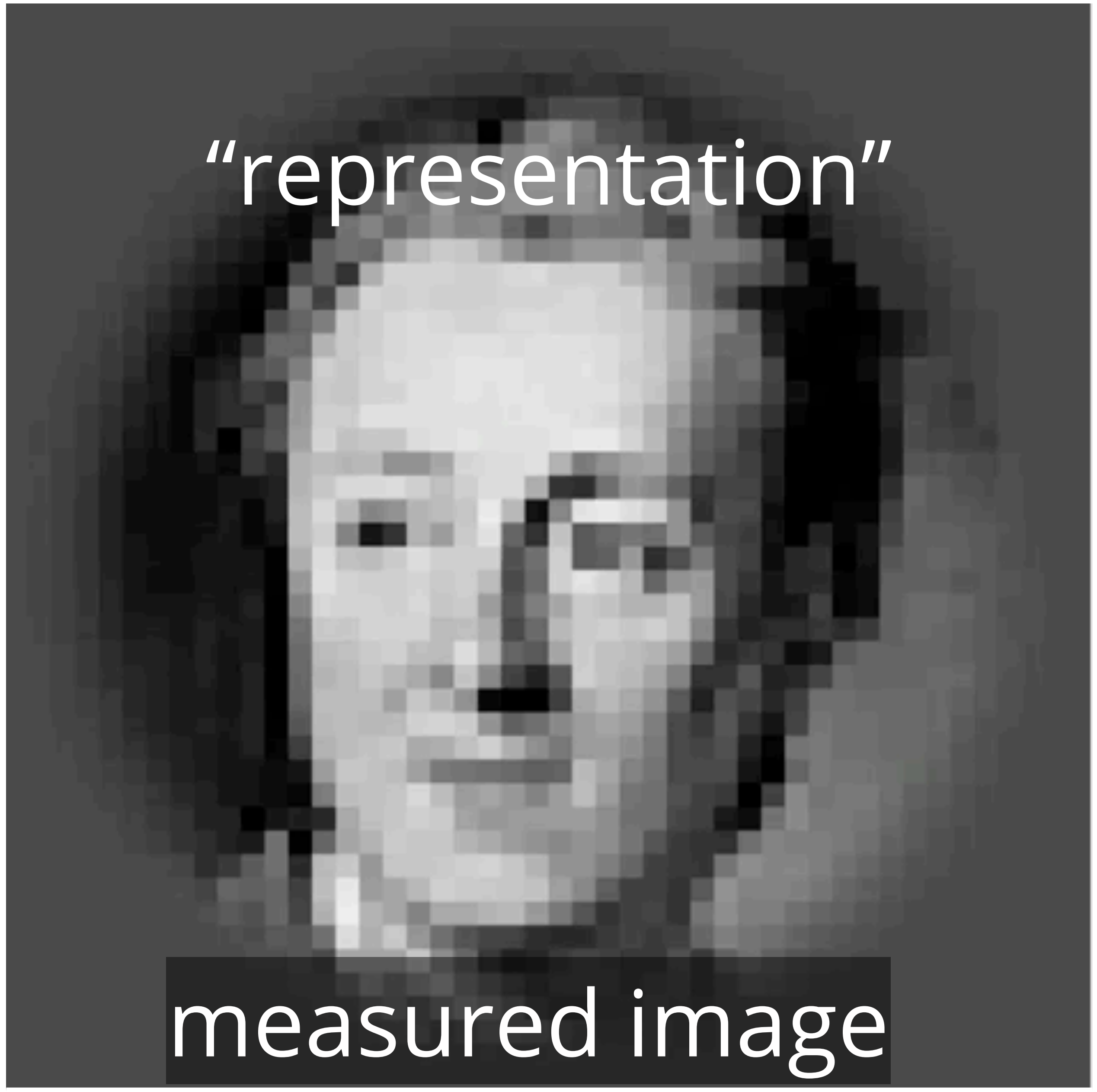


Matlab, Numpy great for discrete images

Objects versus **images**

Measurements of objects produce **images**

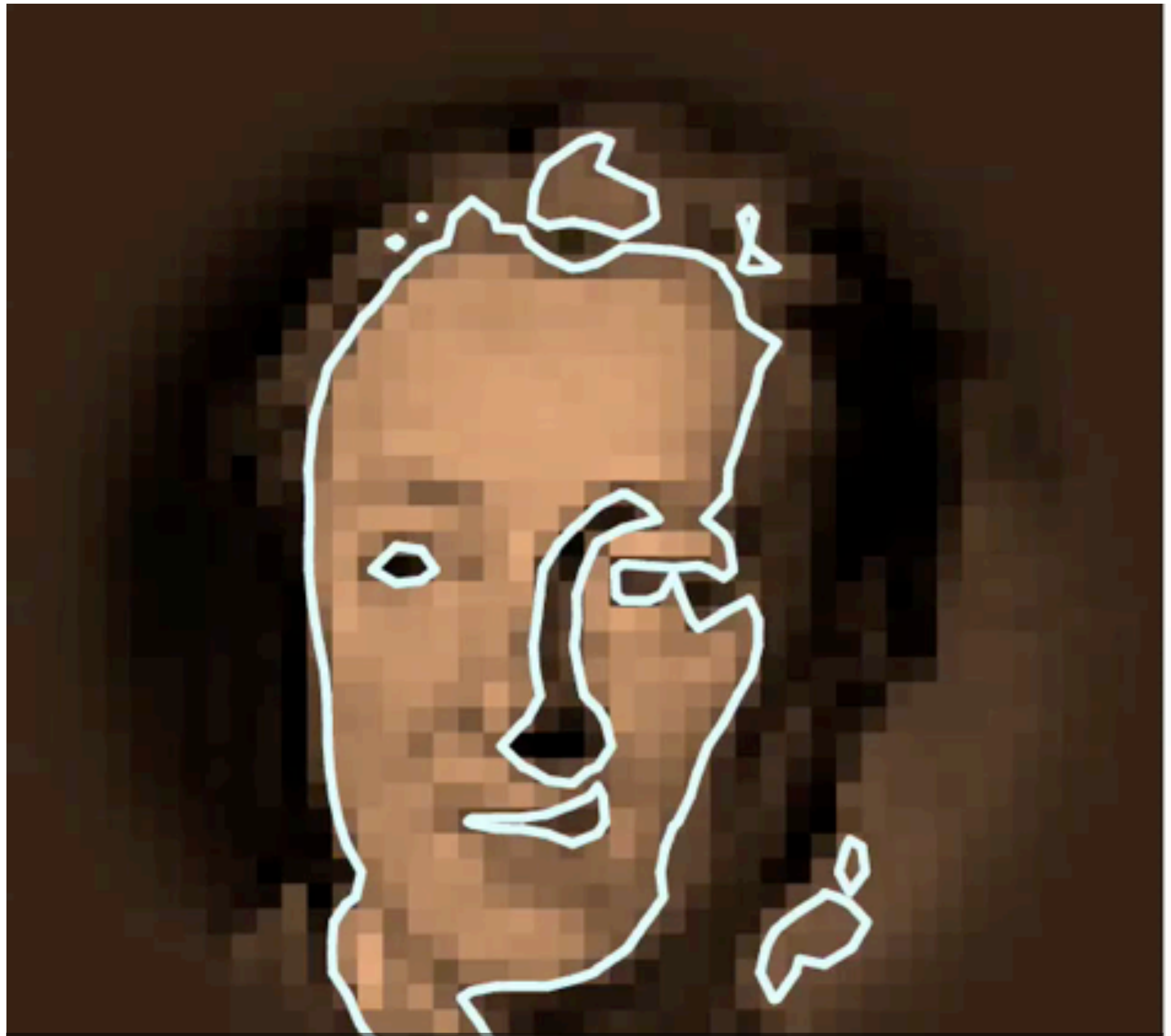




Goal of scientific visualization & analysis is to make statements about the underlying **objects** being studied

Objects versus **images**

Grid orientation/spacing is property of **image** (“representation”)



Marching Squares (on grid)

Newton’s method (on field)

Diderot’s support for continuous fields helps get away from grid details towards object properties

Objects versus **images**

Previous work from 1928:

La trahison des images, Magritte

http://edc13.education.ed.ac.uk/phild/files/2013/01/margritti-this-is-not-a-pipe.jpeg

Minimal Diderot program 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

Computation decomposed into set of mostly autonomous **strands** w/ bulk synchronous execution

