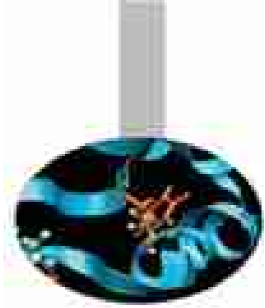
Scientific visualization algorithms

Luigi Calori

Color mapping

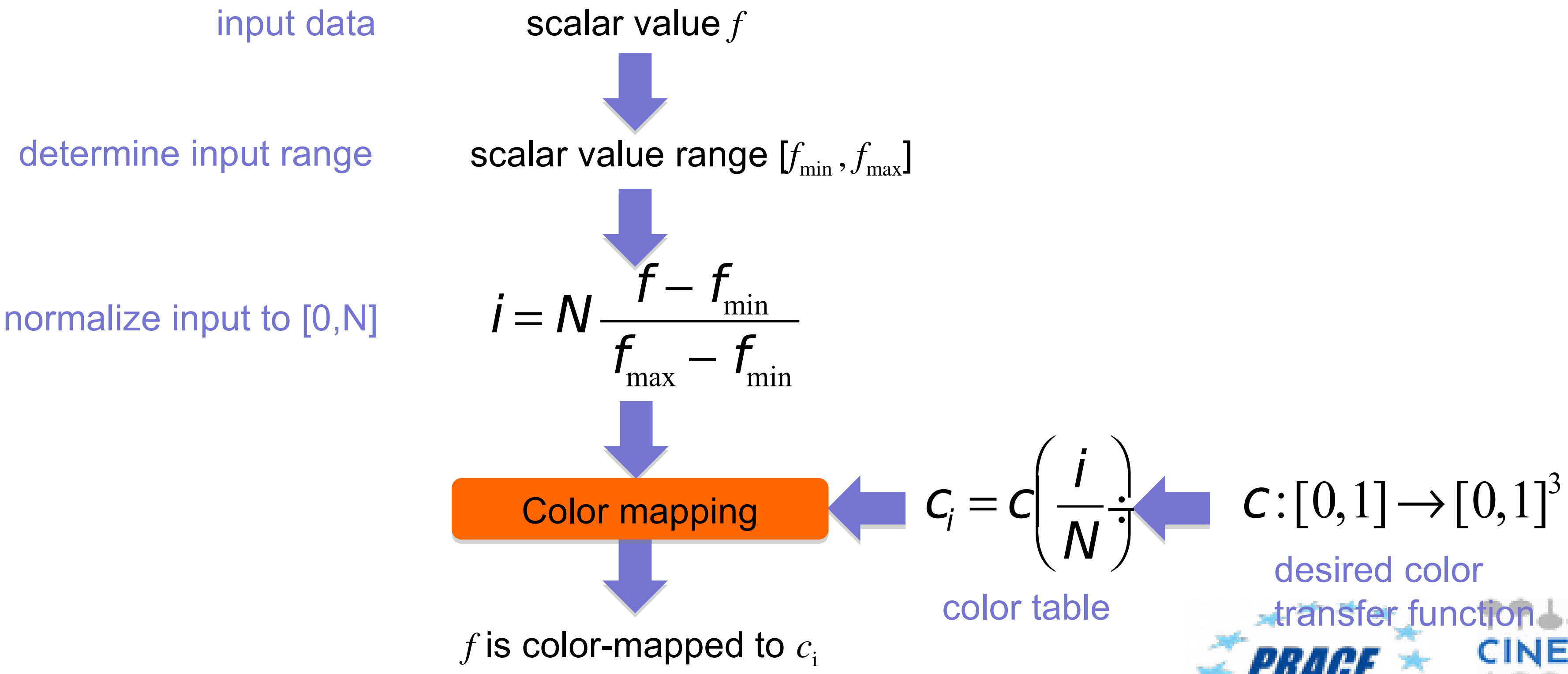


Basic idea

- Map each scalar value $f \in \mathbb{R}$ at a point to a color via a function $c : [0,1] \rightarrow [0,1]^3$

Color tables

- precompute (sample) c and save results into a table $\{c_i\}_{i=1..N}$
- index table by normalized scalar values



input data

scalar value f

determine input range

scalar value range $[f_{\min}, f_{\max}]$

normalize input to [0,N]

$$i = N \frac{f - f_{\min}}{f_{\max} - f_{\min}}$$

Color mapping

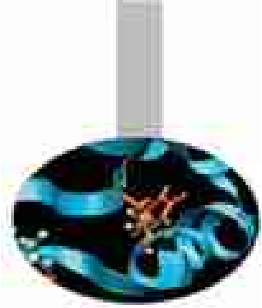
$$c_i = c\left(\frac{i}{N}\right)$$

color table

$$c : [0,1] \rightarrow [0,1]^3$$

desired color transfer function

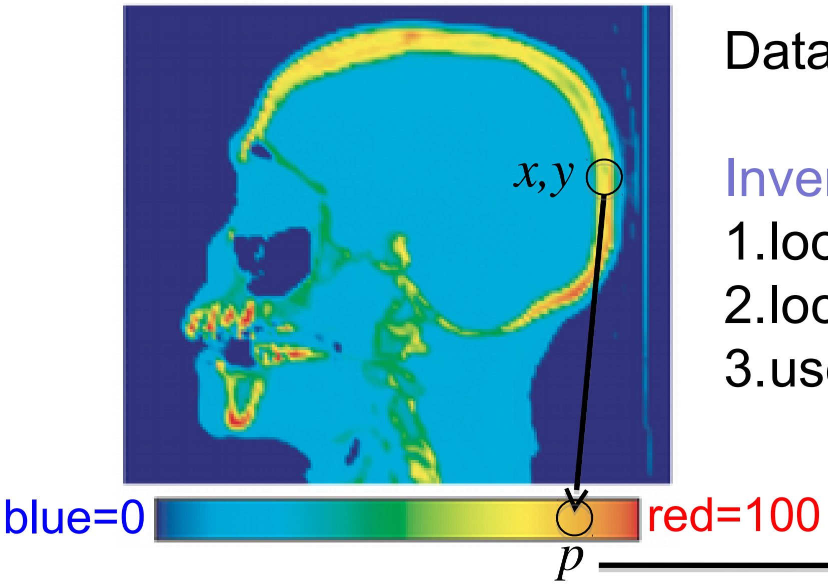
f is color-mapped to c_i



Colormap design

What makes a good colormap?

- map scalar values to colors *intuitively*…
- …so we can visually *invert* the mapping to tell scalar values from colors



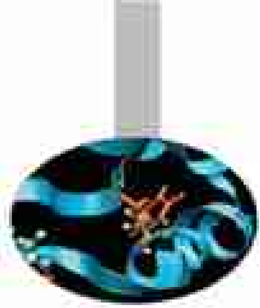
Data values mapped to RGB colors via a colormap

Invert mapping:

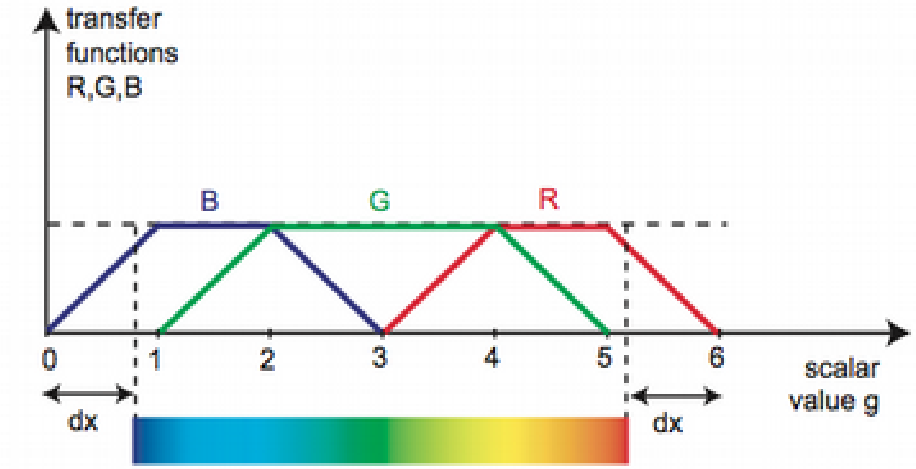
1. look at some point (x,y) in the image → color c
2. locate c in colormap at some position p
3. use the colormap legend to derive data value s from p

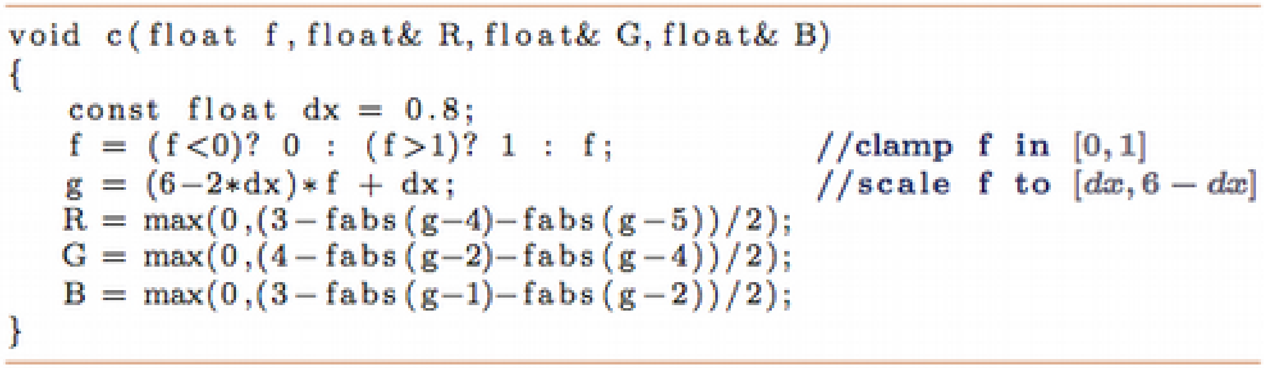
answer: $s = 90$

Rainbow colormap



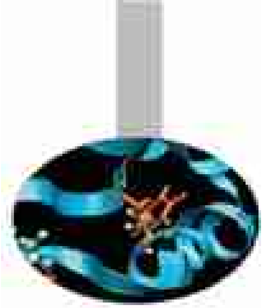
- probably the most (in)famous in data visualization
- intuitive ‘heat map’ meaning
    - cold colors = low values
    - warm colors = high values





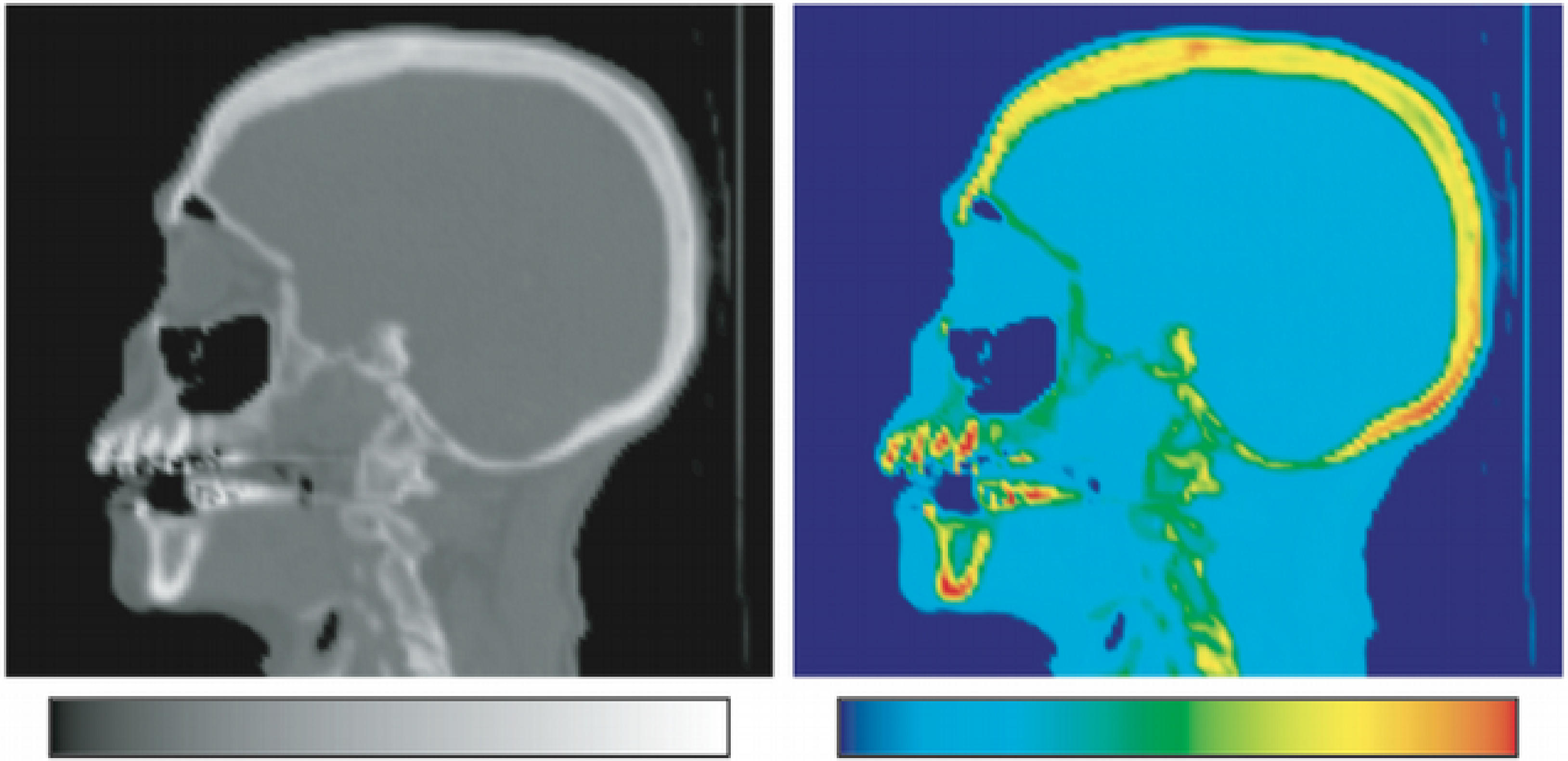
```
void c(float f, float& R, float& G, float& B)
{
    const float dx = 0.8;
    f = (f<0)? 0 : (f>1)? 1 : f;          //clamp f in [0,1]
    g = (6-2*dx)*f + dx;                  //scale f to [dx,6-dx]
    R = max(0,(3-fabs(g-4)-fabs(g-5))/2);
    G = max(0,(4-fabs(g-2)-fabs(g-4))/2);
    B = max(0,(3-fabs(g-1)-fabs(g-2))/2);
}
```

Gray-value colormap



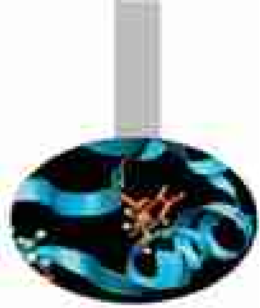
- brightness = value
- natural in some domains (X-ray, angiography)

2D slice in 3D CT dataset
Scalar value: tissue density



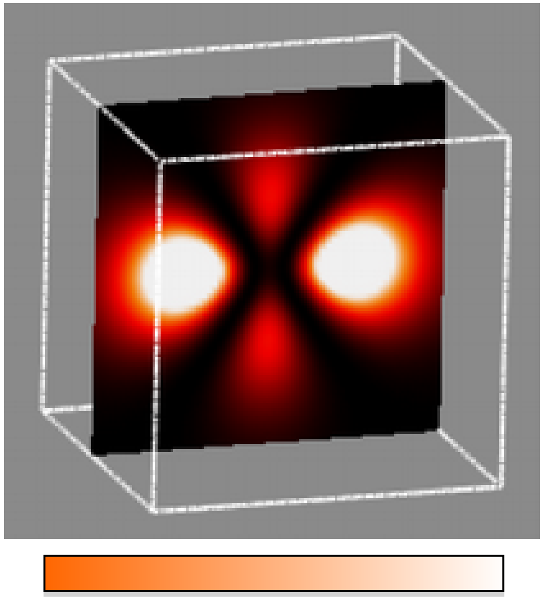
Gray-value colormap
- white = hard tissues (bone)
- gray = soft tissues (flesh)
- black = air

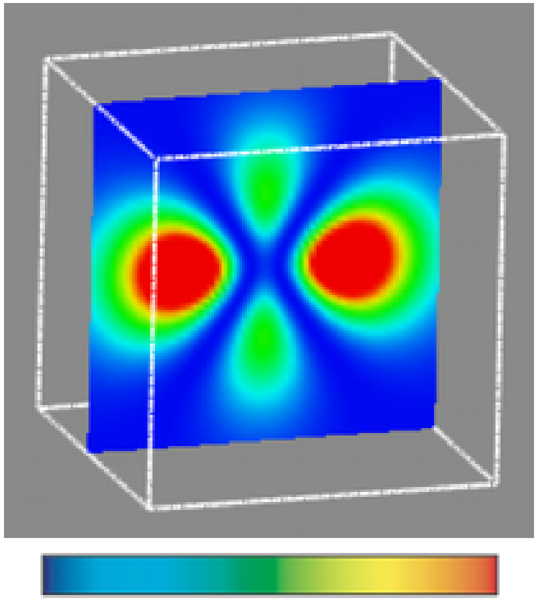
Rainbow colormap
- red = hard tissues (bone)
- blue = air
- other colors = soft tissues

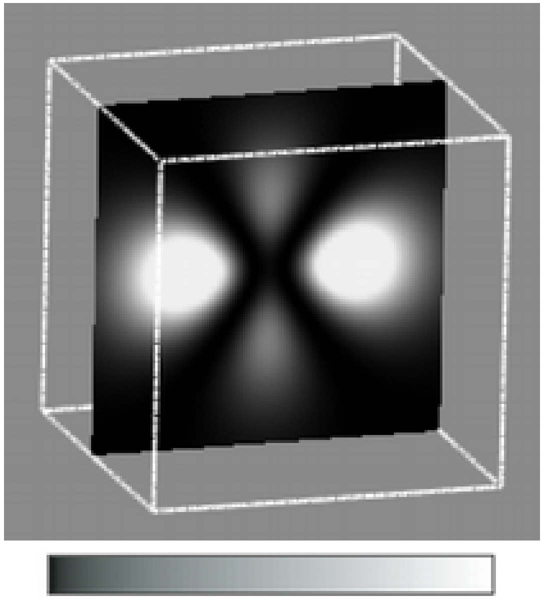


Colormap comparison

2D slice in 3D hydrogen atom potential field







Heat colormap
- maxima highlighted well
- lower values better separable than with gray-value colormap

Heat colormap
- maxima not prominent
- lower values better
- separable

Gray-value colormap
- maxima are highlighted well
- lower values are unclear

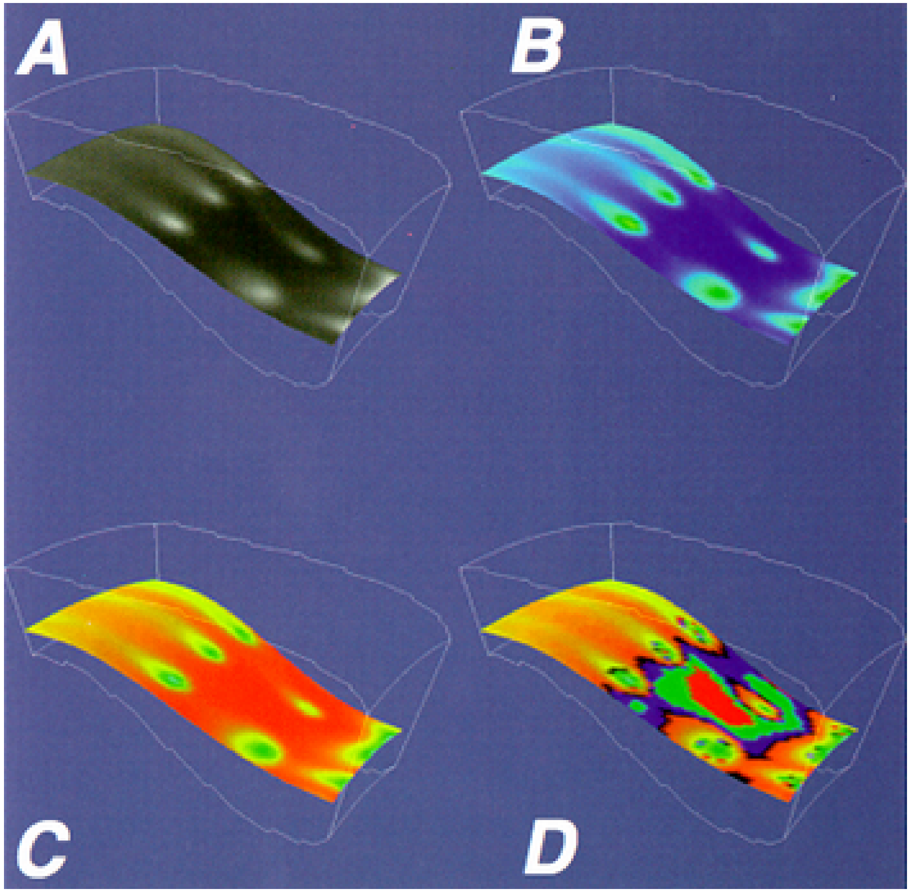
Which is the better colormap? Depends on the application context!

Colormap comparison

2D slice in 3D pressure field in an engine

A. Gray-value colormap
- maxima highlighted well
- low-contrast

B. Purple-to-green colormap
- maxima highlighted well
- good high-low separation



C. Red-to-green colormap
- luminance not used
- color-blind problems..

D. 'Random'
- equal-value zones visible
- little use for the rest

Which is the better colormap? Depends on the application context!

Colormap design techniques

We cannot give universal design rules

- but some technical guidelines/tricks still exist

1. Fully use the perceptual spectrum

- colormap entries should differ in more, rather than less, HSV components

scalar value ~ V; H,S not used

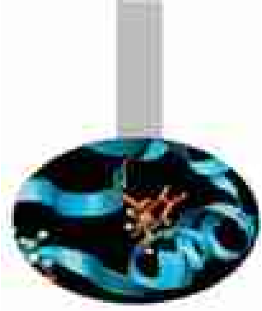
scalar value ~ H; S,V not used

scalar value ~ H,V; S not used

2. Colormap should be easily invertible

- avoid colormap entries with
  - similar HSV entries
  - which are *perceived* as similar (see color blindness issues)
  - which are hard to perceive (e.g. dark or strongly desaturated colors)

Good design guidelines: www.colorbrewer.org

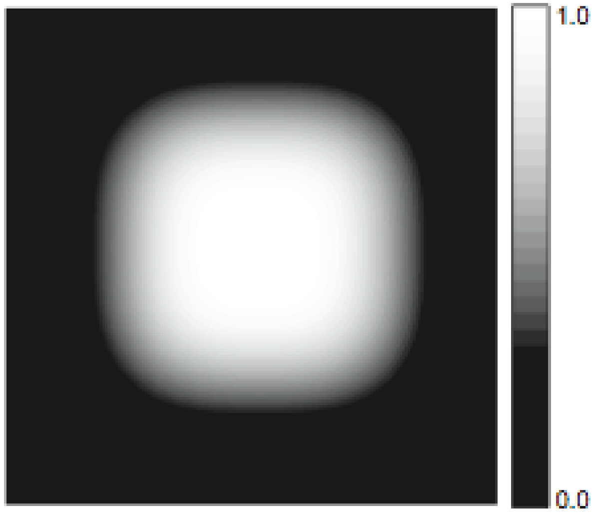


Colormap design techniques

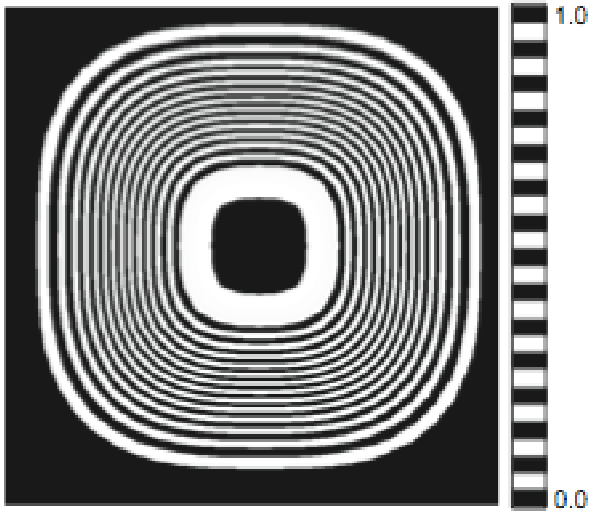
3. Design based on what you *need* to emphasize

- specific value ranges
- specific values
- value change rate (1st derivative of scalar data)
- …

2D function $f(x,y)=e^{-10(x^4+y^4)}$



Gray-scale colormap
- highlights plateaus
- value transitions hard to see



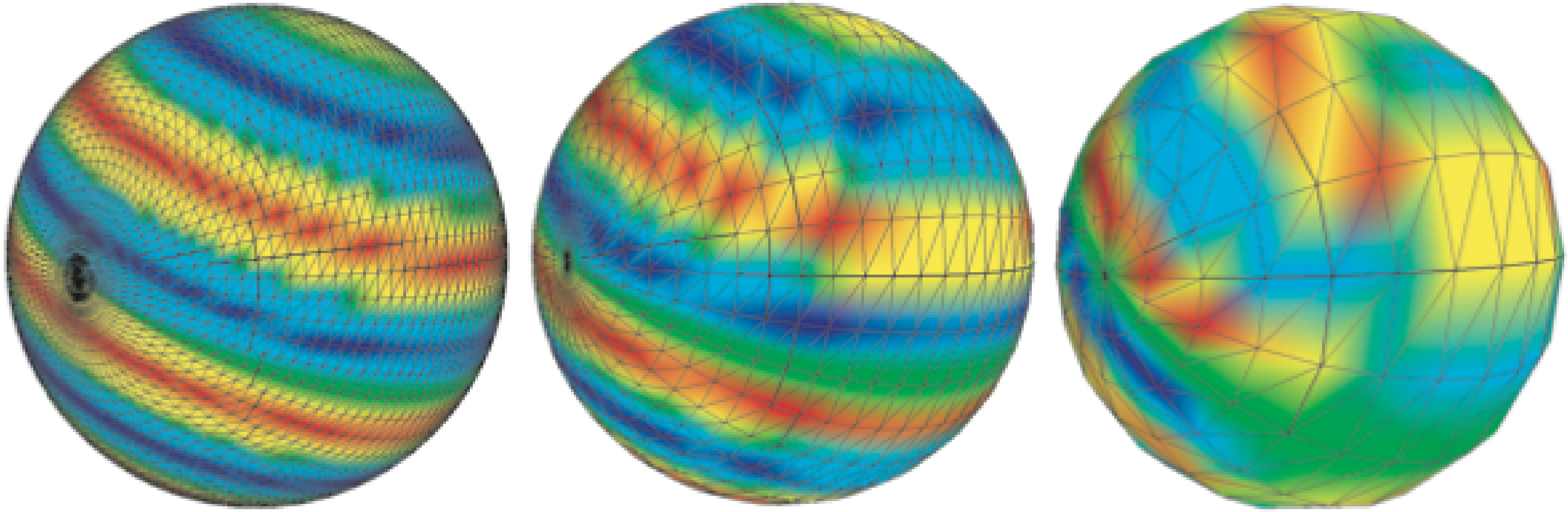
Zebra colormap
- highlights value variations (1st derivative)
- dense, thin bands: fast variation
- thick bands: slow variation

Colormap implementation details

Where to apply the colormap?