Input parameters for initialization

Strand state, including **`output`**

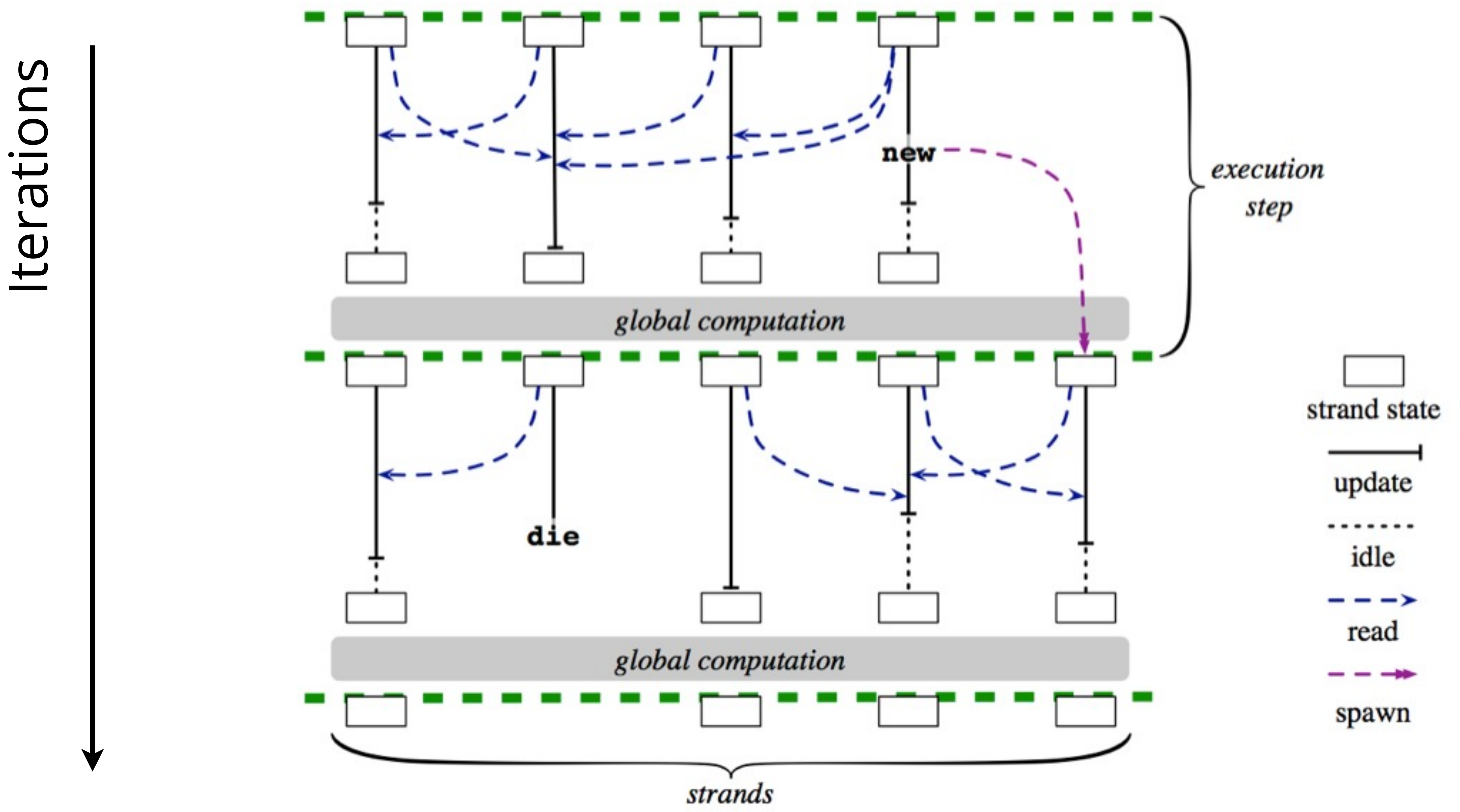
The **`update`** method implements one iteration of program

Strands finish by **`die`** or **`stabilize`**, or use **`new`** to make more strands

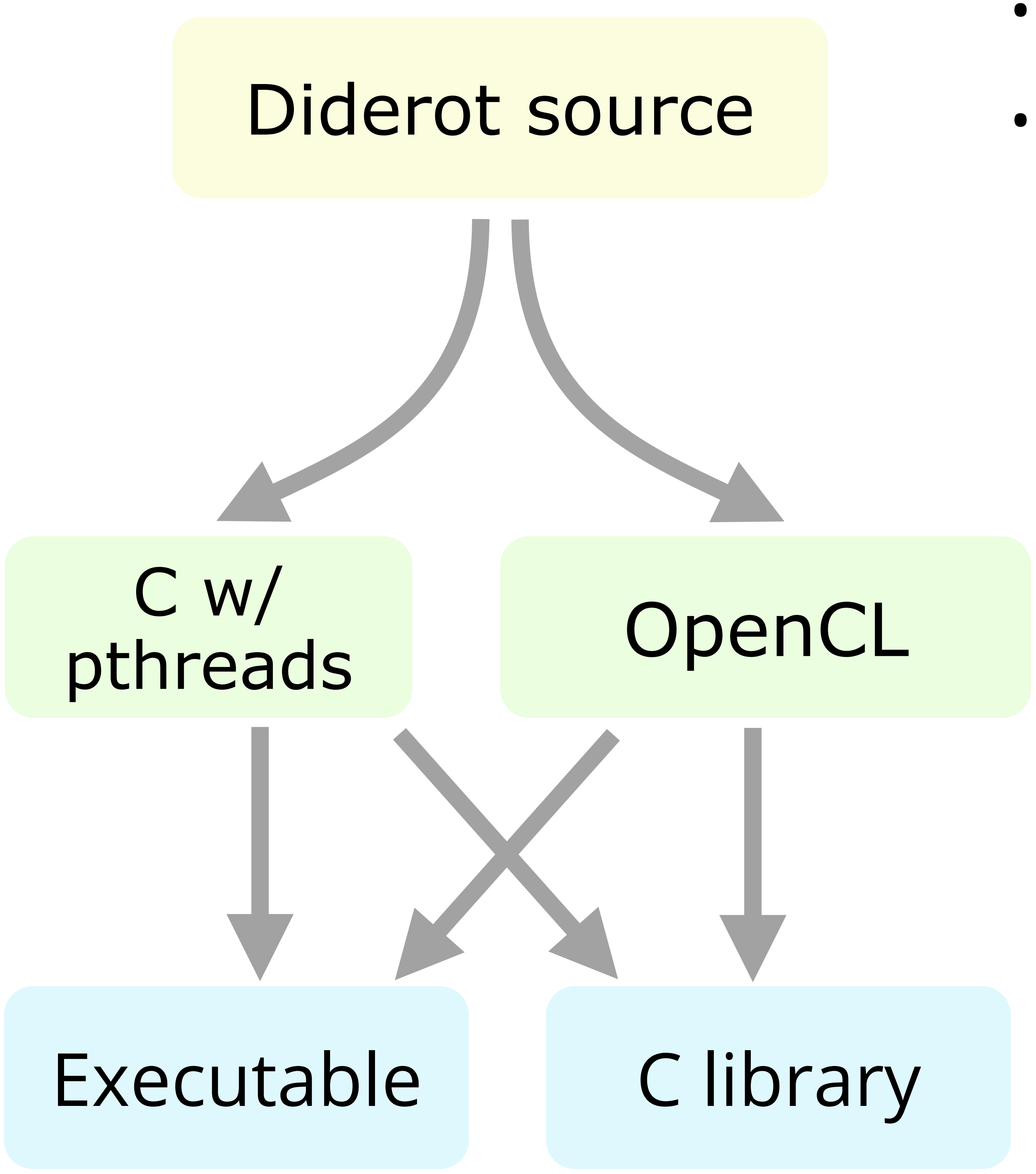
Initialization of collection of strands with comprehension notation

Parallel execution (in CPU or GPU) handled by run-time

Bulk synchronous execution model

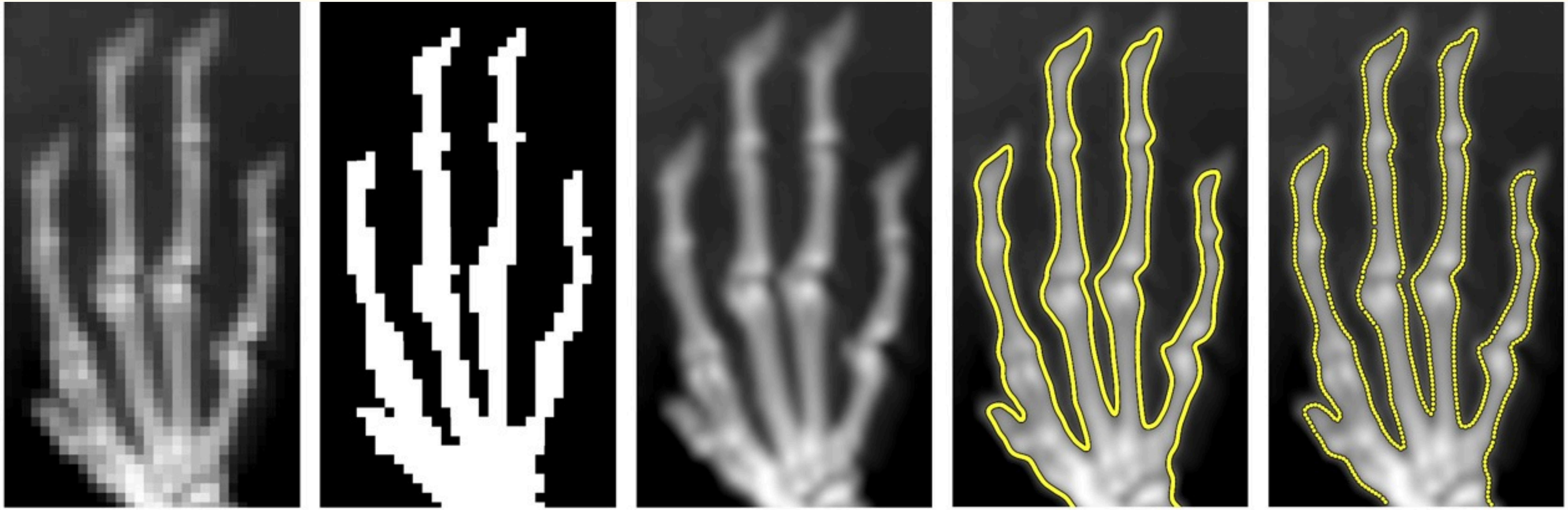


Compilation



- Compiler written in SML/NJ
- Three stages of intermediate representation (IR)
  - "EIN" IR is like lambda calculus meets Einstein summation notation
  - Produces identities:
    - $\nabla \bullet (\nabla \times V) = 0$
    - $\mathrm{Trace}(u \otimes v) = u \bullet v$
  - Section 5.1 of paper
  - Uses **`clang`** to compile executable or C library