- per grid-cell vertex

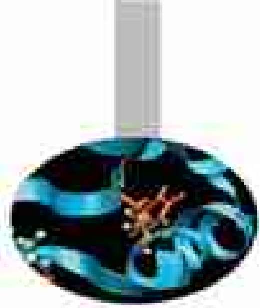
2D periodic high-frequency function



64x64 points 32x32 points 16x16 points

As we decrease the sampling frequency, strong colormapping artifacts appear

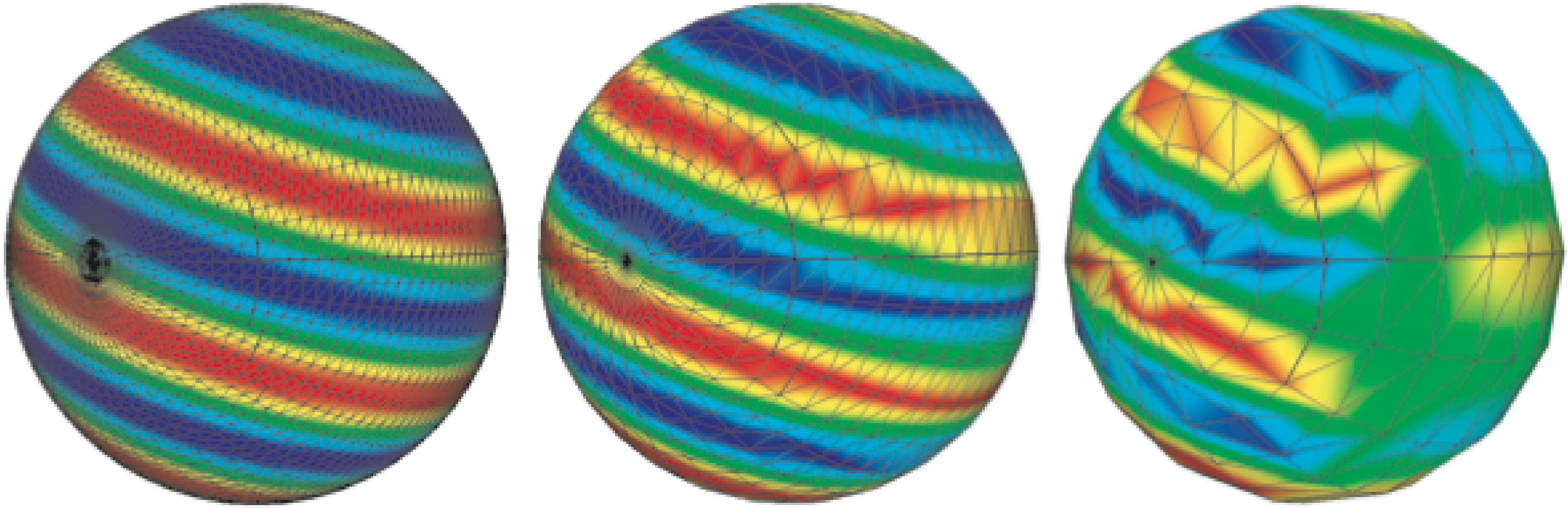
Colormap implementation details



Where to apply the colormap?

- per pixel drawn – better results than per-vertex colormapping
- done using 1D textures

2D periodic high-frequency function



64x64 points 32x32 points 16x16 points

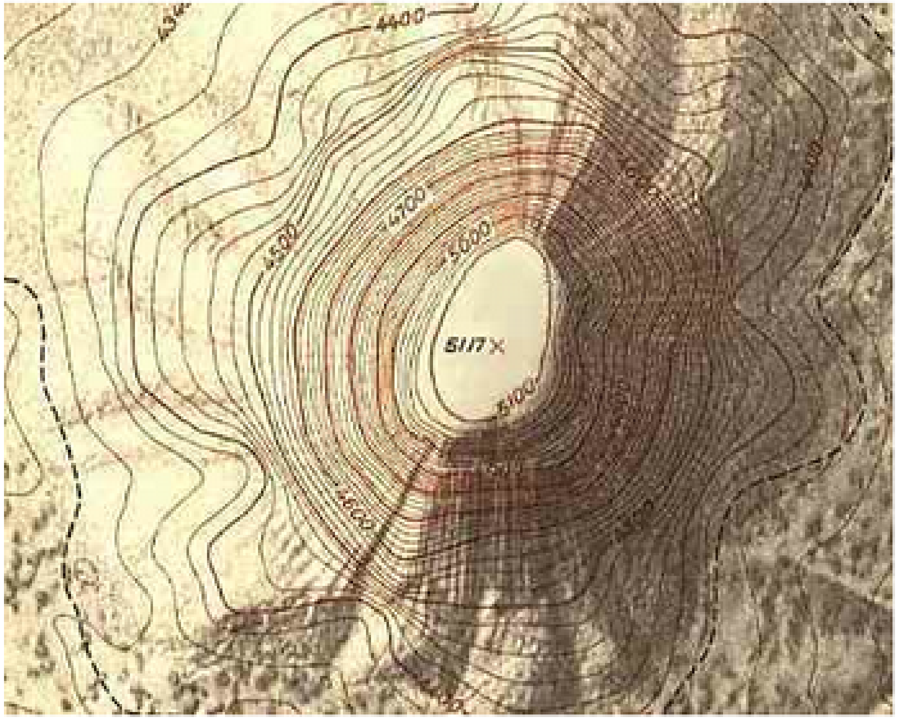
Explanation

- per-vertex: $f \to c(f) \to$ interpolation$(c(f))$ color interpolation can fall outside colormap!
- per-pixel: $f \to$ interpolation$(f) \to c($interpolation$(f))$ colors always stay in colormap

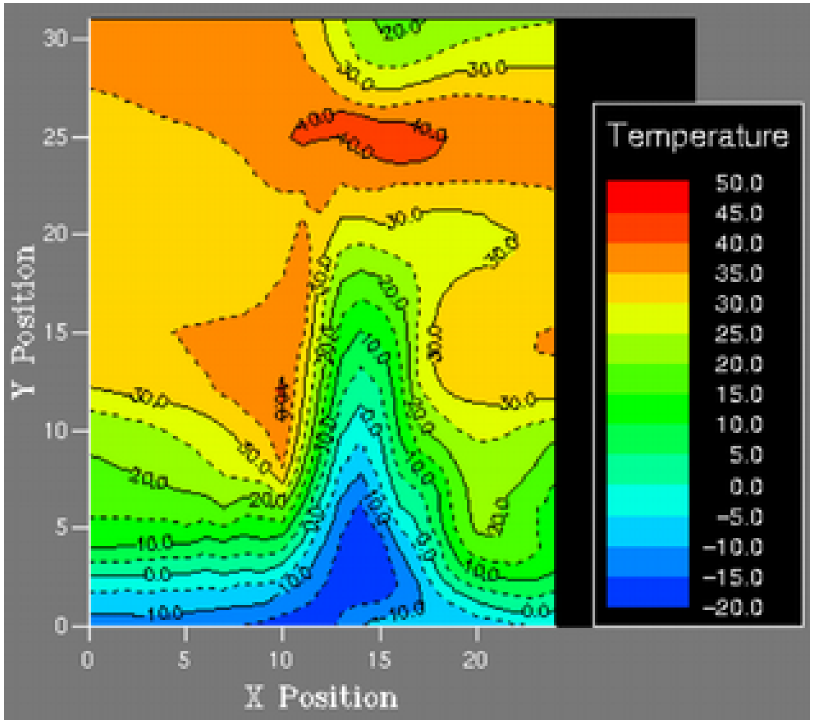
Contouring

Contours are known for hundreds of years in cartography

- also called *isolines* (‘lines of equal value’)

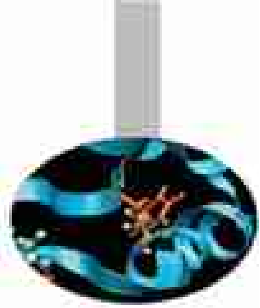


hand-drawn contours on geographical map



computer-generated contours of temperature map

Contour properties



Definition

$$I(f_0) = \{ x \in D \,|\, f(x) = f_0 \}$$

Contours are always closed curves (except when they exit D)

- why? Recall that f is C^0

Contours never (self-)intersect, thus are nested

- why? Think what would mean if a point belonged to two *different* contours

Contours cut D into values smaller resp. larger than the isovalue

- why? Think of definition

Contour properties

Contours are always orthogonal to the scalar value's gradient

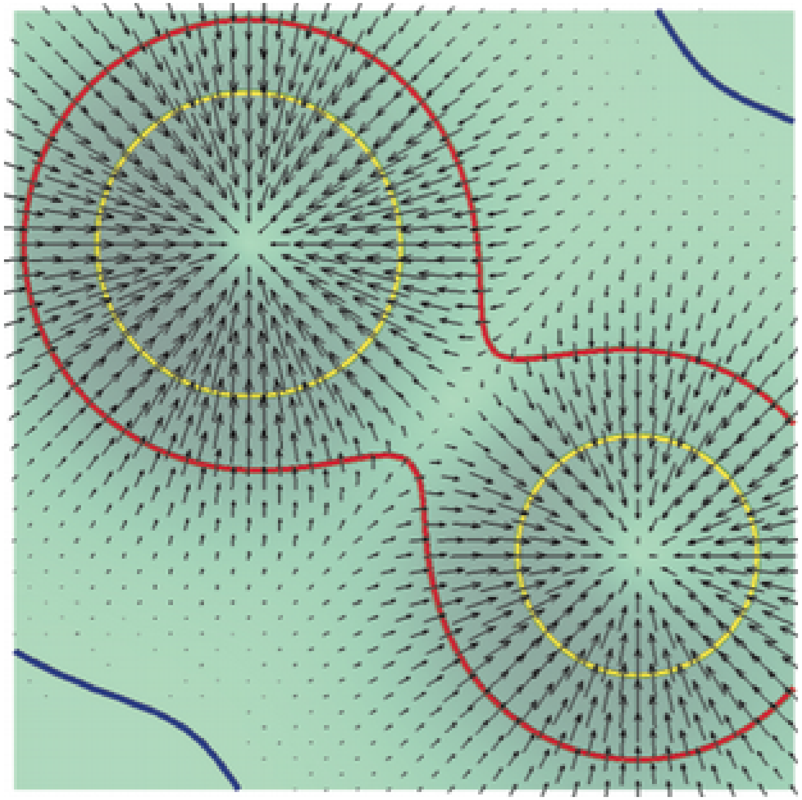
- why? Recall definitions

$$I(f_0)=\{x\in D \mid f(x)=f_0\}$$

contour: $\frac{\partial f}{\partial I}=0$ since f constant along I

$$\nabla f=\left(\frac{\partial f}{\partial x},\frac{\partial f}{\partial y}\right)$$

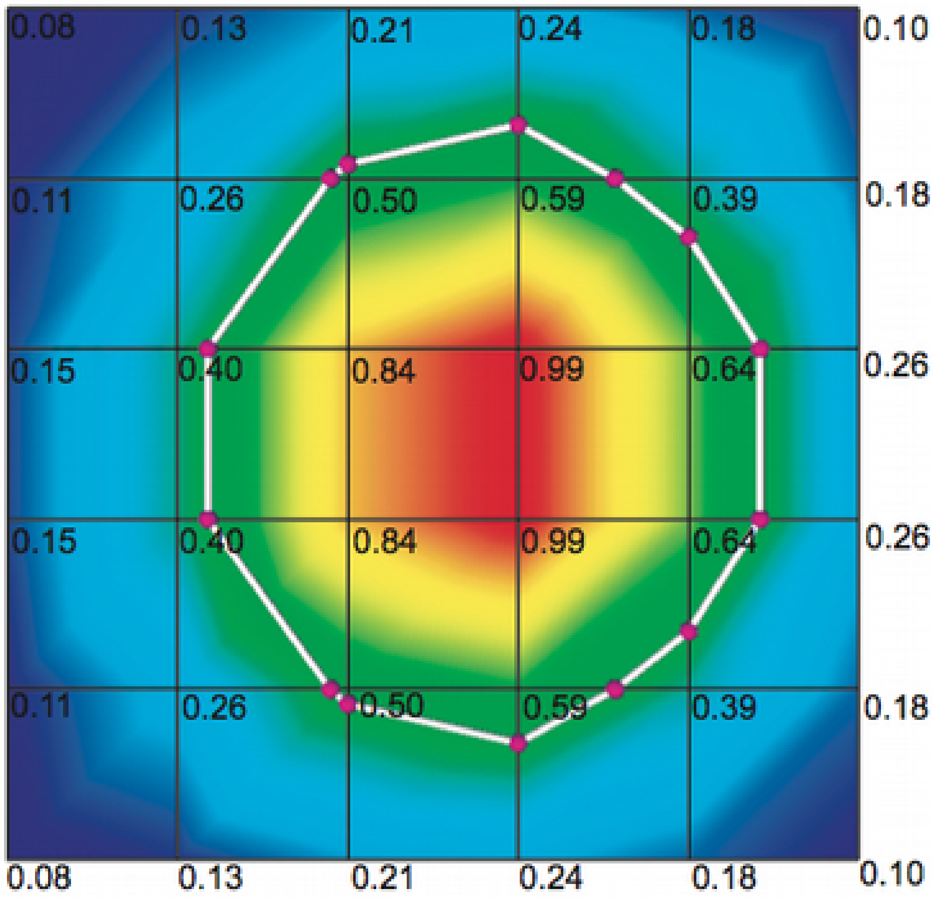
gradient: $\frac{\partial f}{\partial(\nabla f)}=\max$ by definition of gradient



gradient of a scalar field (drawn with arrows) is orthogonal to contours

www.cs.rug.nl/svcg

Basic contouring algorithm



```
for(each cell $c$ in $D$)
{
    $S = \emptyset$        //no contour-edge cuts
    for(each edge $e=(p_i,p_j)$ of $c$)
    {
        if($f_i < v < f_j$)        //$e$ cuts contour
        {
            $q = \frac{p_i(v_j - v) + p_j(v - v_i)}{v_j - v_i}$
            $S = S \cup q$
        }
    }
    connect points in $S$ with lines to build contour;
}
```

Works OK but it is

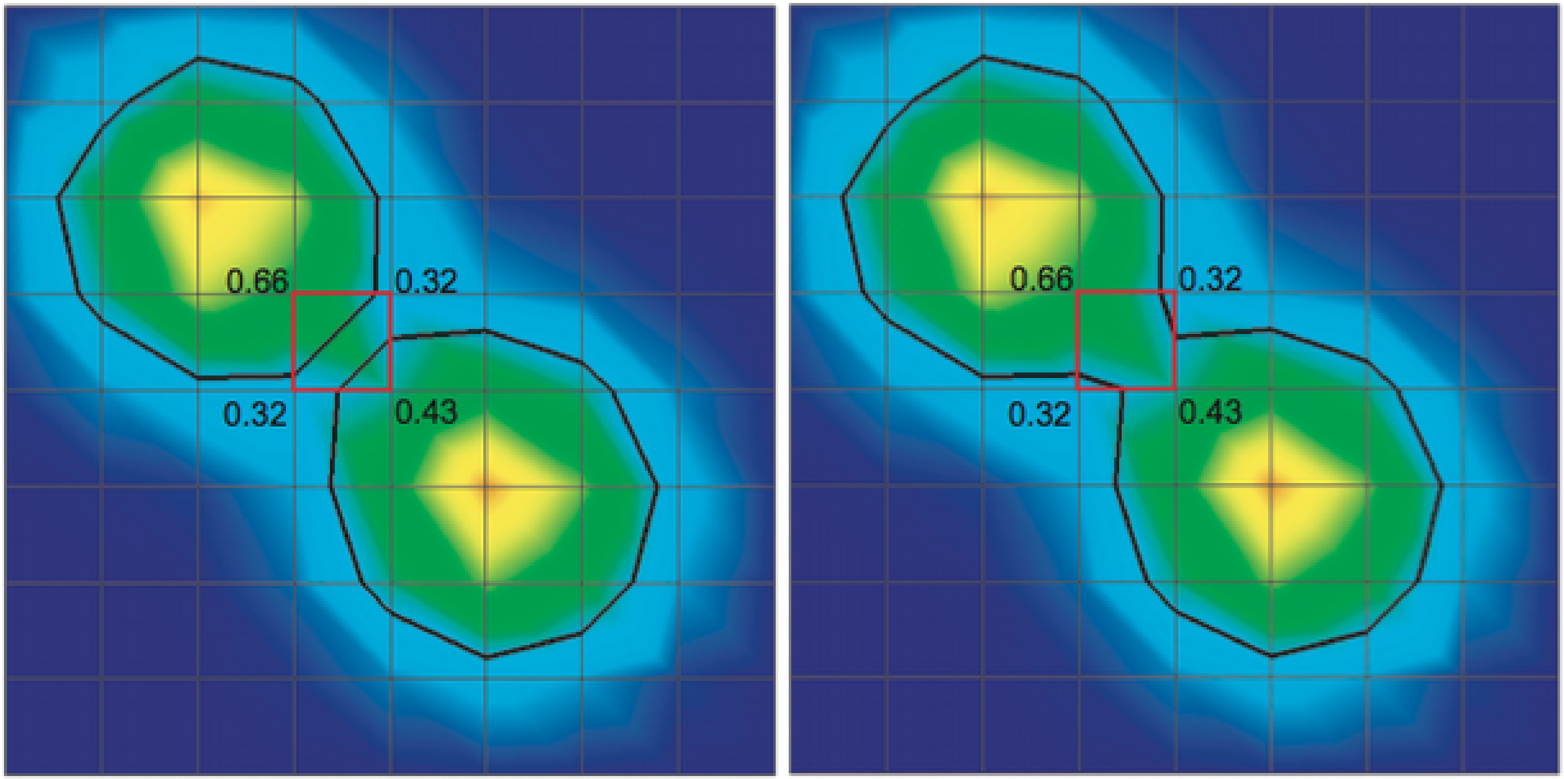
- cumbersome: connecting contour-edge cuts into lines is not trivial to program
- slow: edges intersecting contours are processed twice

Question

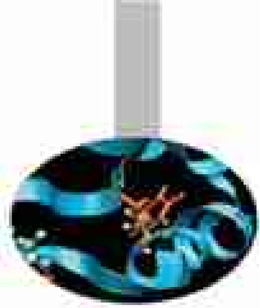
- Are contours piecewise-linear? Why (not)?

Contouring ambiguity

Each edge of the red cell intersects the contour
- which is the right contour result?



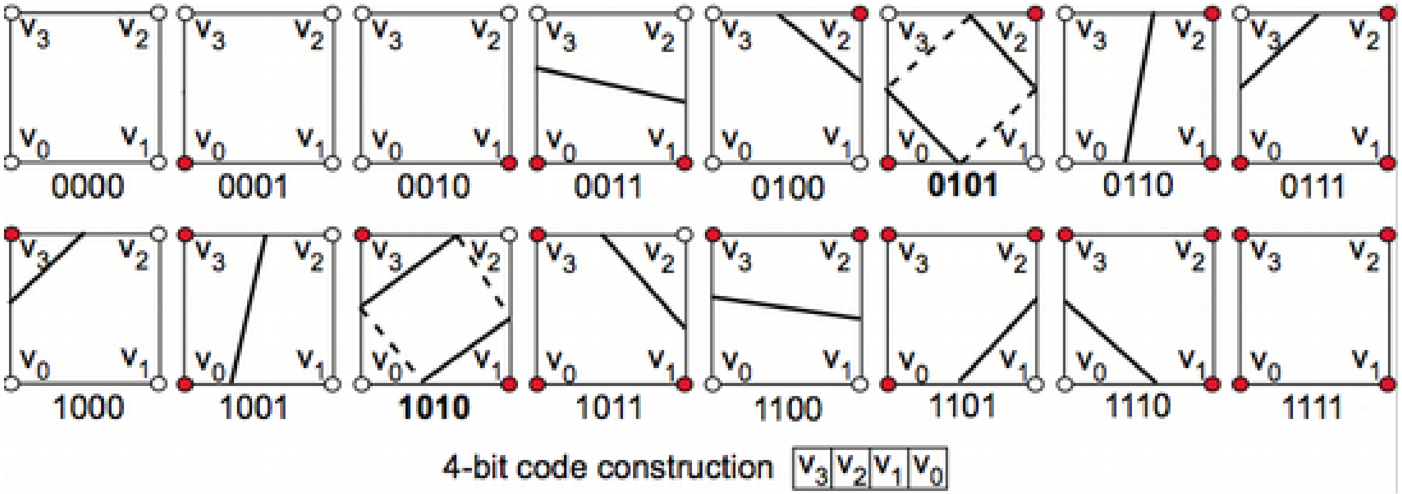
Both answers are equally correct!
- we could discriminate only if we had higher-level information (e.g. topology)
- at cell level, we cannot determine more
- same would happen if we first split quads into triangles (2 splits possible..)



Marching squares

Fast implementation of 2D contouring on quad-cell grids

1. Encode inside/outside state of each vertex w.r.t. contour in a 4-bit code



e.g.
inside: $f > f_0$
outside: $f \leq f_0$

2. Process all dataset cells

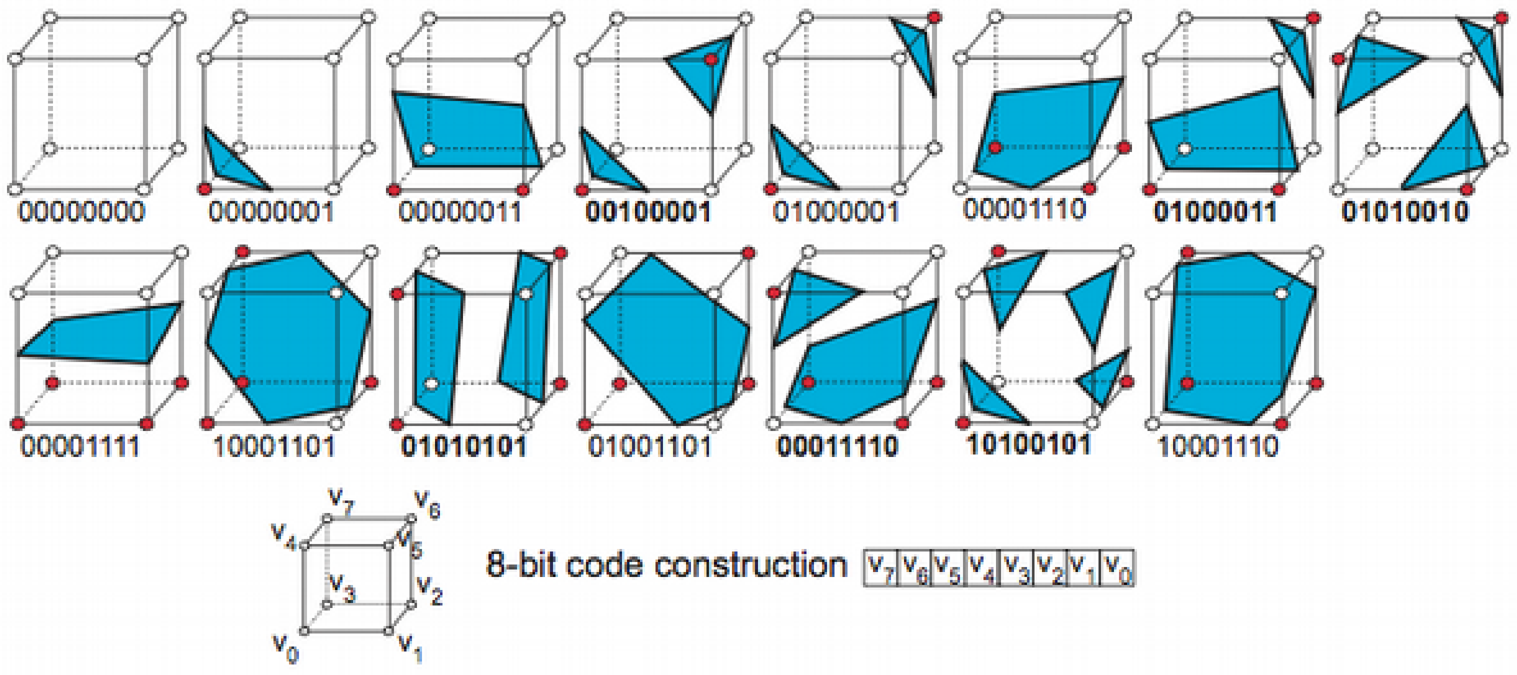
- for each cell, use codes as pointers into a jump-table with 16 cases
- each case has hand-optimized code to
  - compute only the existing edge-contour intersections
  - automatically create required contour segments (connect intersections)
  - reuse already-computed contour segment vertices from previous cells

Note: same can be done for triangles ('marching triangles')

Marching cubes

Fast implementation of 3D contouring (isosurfaces) on parallelepiped-cell grids

1. Encode inside/outside state of each vertex w.r.t. contour in a 8-bit code



e.g.
inside: $f > f_0$
outside: $f \le f_0$

Invented by Bill Lorensen at GE, one of the authors of VTK from Kitware

Marching cubes (cont'd)

- For each case
  - compute the cell-contour intersection → triangles, quads, pentagons, hexagons
  - triangulate these on-the-fly → triangle output only

3. Treat ambiguous cases

- 6 such cases (see **bold**-coded figures on previous slide)
- harder to solve than in 2D (need to prevent false cracks in the surface)
- see Sec. 5.3 for algorithmic details