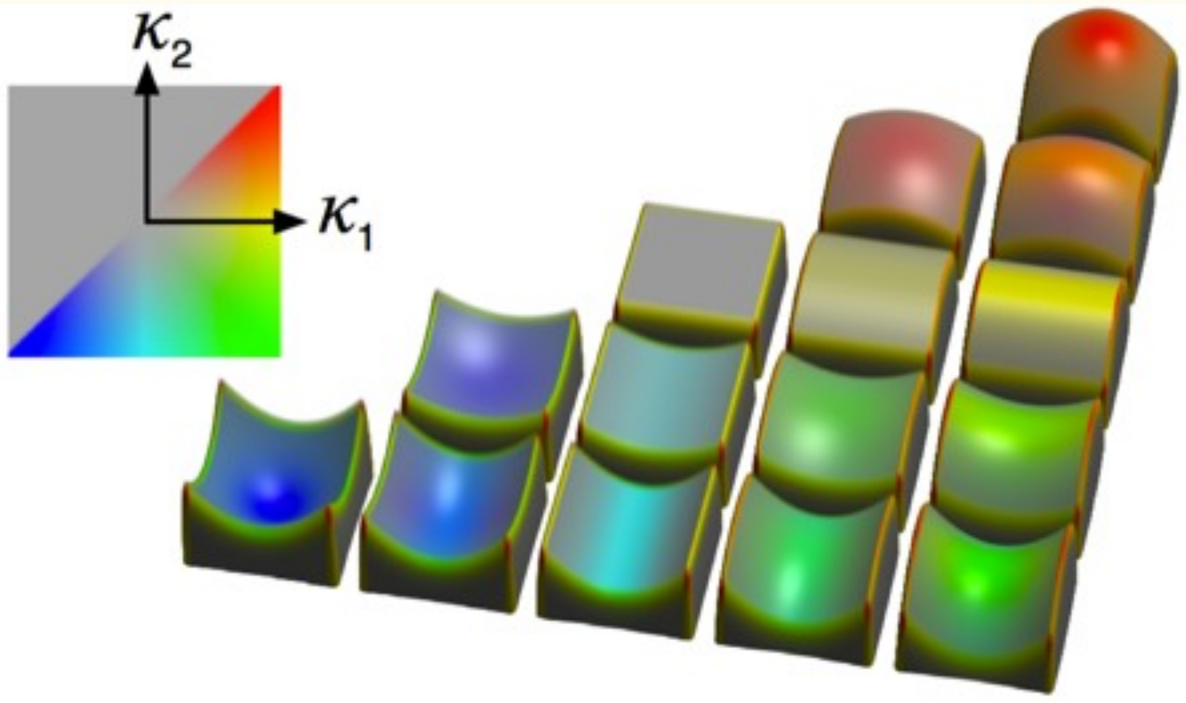
Defining fields from images in Diderot



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

- Convolve image data (a) with kernel to get continuous field (c)
- `field#1(2)[] F = ctmr ⊛ image("hand.nrrd");`
- `field#N(D)[S]`: C^N continuous field: $\mathbb{R}^D \rightarrow$ tensors shape S
  - []: scalar, [3]: 3-vector, [3,3]: 3x3 matrix (**Appendix A** gives grammar)

Revisiting curvature measurement [Kindlmann-VIS-2003]



"
1. Measure the first partial derivatives comprising the gradient $\mathbf{g}$. Compute $\mathbf{n} = -\mathbf{g}/|\mathbf{g}|$, and $\mathbf{P} = \mathbf{I} - \mathbf{n}\mathbf{n}^T$.
2. Measure the second partial derivatives comprising the Hessian $\mathbf{H}$ (Equation 1). Compute $\mathbf{G} = -\mathbf{PHP}/|\mathbf{g}|$.
3. Compute the trace T and Frobenius norm F of $\mathbf{G}$. Then,

$$\kappa_1 = \frac{T + \sqrt{2F^2 - T^2}}{2}, \ \kappa_2 = \frac{T - \sqrt{2F^2 - T^2}}{2}.$$
"

Direct (coordinate-free) notation encourages basis-independent (**representation-independent**) code

```
// volume dataset
field#2(3)[] F = bspln3⊛image("quads.nrrd");
// RGB colormap of K1, K2
field#0(2)[3] RGB = tent⊛image("rgb.nrrd");
...
  update {
    ...
    vec3 g = ∇F(pos);
    vec3 n = -g/|g|; // or -normalize(g);
    tensor[3,3] H = ∇⊗∇F(pos);
    tensor[3,3] P = identity[3] - n⊗n;
    tensor[3,3] G = -(P•H•P)/|grad|;
    real disc = sqrt(2*|G|^2 - trace(G)^2);
    real k1 = (trace(G) + disc)/2;
    real k2 = (trace(G) - disc)/2;
    // find material RGBA
    vec3 matRGB = RGB([k1, k2]);
    ...
}
```

Volume rendering soft isosurfaces

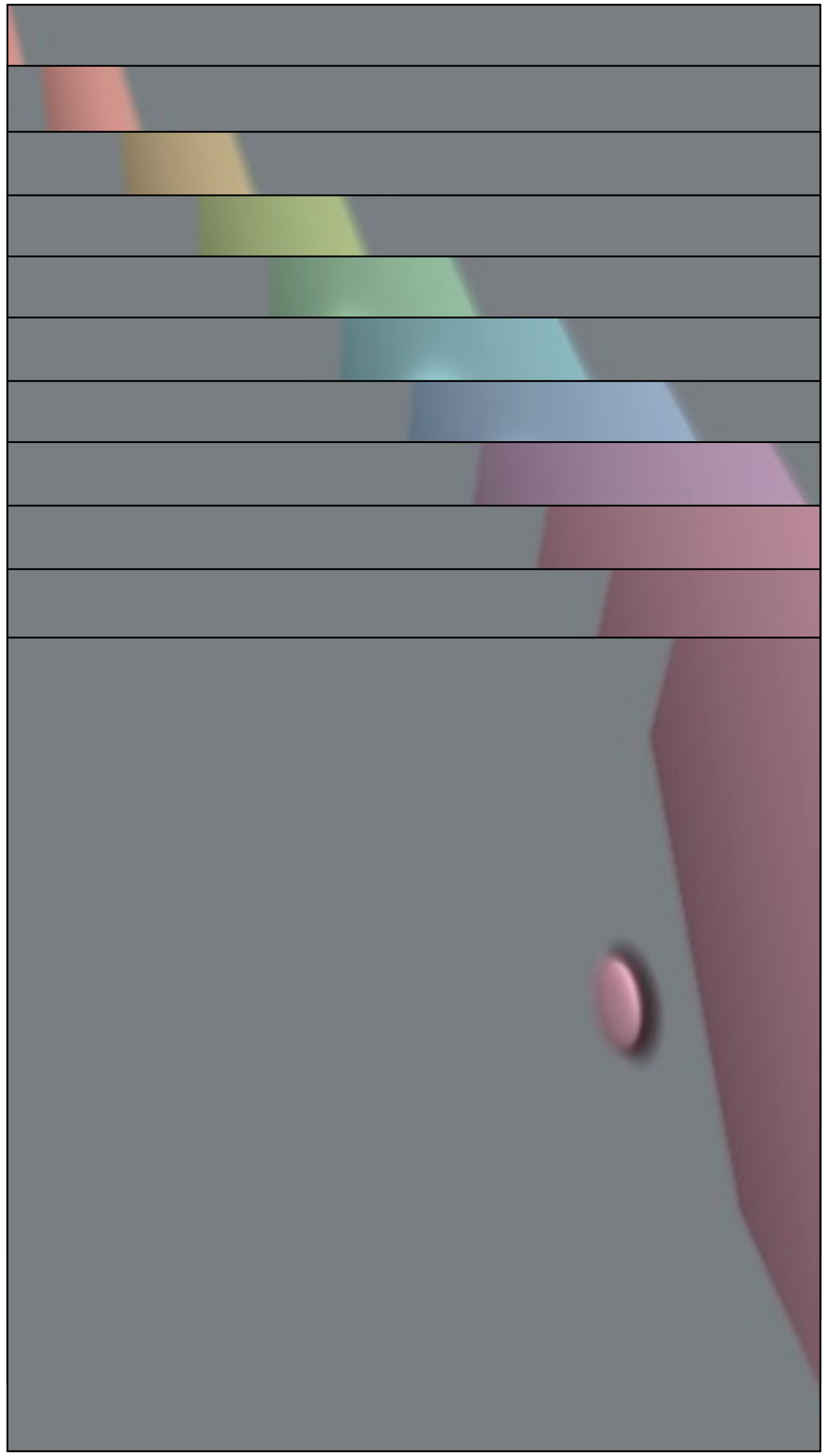
```
field#0(1)[3] cmap = tent ⊛ image("isobow.nrrd");
field#4(3)[] V = bspln5 ⊛ image("canny.nrrd");
field#4(3)[] F = V - isoval;
...
function real alpha(real v, real g) = max(0, 1 - |v|/(g*thick));
...
strand raycast(int ui, int vi) {
  real transp = 1;
  vec3 rgb = [0,0,0]; output vec4 rgba = [0,0,0,0];
  update {
    if (rayN > camVspFar) { stabilize; }
    real val = F(x);
    vec3 grad = -∇F(x);
    real a = alpha(val, |grad|);
    real shade = max(0, normalize(grad)•light);
    rgb += transp*a*(0.2 + 0.8*shade)*color(x);
    transp *= 1 - a;
  }
  stabilize {
    real a = 1-transp;
    if (a > 0) rgba = [rgb[0]/a, rgb[1]/a, rgb[2]/a, a];
  }
}
initially [ raycast(ui, vi)
             | vi in 0..iresV-1, ui in 0..iresU-1 ];
```