4. Compute isosurface normals

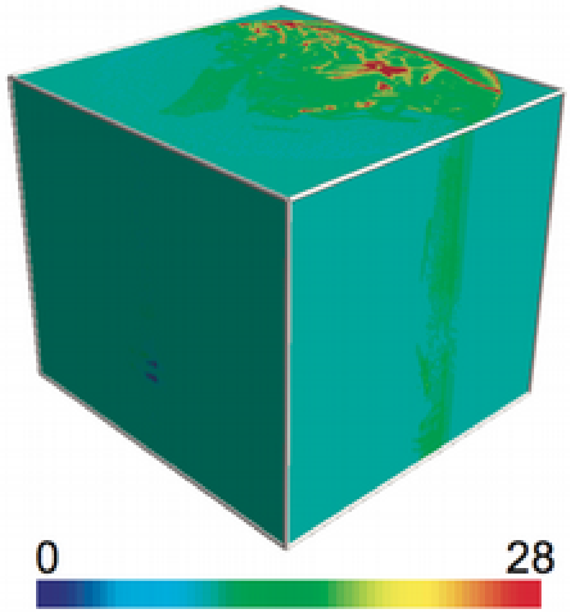
- by face-to-vertex normal averaging (see Module 2, Data resampling)
- directly from data

$$\forall x \in I,\ n_I(x) = -\frac{\nabla f(x)}{\|\nabla f(x)\|}$$

(gradient is normal to contours, see previous slides)

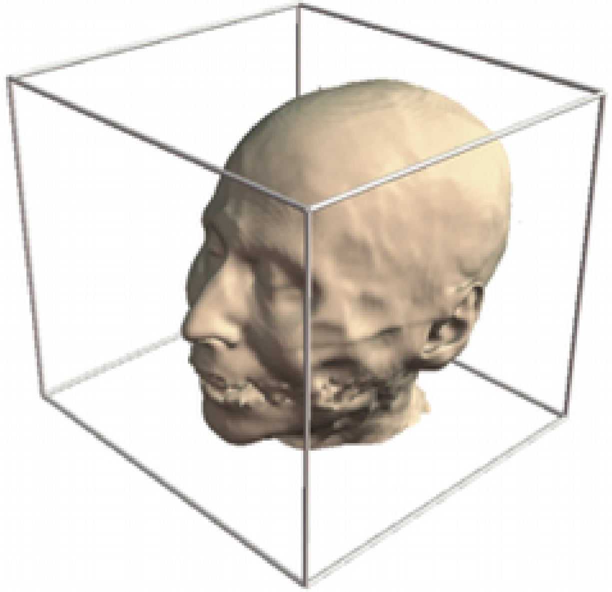
5. Draw resulting surface as a (shaded) unstructured triangle mesh

Marching cubes

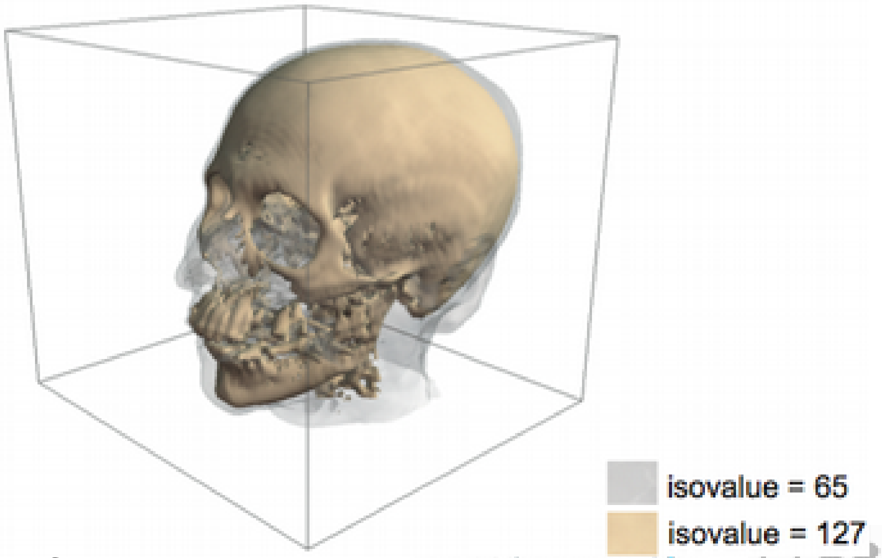


scalar CT volume
(tissue density)

isosurfaces



isosurface for scalar value
corresponding to skin

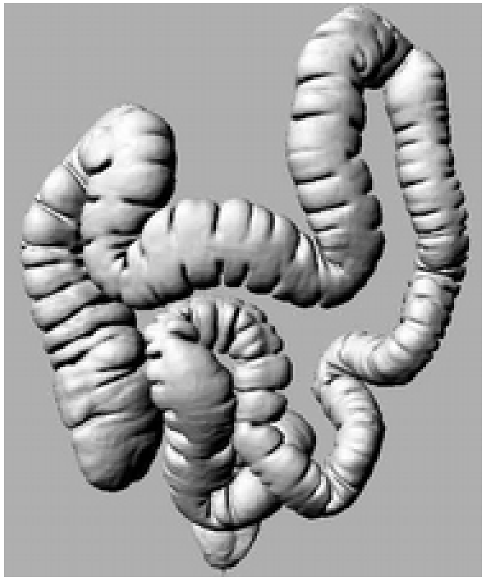


isosurfaces for skin and bone

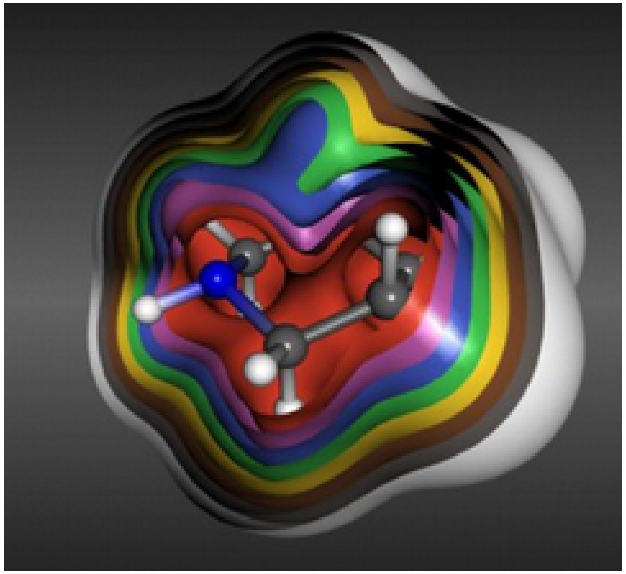
- extremely simple to use tool
- insightful results

www.cs.rug.nl/svcg

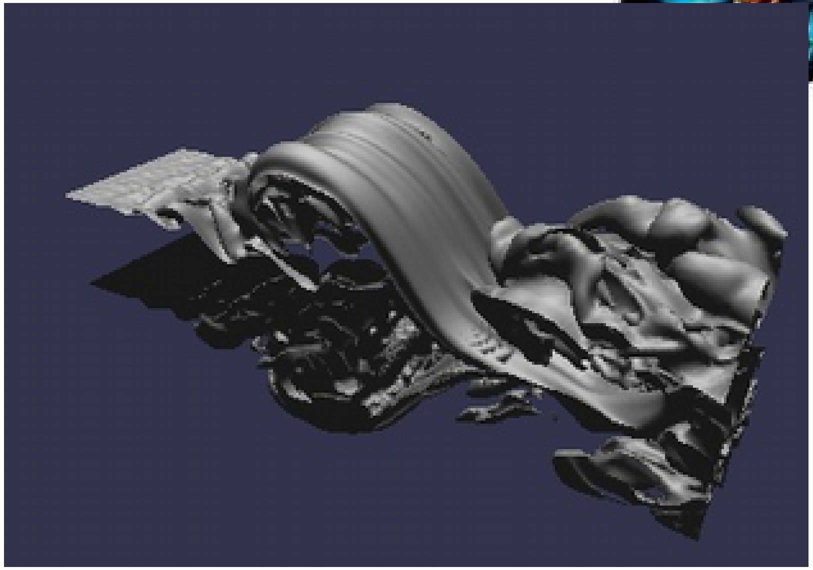
Isosurface examples



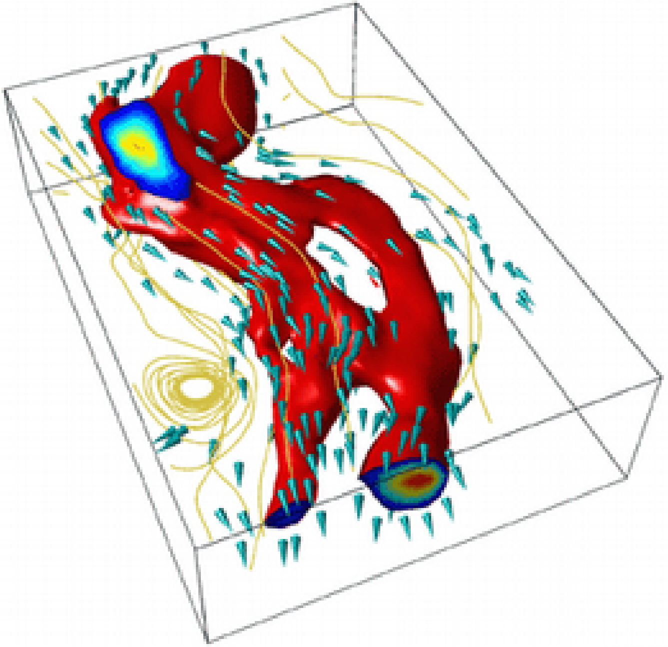
colon (CT dataset)



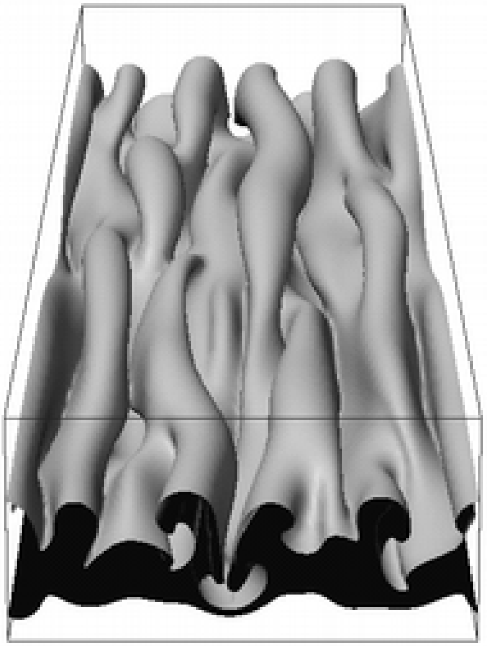
electron density in molecule



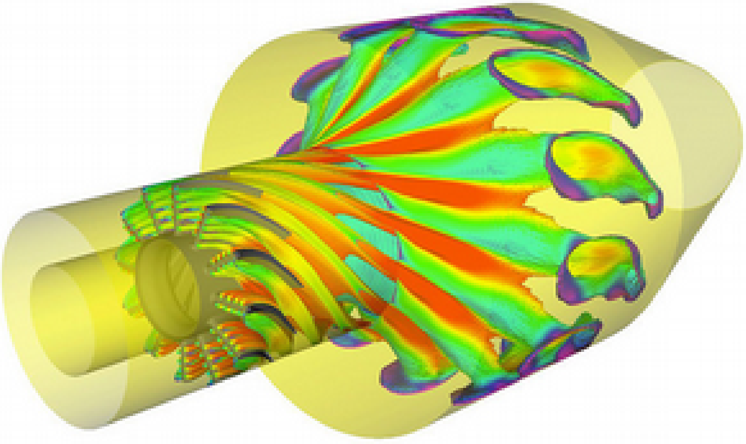
velocity in 3D fluid flow



velocity in 3D fluid flow

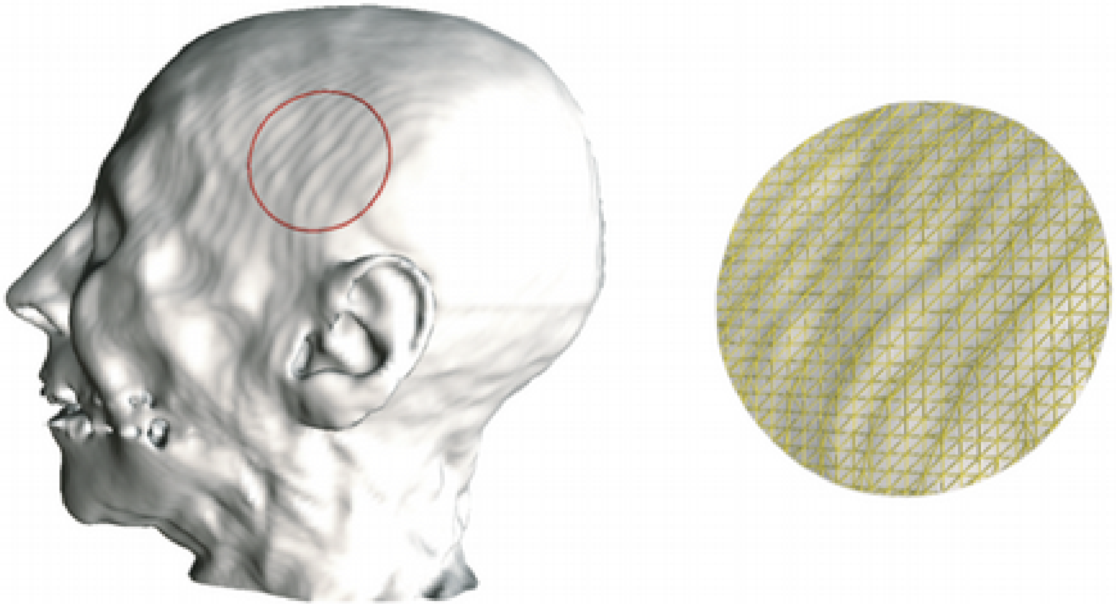


magnetic field in sunspots



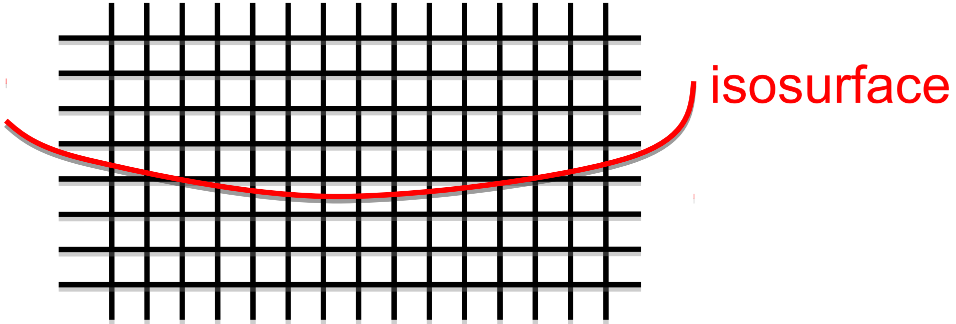
fuel concentration, colored
by temperature in jet engine

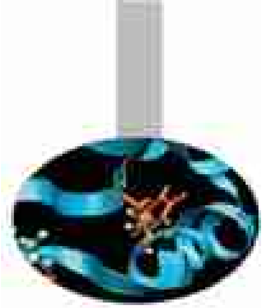
Marching cubes – technical points



Does this person have wavy wrinkles on his head’s skin?

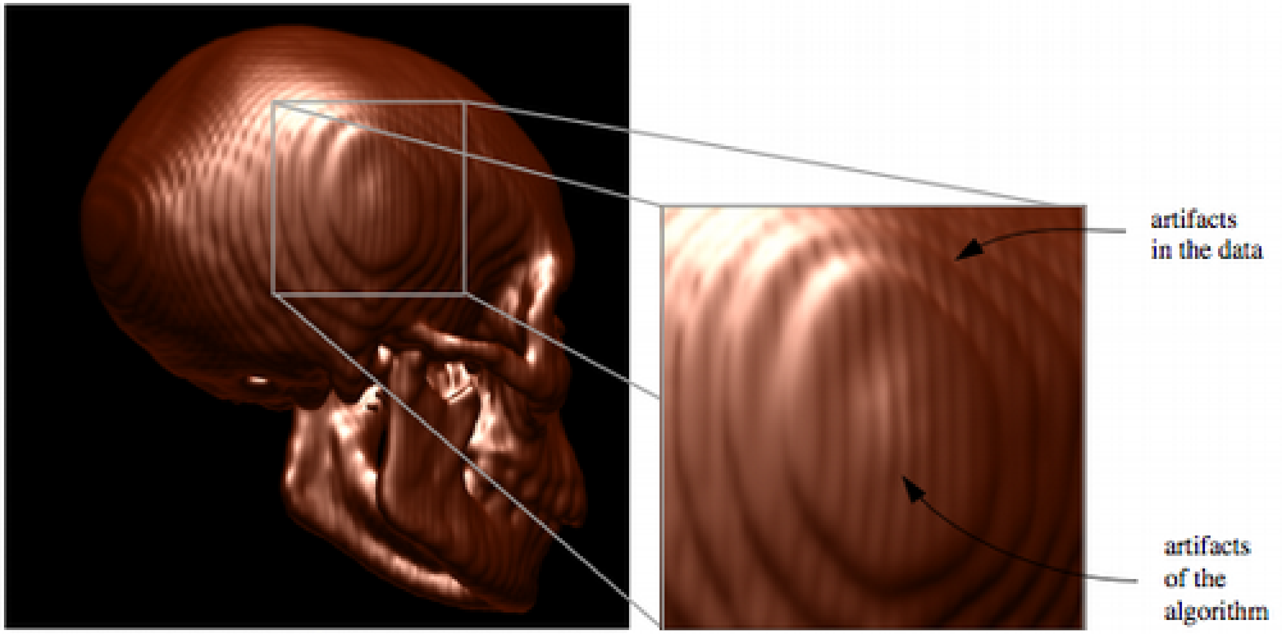
- so it looks from the visualization…
- these are so-called ‘ringing artifacts’
    - due to the near-tangent orientation of the isosurface w.r.t. finite-resolution volume grid

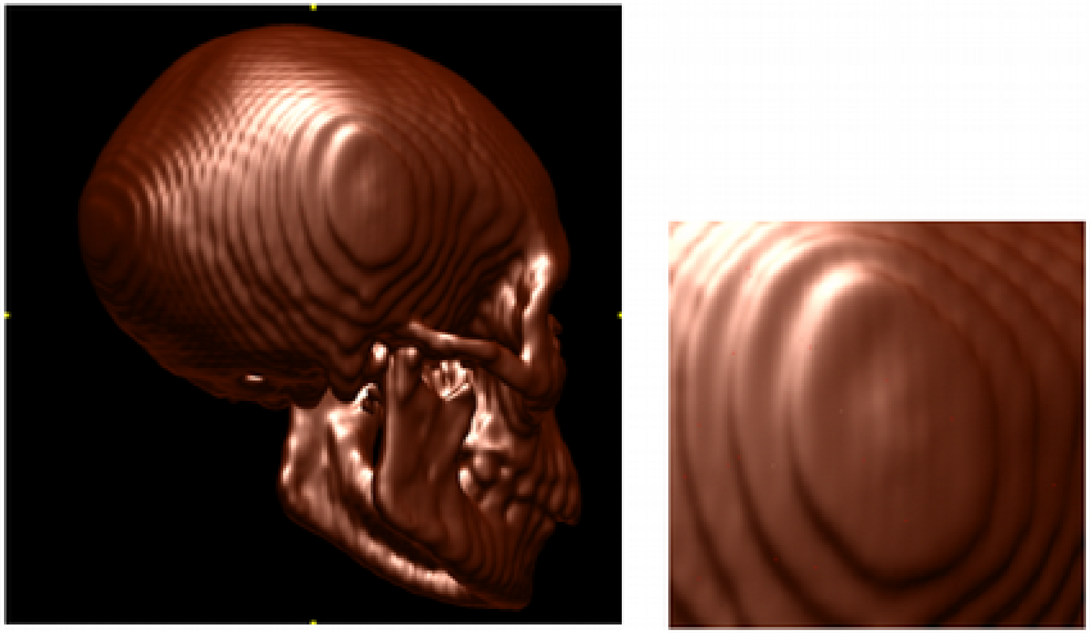




Marching cubes – technical points

A closer look at ringing artifacts





Two kinds of artifacts

- from data: cannot remove easily
- from algorithm (due to linear interpolation)

Removing algorithm artifacts

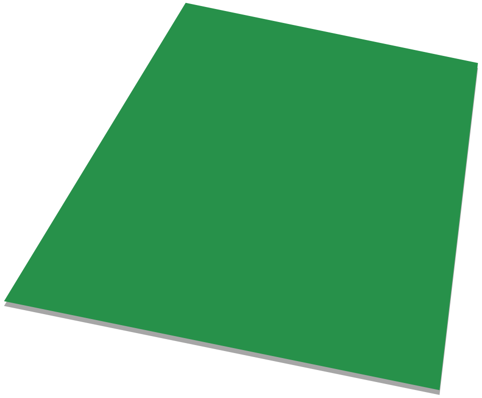
- use higher-order interpolants (e.g. splines)

E. C. LaMar, B. Hamann, K. Joy, High-Quality Rendering of Smooth Isosurfaces, JVCA vol. 10, 1999, 79-90

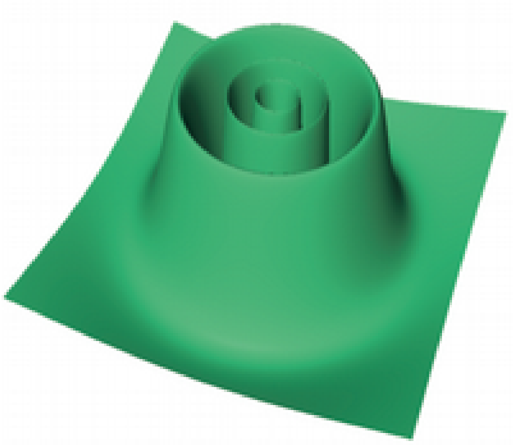
Height / displacement plots

Displace a given surface $S \subseteq D$ in the direction of its normal
Displacement value encodes the scalar data f

$$S_{displ}(x) = x + n(x)\,f(x), \quad \forall x \in S$$



input surface S

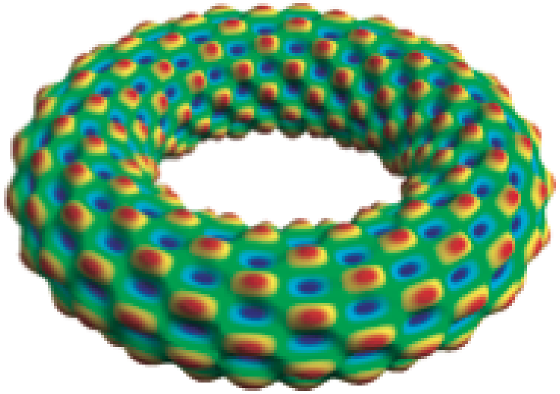


displaced surface S_{displ}

Height plot

- $S = xy$ plane
- displacement always along z

input surface S

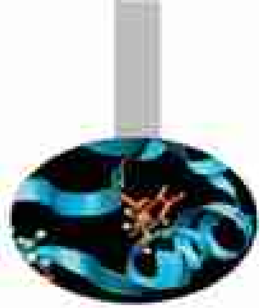


displaced surface S_{displ}

Displacement plot

- S = any surface in $\mathbf{R}^3$
- useful to visualize
  3D scalar fields

Vector algorithms



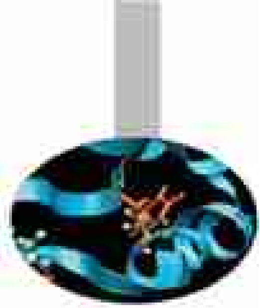
1. **Scalar derived quantities**
    - divergence, curl, vorticity

2. **0-dimensional shapes**
    - hedgehogs and glyphs
    - color coding

3. **1-dimensional and 2-dimensional shapes**
    - displacement plots
    - stream objects

4. **Image-based algorithms**
    - image-based flow visualization in 2D, curved surfaces, and 3D

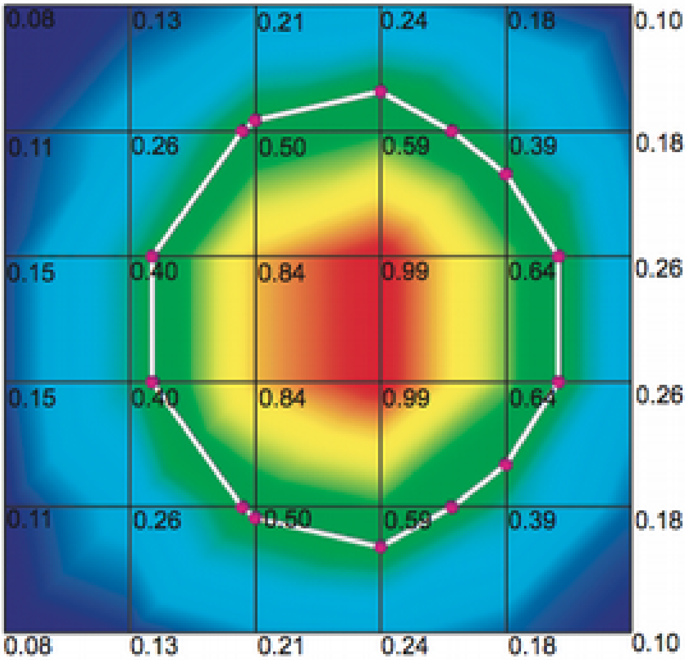
Basic problem



Input data

- vector field $v : D \rightarrow \mathbf{R}^n$
- domain D 2D planar surfaces, 2D surfaces embedded in 3D, 3D volumes
- variables $n=2$ (fields tangent to 2D surfaces) or $n=3$ (volumetric fields)

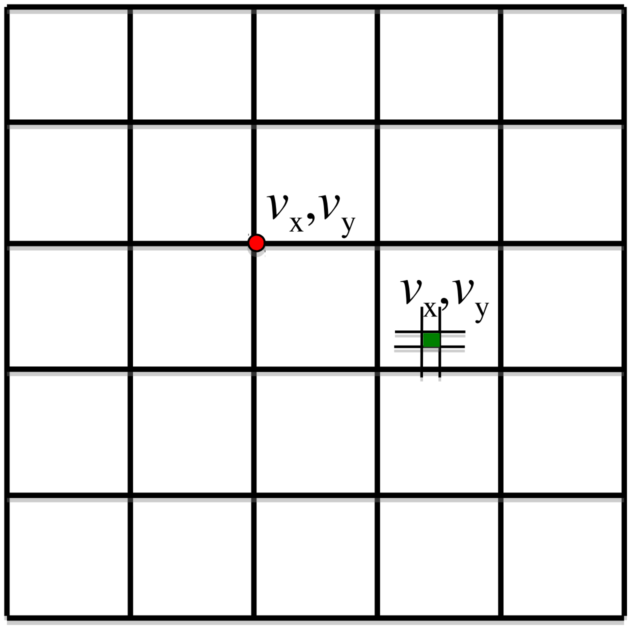
Challenge: comparison with scalar visualization



0.08	0.13	0.21	0.24	0.18	0.10
0.11	0.26	0.50	0.59	0.39	0.18
0.15	0.40	0.84	0.99	0.64	0.26
0.15	0.40	0.84	0.99	0.64	0.26
0.11	0.26	0.50	0.59	0.39	0.18
0.08	0.13	0.21	0.24	0.18	0.10

Scalar visualization

- challenge is to map D to 2D screen
- after that, we have 1 pixel per scalar value



Vector visualization

- challenge is to map D to 2D screen
- after that, we have
- 1 pixel for 2 or 3 scalar values!