Isosurface is zero level-set

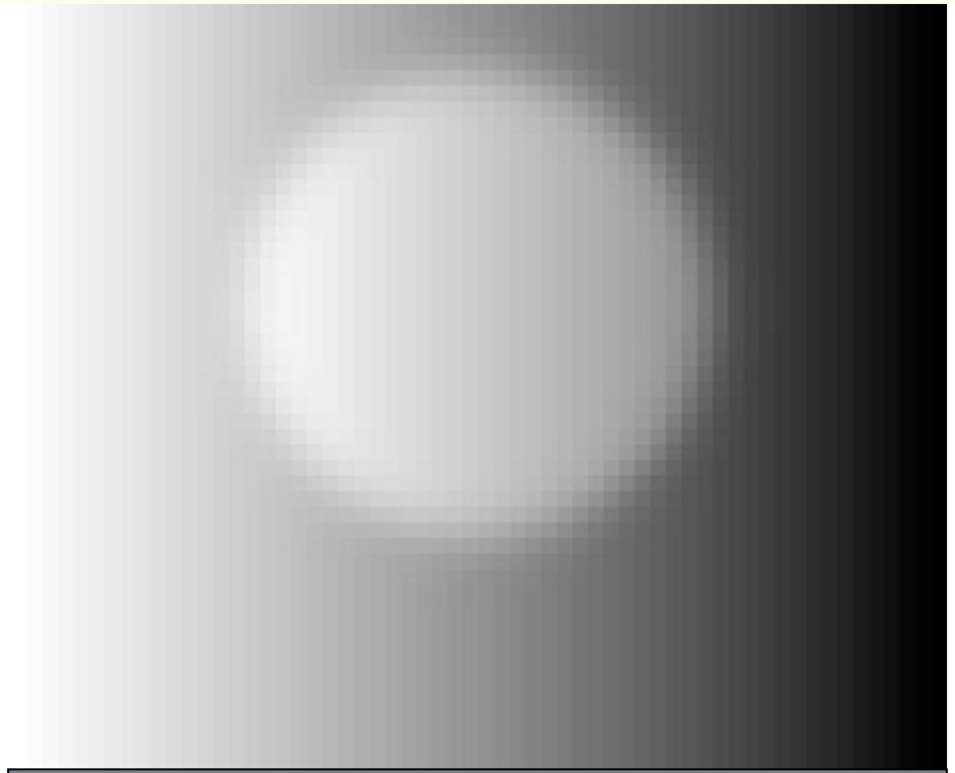
[Levoy-CGnA-1988]

Over operator with pre-multiplied alphas

set final output rgba



Volume rendering material boundaries



How to show material boundaries?

Canny edge [Canny-PAMI-1986]:

$|\nabla v|$ maximal w.r.t motion along $\nabla v/|\nabla v|$

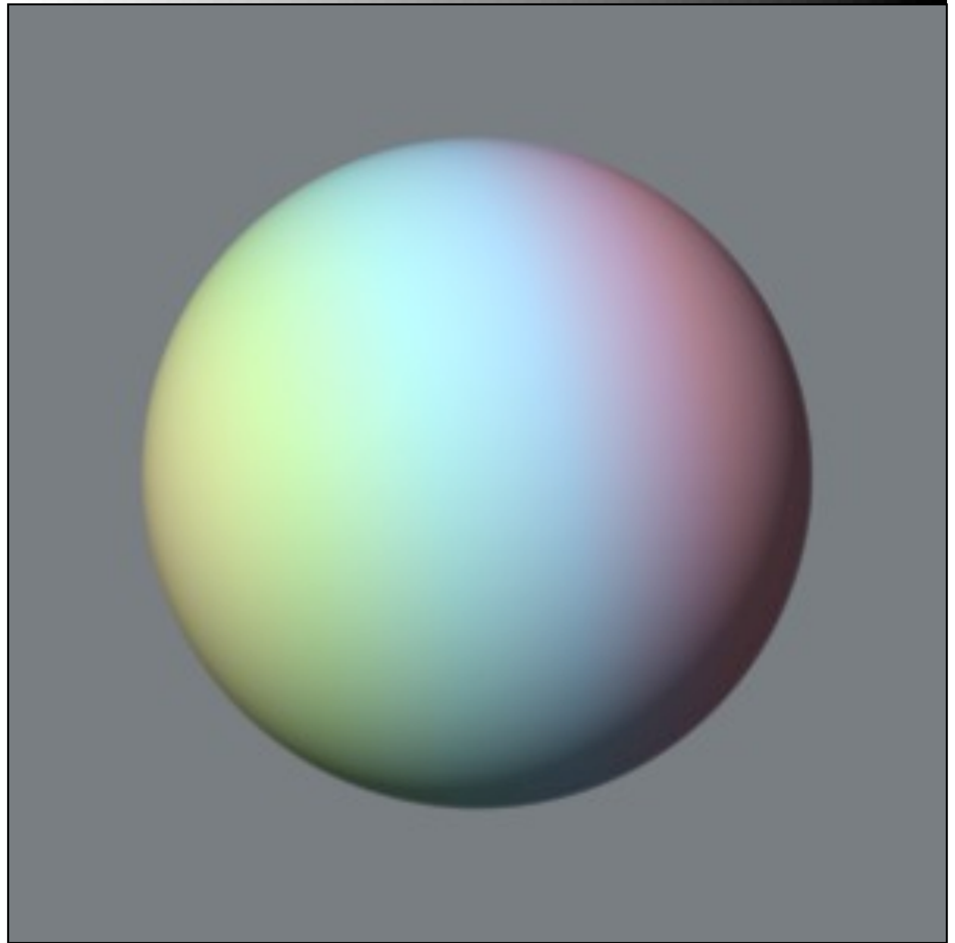
$\Rightarrow \nabla|\nabla v| \cdot \nabla v/|\nabla v| == 0$

Change one line of Diderot code:

```
field#4(3)[] F = V - isoval;
field#2(3)[] F = ∇|∇v| • ∇v/|∇v|;
```

For shading, Diderot computes ∇F

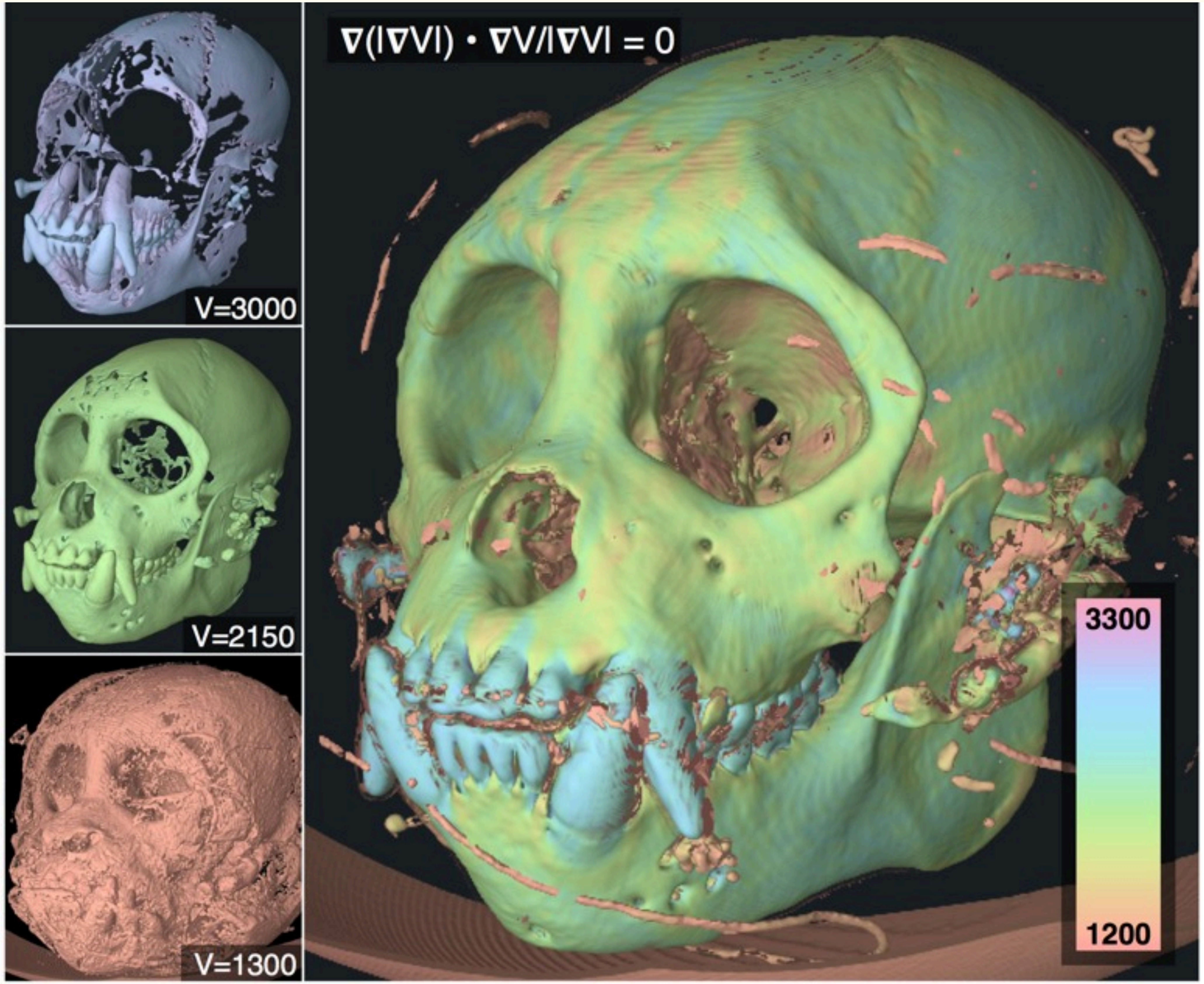
involves 3rd derivatives (!)



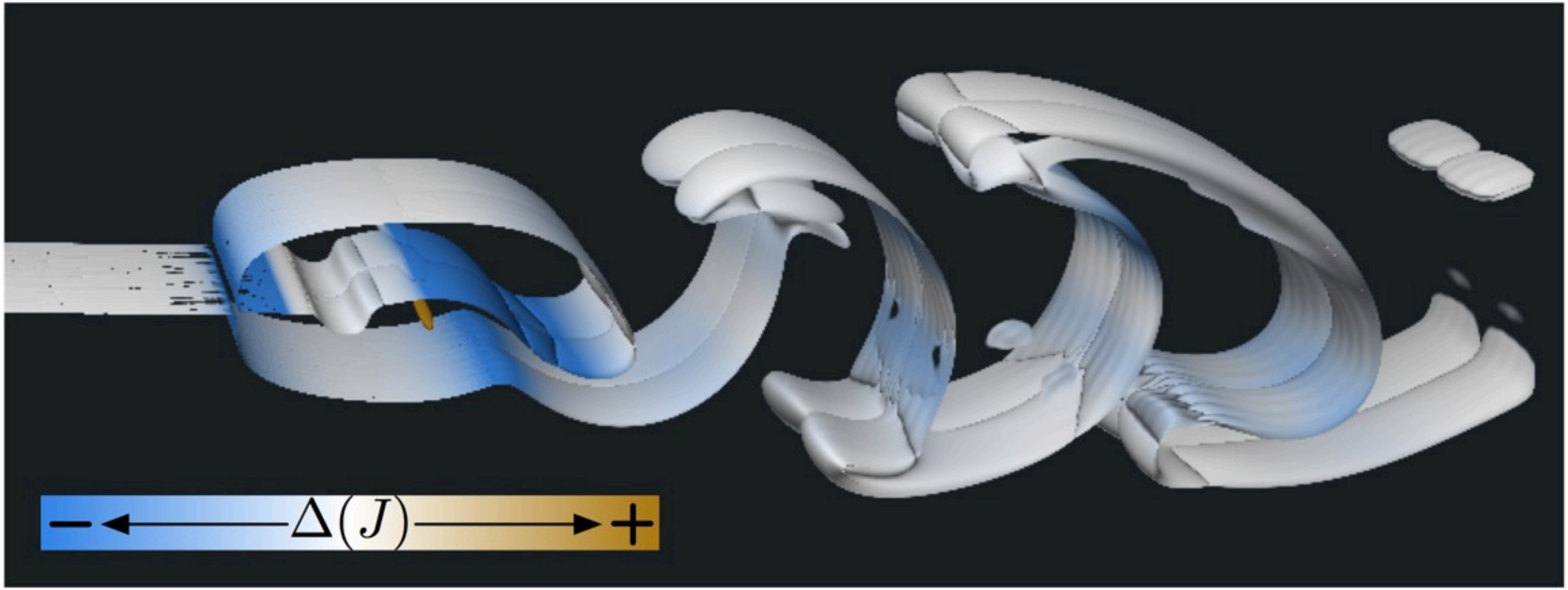
Canny edges in real CT scan

There is no isosurface that captures the bone surface

Canny edge surface shows underlying value (novel vis)



Rendering flow field structure




```
field#4(3)[3] V = bspln5 ⊛ image("flow.nrrd");
field#3(3)[] F = (V/|V|) • (∇×V/|∇×V|);
```

Normalized Helicity [Degani-AIAAJ-1990]

Rendering anisotropy of diffusion tensor field

```
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)[] FA = sqrt(3.0/2.0)*|E|/|V| - isoval;
```

Compare with original definition

[Basser-JMRB-1996]

"
$$\underline{\boldsymbol{D}} = \underline{\mathbf{D}} - \langle \mathrm{D} \rangle \underline{\mathbf{I}}$$

$$\mathrm{FA} = \sqrt{\frac{3}{2}} \frac{\sqrt{\underline{\boldsymbol{D}} : \underline{\boldsymbol{D}}}}{\sqrt{\underline{\mathbf{D}} : \underline{\mathbf{D}}}}$$
"

Not just for volume rendering!

Streamlines in flow field


```
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;
    }
  }
}
```

Output is set of sequence of points

Legible integration

Compile to executable or C Library

Stand-alone executable w/ command-line interface

- each input has corresponding option

  `input real isoval = 10;` ⇒ … `-isoval 10` …

Compile to library, with API for

- **Setting inputs, retrieving outputs**

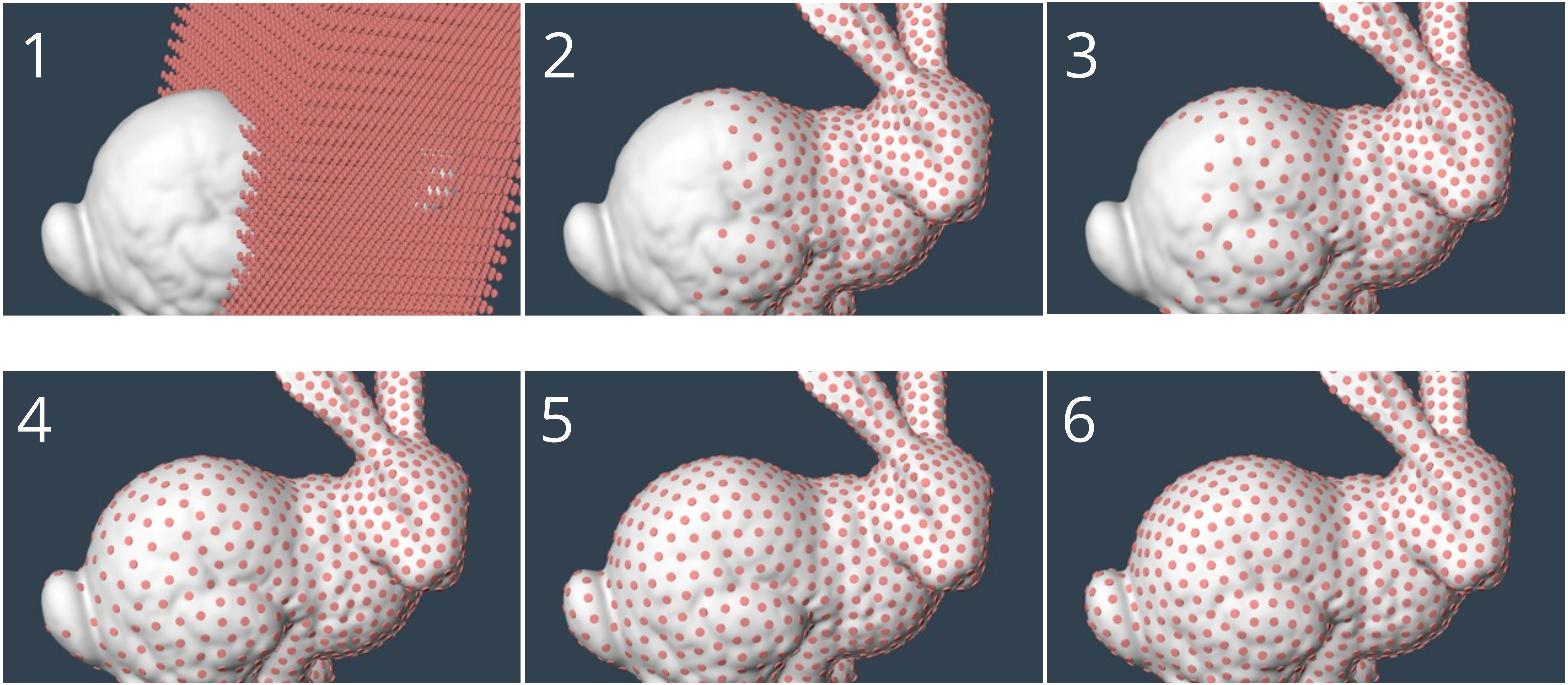
  ```
  ISO_InVarSet_isoval(ISO_World_t *wrld, float v);
  ISO_OutputGet_pos(ISO_World_t *wrld, Nrrd *data);
  ```