First solution: Reuse scalar visualization

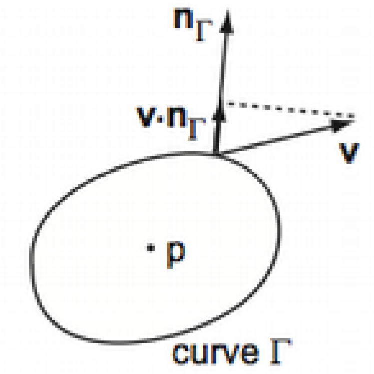
- compute derived scalar quantities from vector fields
- use known scalar visualization methods for these

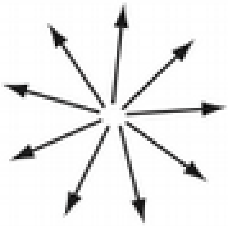
1.Divergence

- think of vector field as encoding a fluid flow
- intuition: amount of mass (air, water) created, or absorbed, at a point in D
- given a field $\mathbf{v} : \mathbf{R}^3 \to \mathbf{R}^3$, $\text{div } \mathbf{v} : \mathbf{R}^3 \to \mathbf{R}$ is

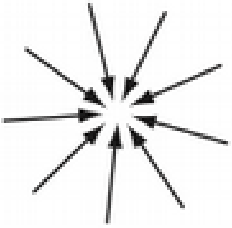
$$\text{div } \mathbf{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}$$ equivalent to $$\text{div } \mathbf{v} = \lim_{\Gamma \to 0} \frac{1}{|\Gamma|} \int_{\Gamma} (\mathbf{v} \cdot \mathbf{n}_{\Gamma}) ds$$

$\text{div } \mathbf{v}$ is sometimes denoted as $\nabla \times v$

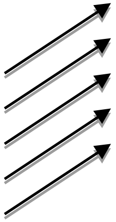




source
$\text{div } \mathbf{v}>0$



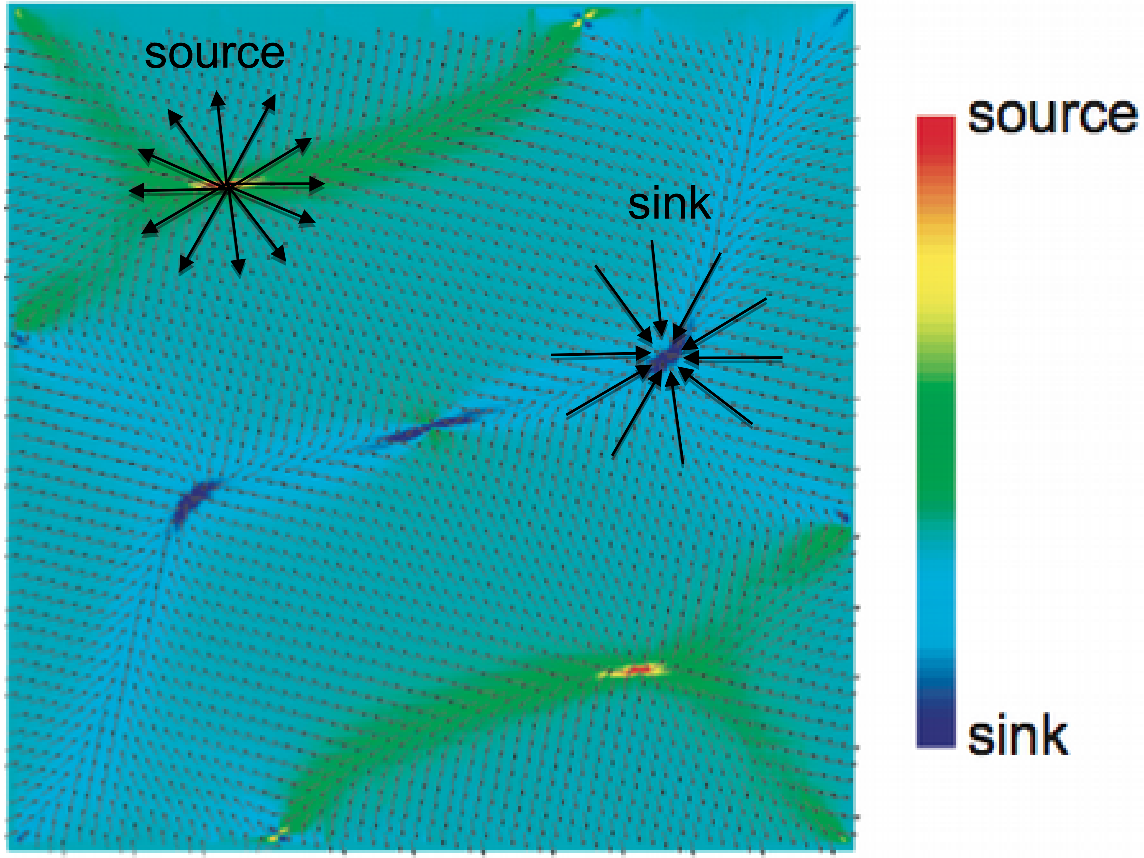
sink
$\text{div } \mathbf{v}<0$



laminar flow
$\text{div } \mathbf{v}=0$

Divergence: Reuse scalar visualization

- compute using definition with partial derivatives
- visualize using e.g. color mapping



- gives a good impression of where the flow ‘enters’ and ‘exits’ some domain

Curl

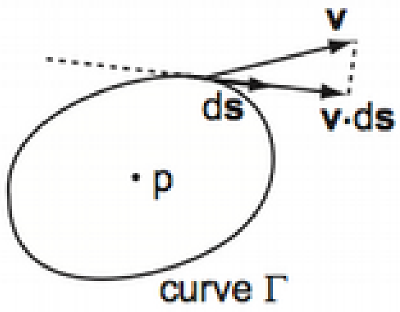
2. Curl (also called rotor)

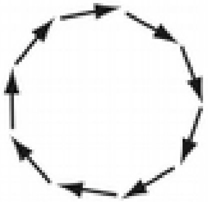
- consider again a vector field as encoding a fluid flow
- intuition: how quickly the flow 'rotates' around each point?
- given a field $\mathbf{v} : \mathbf{R}^3 \to \mathbf{R}^3$, $\mathrm{rot}\ \mathbf{v} : \mathbf{R}^3 \to \mathbf{R}^3$ is

$$\mathrm{rot}\ \mathbf{v} = \left(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z}, \frac{\partial v_x}{\partial z} - \frac{\partial v_z}{\partial x}, \frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y} \right)$$

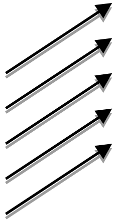
equivalent to

$$\mathrm{rot}\ \mathbf{v} = \lim_{\Gamma \to 0} \frac{1}{|\Gamma|} \int_\Gamma \mathbf{v} \cdot \mathrm{d}\mathbf{s}$$

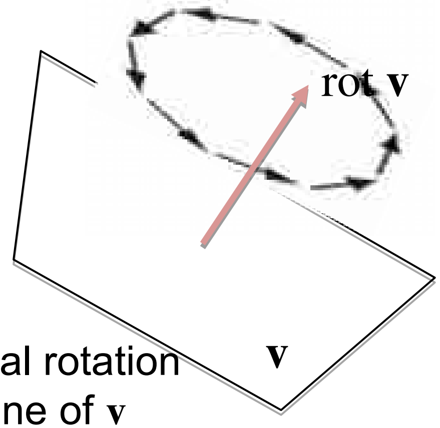




rotational flow
$\|\mathrm{rot}\ \mathbf{v}\| > 0$



laminar flow
$\mathrm{rot}\ \mathbf{v} = \mathbf{0}$

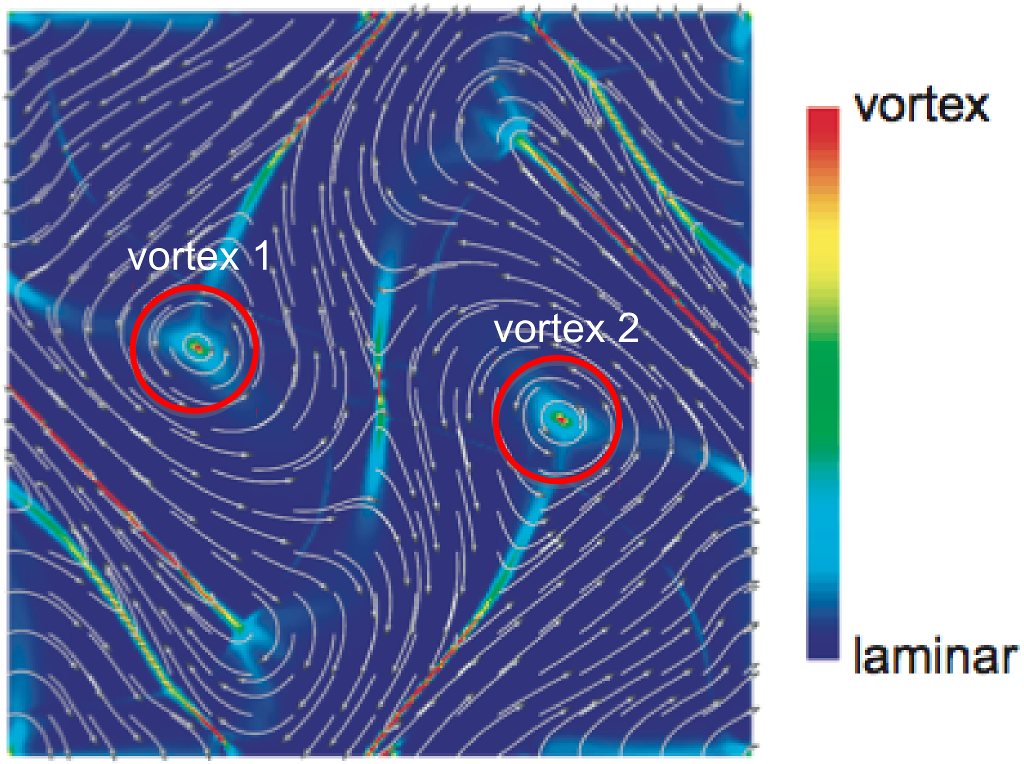


local rotation plane of **v**

$\mathrm{rot}\ \mathbf{v}$ is sometimes denoted as $\nabla \times v$

Curl

- compute using definition with partial derivatives
- visualize magnitude $\|\mathrm{rot}\ \mathbf{v}\|$ using e.g. color mapping

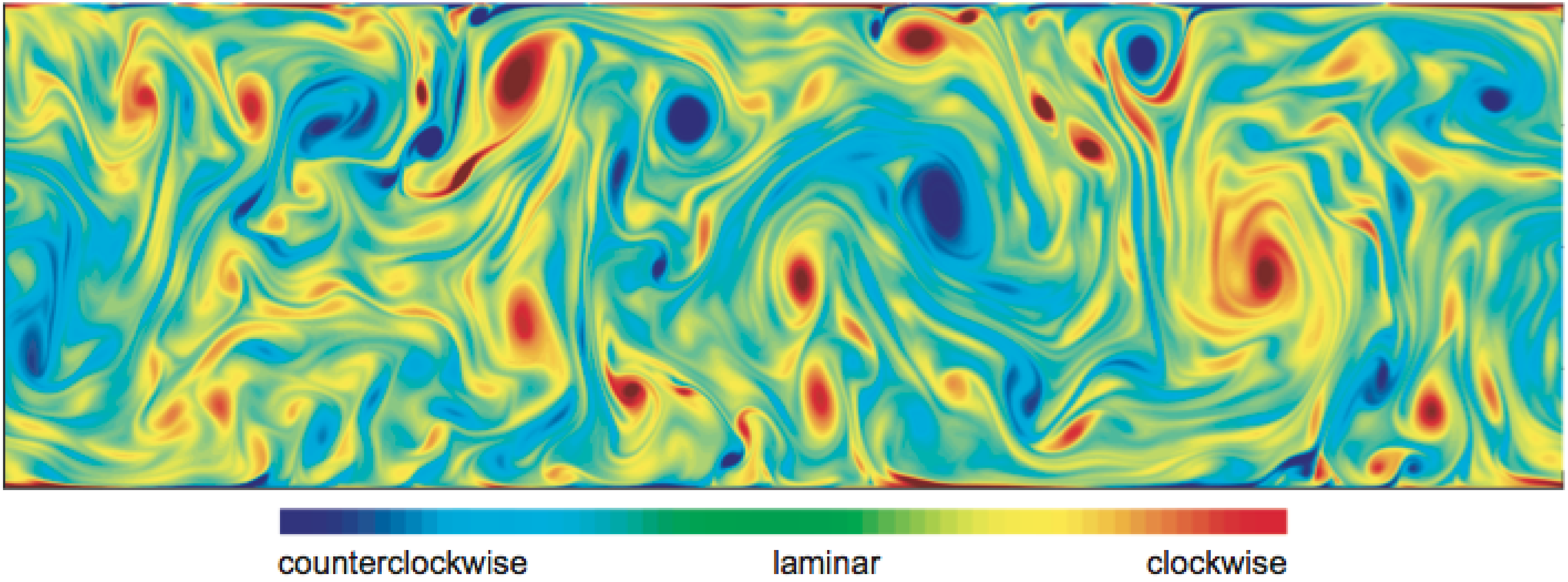


- very useful in practice to find vortices = regions of high vorticity
- these are highly important in flow simulations (aerodynamics, hydrodynamics)