- Initializing, stepping through computation

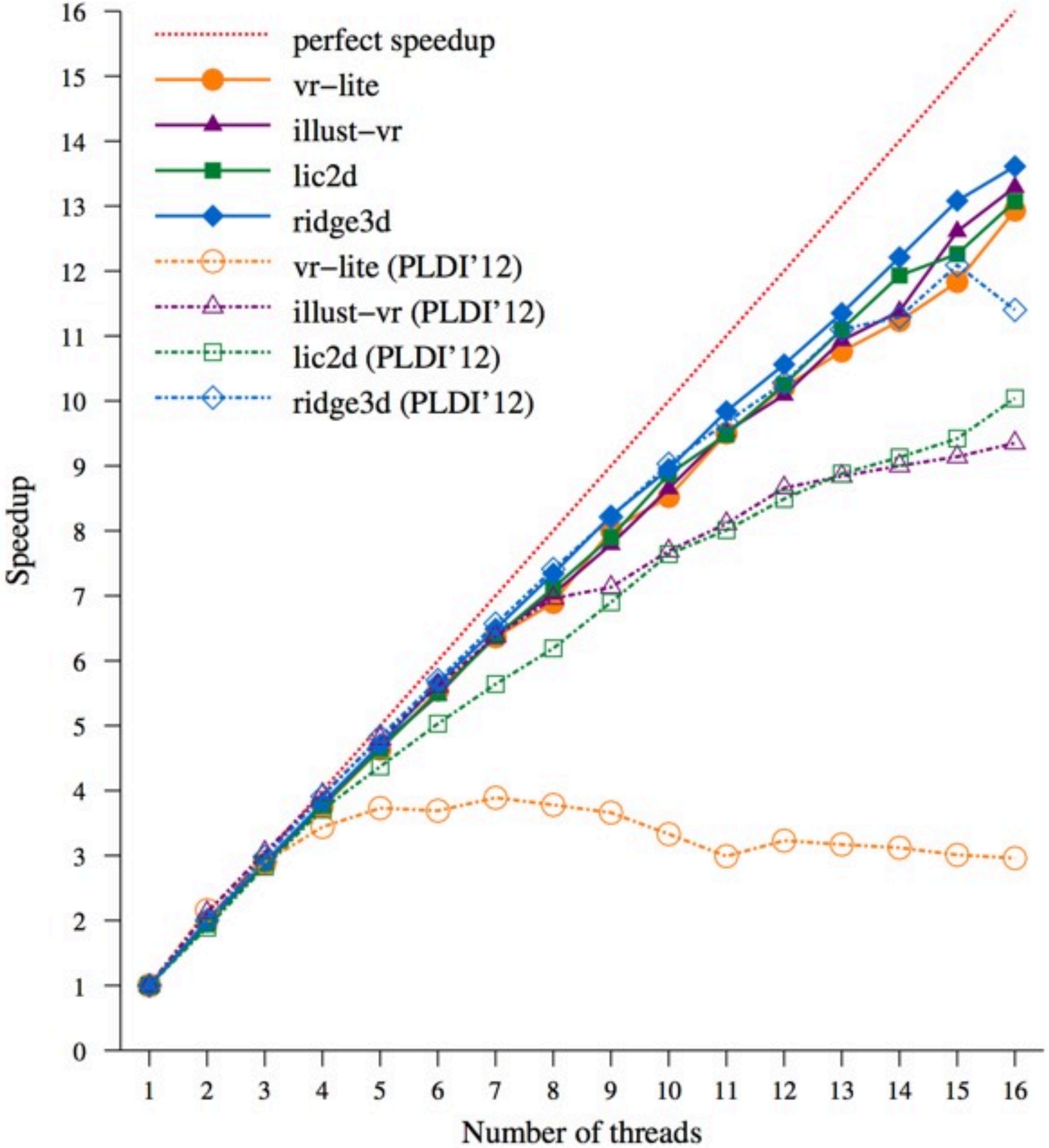
Appendix B: 2D particle system example

Let’s watch 3D particle system go …

(snapshots from interactive demo shown during talk)



Speedup curves (on CPU)



Significant improvement in speedup relative to previous 2012 paper in Programming Language Design and Implementation

[Chiw-PLDI-2012]

Performance numbers

Program	Teem	Diderot (PLDI ’12) Seq.	1P	6P	12P	16P	Diderot (this paper) 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

Execution times in seconds, averaged over 10 runs

“Teem” = hand-coded C, not parallel (no pthreads)

Intel Xeon E5-2687W (16 cores), Ubuntu 12.04.

OpenCL w/ NVIDIA Tesla K20c, using NVIDIA’s CUDA 6.0 driver

Appendix C compares with hand-written OpenCL

Ongoing Work

- Stronger math abstractions
  - Declarative mathematical statement of algorithm
  - Time-varying fields (time as special dimension)
- Better computing
  - New backends: CUDA and MPI (for larger datasets)
  - Better GPU performance through OpenCL
  - New fields: (higher-order) Finite Element Meshes
- Better usability: debugger, GUI generation

Conclusions

Good progress on an ambitious goal

Diderot good for:

- Writing **legible** vis programs that run in parallel
- Trying new sci vis methods in terms of fields, tensors

Diderot not (yet) good for:

- Working directly on grids (e.g. Marching Cubes, level-set segmentation, per-pixel classification)
- Fast execution on big data essential, rather than fast implementation

References

[Basser-JMRB-1996] 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.

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

[Chiw-PLDI-2012] C Chiw, G Kindlmann, J Reppy, L Samuels, and N Seltzer. Diderot: A Parallel DSL for Image Analysis and Visualization. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 111–120, June 2012.

[Choi-VIS-2014] 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.

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

[Jablin-IPDPS-2011] J. Jablin, P. McCormick, and M. Herlihy. Scout: High-performance heterogeneous computing made simple. In Proceedings of IEEE International Symposium on Parallel and Distributed Processing, pages 2093–2096, 2011

[Kindlmann-VIS-2003] G Kindlmann, R Whitaker, T Tasdizen, and T Möller. Curvature-Based Transfer Functions for Direct Volume Rendering: Methods and Applications. In Proceedings of IEEE Visualization, pages 513–520, October 2003

[Kindlmann & Scheidegger 2014]: G Kindlmann and C Scheidegger. IEEE Transactions on Visualization and Computer Graphics (Proceedings VIS 2014), 20(12):2181–2190, November 2014.

[Kindlmann-VIS-2015] G Kindlmann, C Chiw, N Seltzer, L Samuels, and J Reppy. Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis. IEEE TVCG (Proceedings VIS 2015), 22(1):867–876, January 2016

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

[McCormick-VIS-2004] P. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In Proceedings of IEEE Visualization 2004, pages 171–178, 2004

[McCormick-JPC-2007] 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.

[McCormick-WOLFHPC-2014] P. McCormick, C. Sweeney, N. Moss, D. Prichard, S. K. Gutierrez, K. Davis, J. Mohd-Yusof. Exploring the Construction of a Domain-Aware Toolchain for High-Performance Computing. Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC). pages 1–10, 2014.

[Rautek-VIS-2014] 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

Thank you!

National Science Foundation CCF-1446412

Data: Mouse paw: University of Utah SCI group, NIH NIGMS grant P41GM103545 | Capuchin Skull: Callum Ross, University of Chicago | Vortex flow field: Resampling by Tino Weinkauf of Navier-Stokes simulation by S. Camarri, M.-V. Salvetti, M. Buffoni, and A. Iollo | Double-point stress field: Xavier Tricoche, Purdue University | Diffusion Tensor Brain: Centre for Functional MRI of the Brain, John Radcliffe Hospital, Oxford University | 2D flow field: Wolfgang Kollmann, UC Davis

- Example programs are accumulating here:
  `https://github.com/Diderot-Language/examples`
- Google Group: `https://goo.gl/kXpxhV`
- If want to follow-along with demos, git pull
  https://github.com/Diderot-Language/examples

(break)

Example running Heron

From a checkout of

`https://github.com/Diderot-Language/examples`

`cd heron`

`../../vis12/bin/diderotc --exec heron.diderot`

`./heron --help` (to see usage info)

`./heron` (run program)

(on one line)

```
unu crop -i vrie.nrrd -min 0 0 -max 1 M |
unu jhisto -b 300 300 |
unu quantize -b 8 -o tmp.png
```

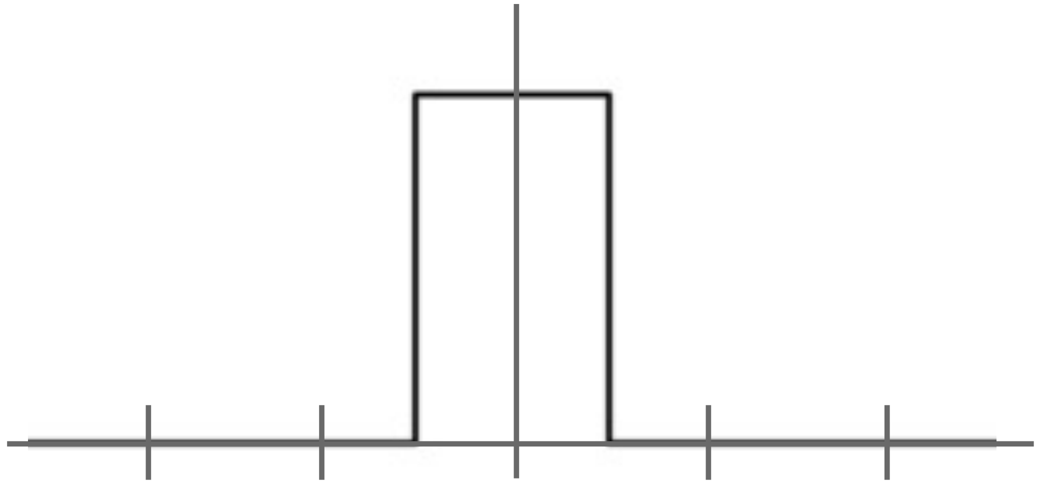
(view tmp.png)

1D Convolution defined

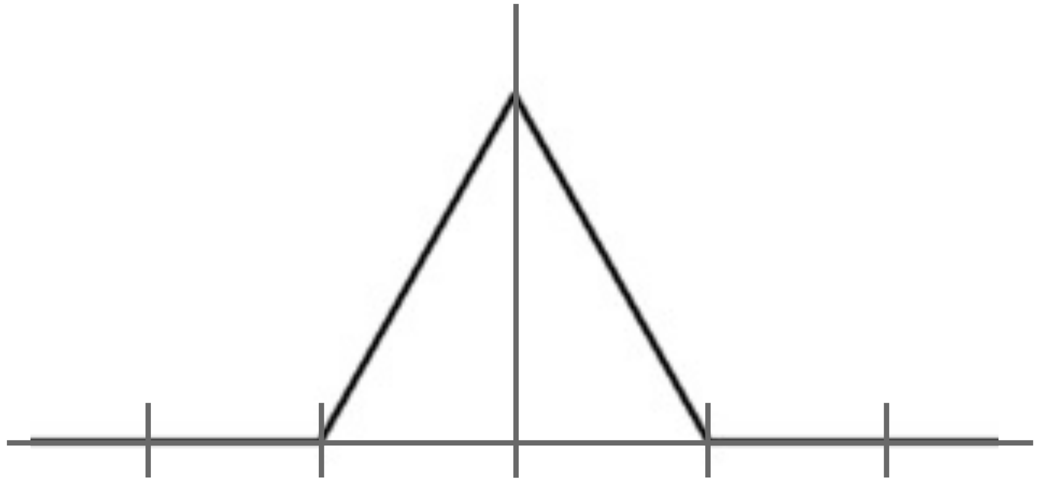
discrete data samples V[], **continuous** reconstruction kernel k

$$f(x) = (g \circledast k)(x) = \int g(t)k(x-t)dt$$
$$= \int \sum_j V[j]\delta(t-j)k(x-t)dt$$
$$= \sum_j V[j] \int \delta(t-j)k(x-t);$$
$$f(x) = \sum_j V[j]k(x-j)$$

2D Convolution examples 1

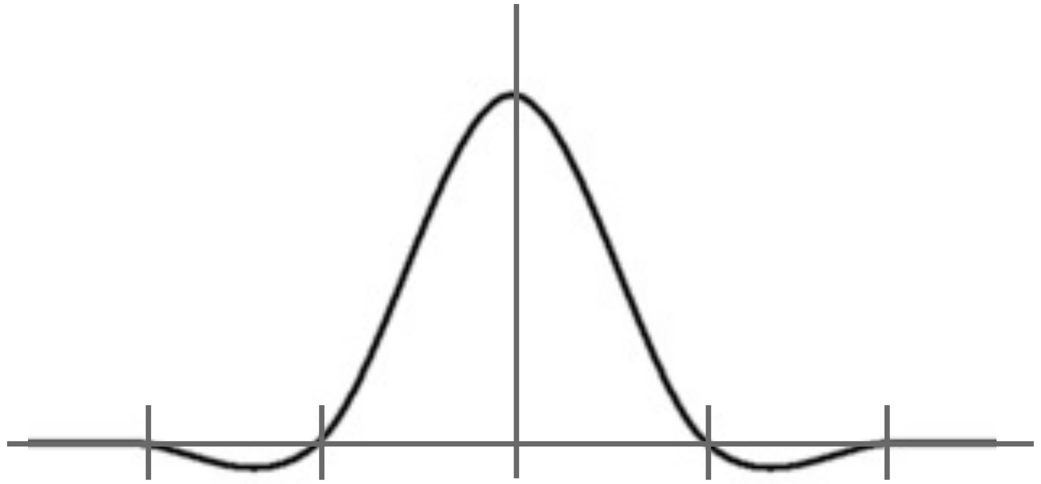


box, nearest neighbor

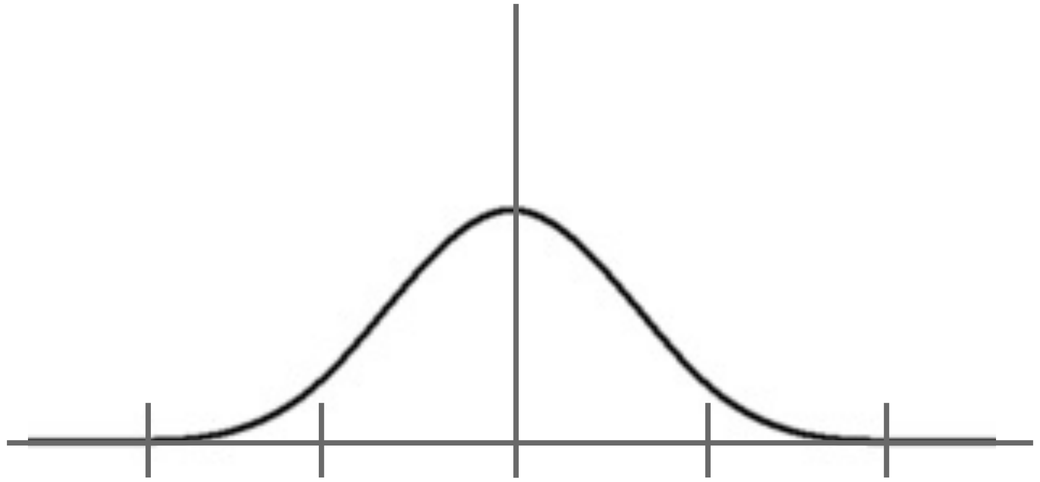


“tent” = linear interpolation

2D Convolution examples 2



“ctmr” = Interpolating cubic
“Catmull-Rom” spline



“bspln3” = (non-interpolating)
Cubic B-spline

Reconstruction with nice kernels

Diderot used for Algebraic Vis paper [Kindlmann & Scheidegger 2014]

Reconstruction with simple ("ctmr", top) vs ("c4hexic" bottom) kernel; c4hexic can exactly reconstruct cubics

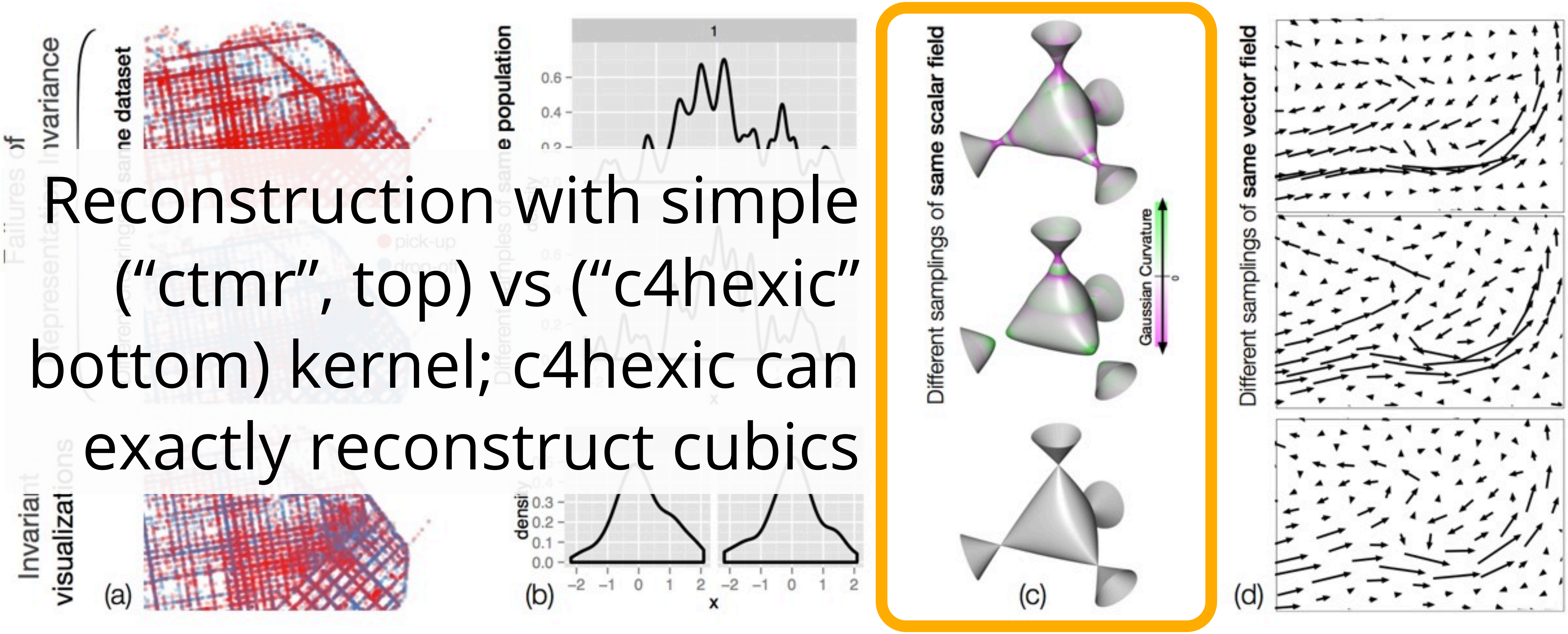


Fig. 2: Our Invariance Principle illustrated with taxi pick-ups and drop-offs (a), two different samples from a population (b), volume renderings of sampled 3D cubic polynomial (c), and vector glyphs in a 2D flow field (d). The upper pair of adjacent visualizations are of exactly the same underlying data or object, but give different impressions due to arbitrary differences in representation, sometimes beyond the control of the designer. The bottom row demonstrates the Invariance Principle with visualizations that do not depend on representation choice.

2D image sampler/viewer again in a checkout of

`https://github.com/Diderot-Language/examples`

```
cd vimg
```

(peruse source code)

(sscand = southern Scandanavia)

```
ln -s ../data/sscand.nrrd img.nrrd
../../vis12/bin/diderotc --exec vimg.diderot
./vimg --help
```

(to see usage info)

```
./vimg -cent 300 400 -fov 30
unu quantize -b 8 -i gray.nrrd -o tmp.png
```

(view tmp.png)

(try again with different kernels; have to recompile each time)

(try again with `-w 1` to see gradient magnitude)

```
./vimg -cent 300 400 -fov 30 -w 2 -iso 1070 -th 40
```

How to get even thickness line?

Taylor's Theorem

$$f(x+\varepsilon) \approx f(x) + f'(x)\varepsilon$$

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

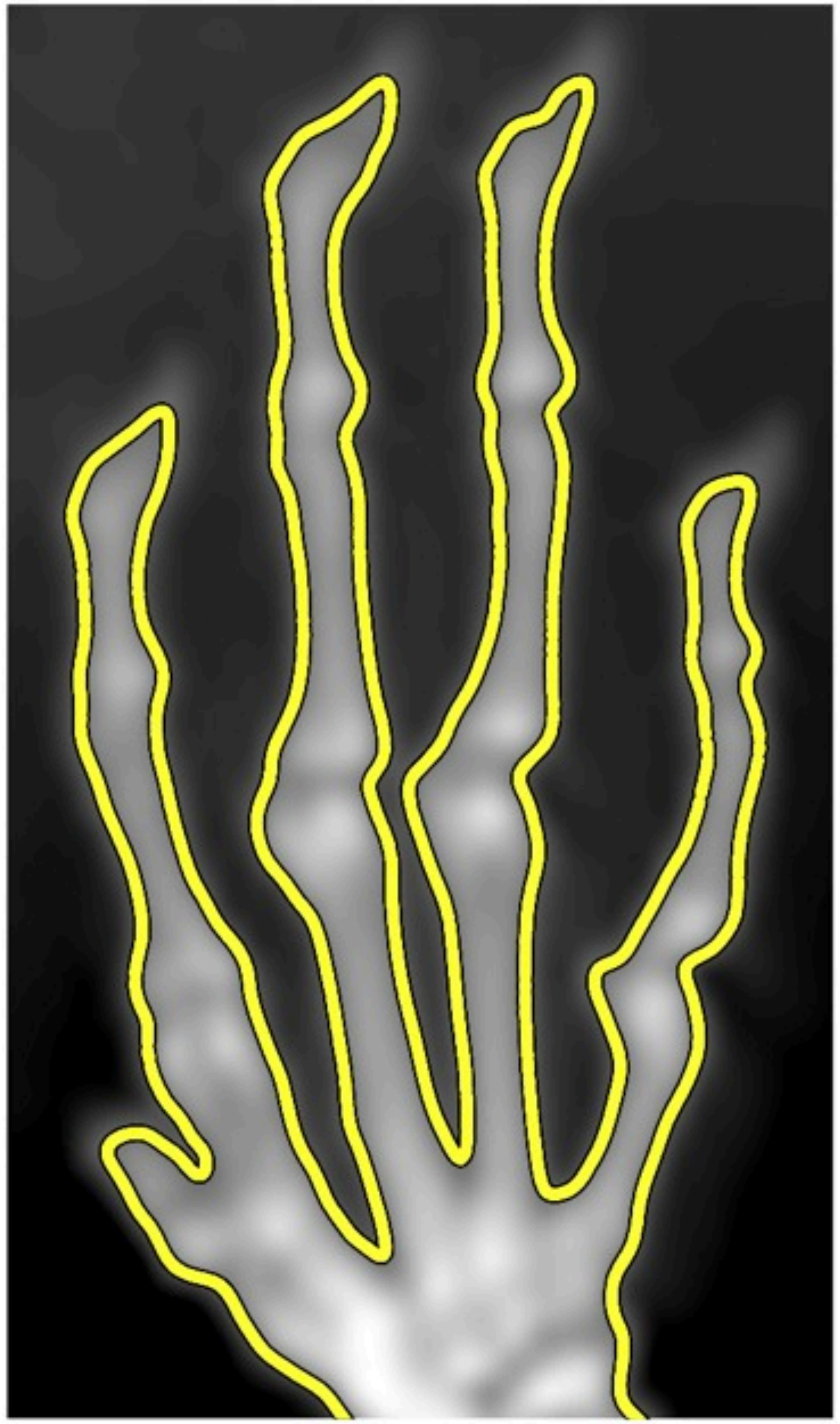
And from this can derive Newton's method…

```
./vimg -cent 300 400 -fov 30 -w 3 -iso 1070 -th 0.4
```

Compare to Levoy's paper [Levoy-CGnA-1988]

Example complete program: isocontour sampling

```
field#1(2)[] F = c4hexic ⊛ image("hand.nrrd");
input int size0; input int size1;
input int stepsMax = 10;
input real epsilon = 0.0001;
input vec2 dir0; input vec2 dir1;
input vec2 orig;
strand isofind(vec2 pos0) {
  output vec2 pos = pos0;
  int steps = 0;
  update {
    // Stop after too many steps or leaving field
    if (steps > stepsMax || !inside(pos, F))
      die;
    // one Newton-Raphson iteration
    vec2 delta = -normalize(∇F(pos)) * F(pos)/|∇F(pos)|;
    pos += delta;
    if (|delta| < epsilon)
      stabilize;
    steps += 1;
  }
}
initially { isofind(orig + ui*dir0 + vi*dir1) |
            vi in 0..(size1-1), ui in 0..(size0-1) };
```



Thanks Again

National Science Foundation CCF-1446412

Data: Mouse paw: University of Utah SCI group, NIH NIGMS grant P41GM103545 | Capuchin Skull: Callum Ross, University of Chicago | Vortex flow field: Resampling by Tino Weinkauf of Navier-Stokes simulation by S. Camarri, M.-V. Salvetti, M. Buffoni, and A. Iollo | Double-point stress field: Xavier Tricoche, Purdue University | Diffusion Tensor Brain: Centre for Functional MRI of the Brain, John Radcliffe Hospital, Oxford University | 2D flow field: Wolfgang Kollmann, UC Davis

- Example programs are accumulating here:
  `https://github.com/Diderot-Language/examples`
- Google Group: `https://goo.gl/kXpxhV`
- **Please share your thoughts on how to write another paper about Diderot! GLK@uchicago.edu**