Curl

Example of vorticity

- 2D fluid flow
- simulated by solving Navier-Stokes equations
- visualized using vorticity



- vortices appear at different scales
- ‘pairing’ of vortices spinning in opposite directions

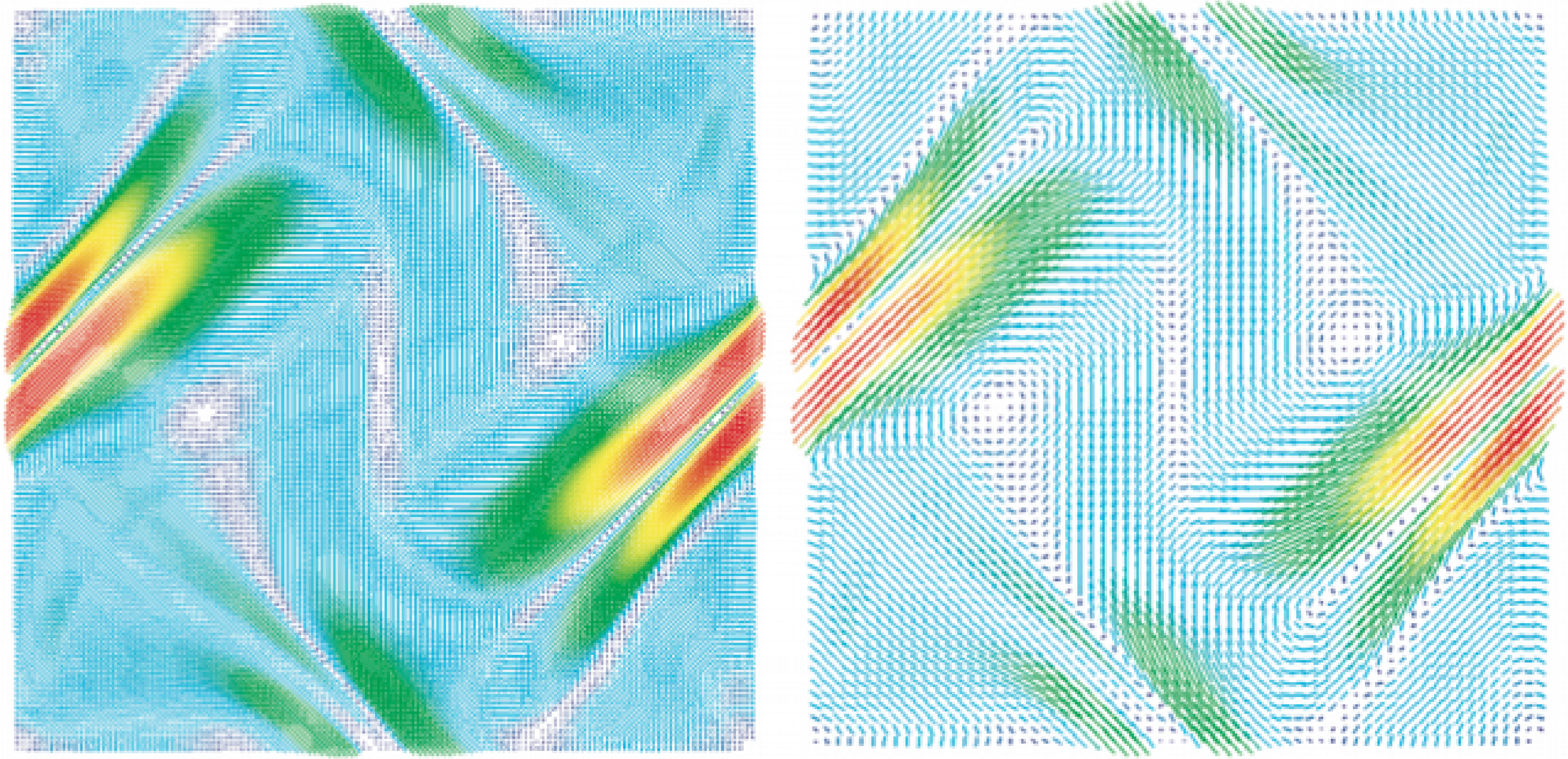
Vector glyphs

Icons, or signs, for visualizing vector fields

- placed by (sub)sampling the dataset domain
- attributes (scale, color, orientation) map vector data at sample points

Simplest glyph: Line segment (hedgehog plots)

- for every sample point $x \in D$
  - draw line $(x, x + k\mathbf{v}(x))$
  - optionally color map $\|\mathbf{v}\|$ onto it

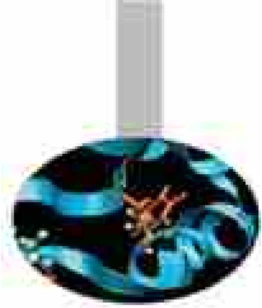


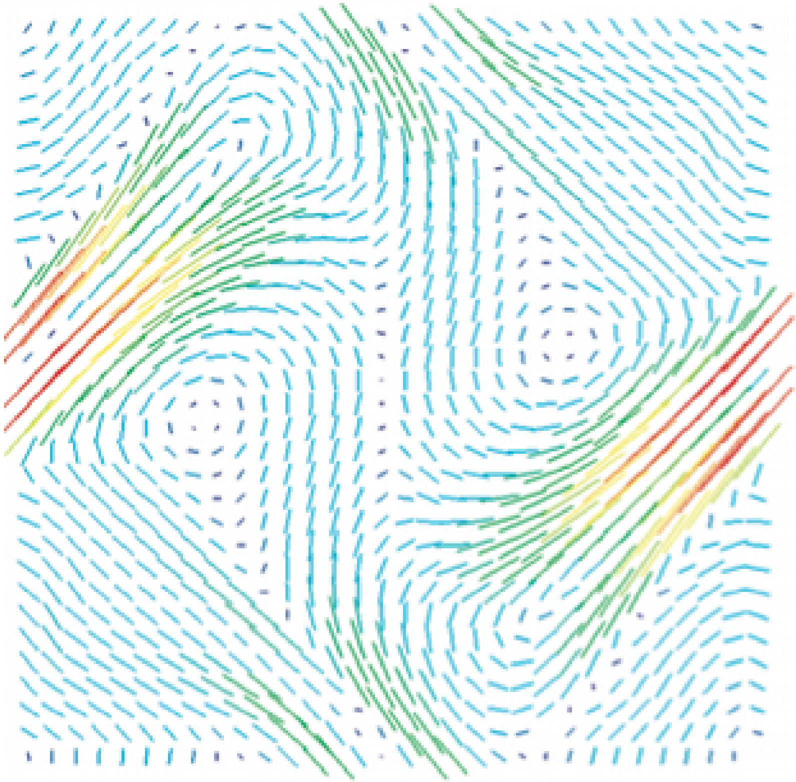
128^2 glyph grid

64^2 glyph grid

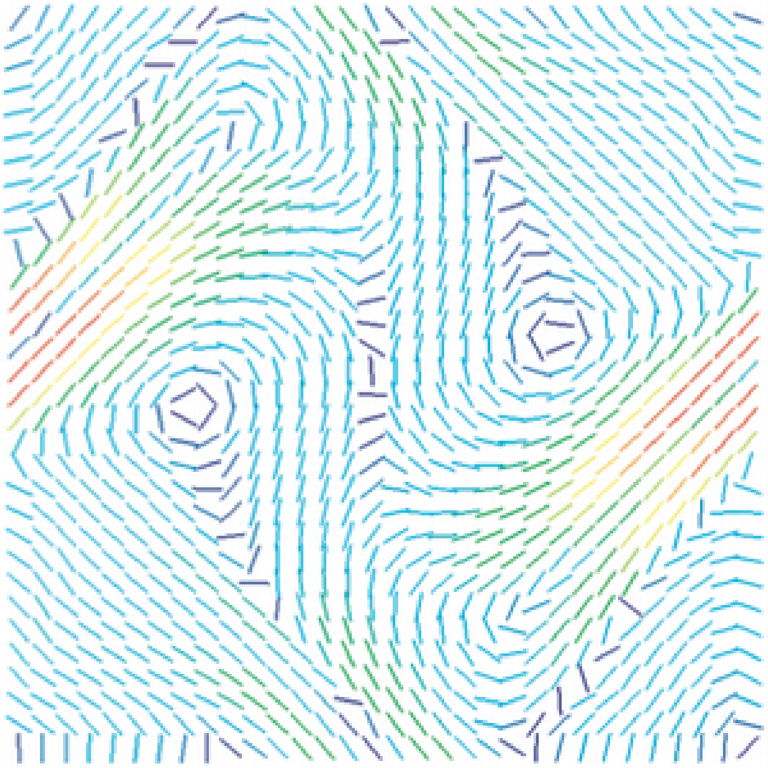
MHD simulation
256^2 grid

Vector glyphs





32^2 glyph grid



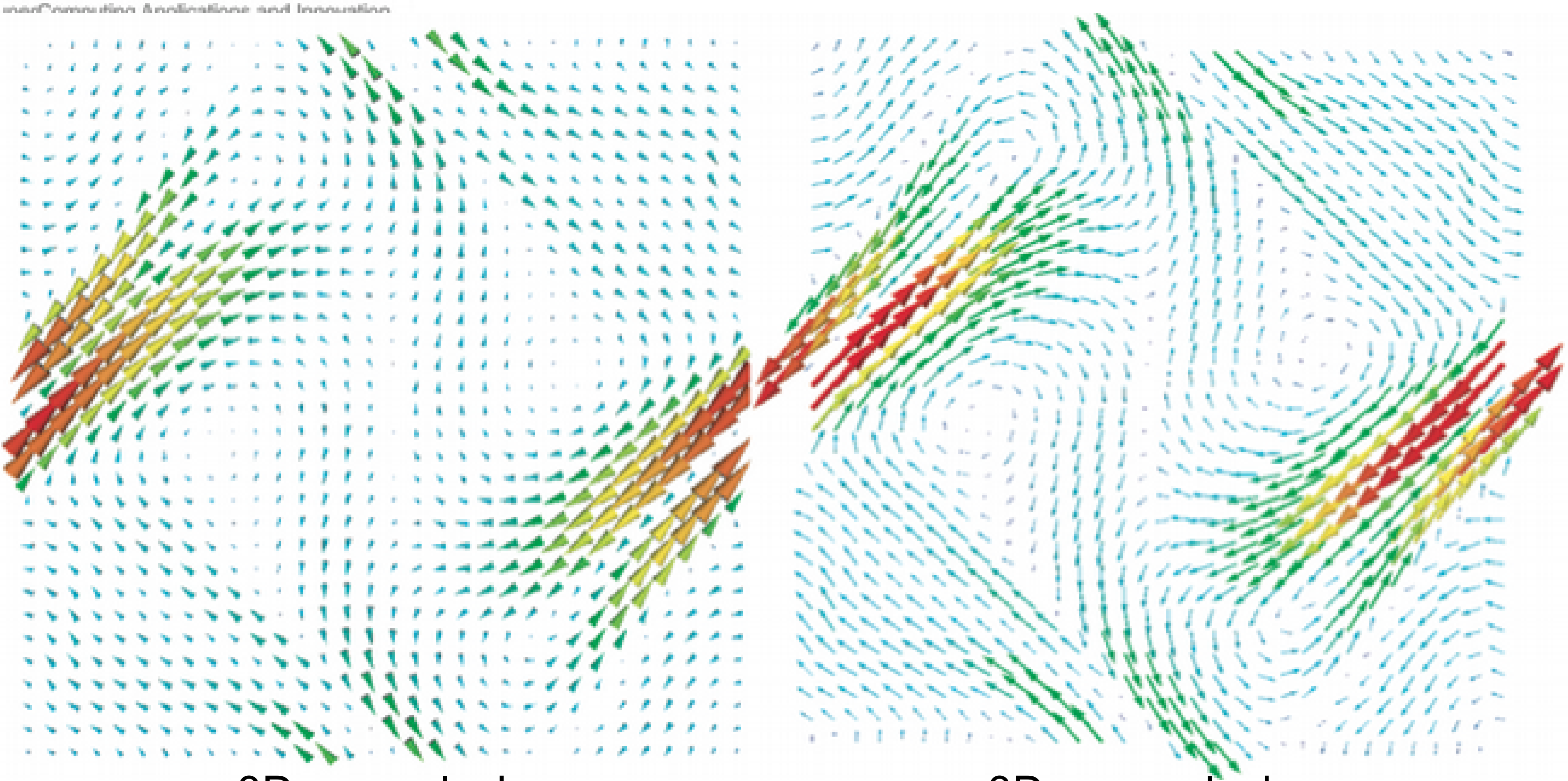
32^2 glyph grid, no line scaling

MHD simulation
256^2 grid

Observations

- trade-offs
  - more samples: more data points depicted, but more potential clutter
  - less samples: less data points depicted, but higher clarity
  - more line scaling: easier to see high-speed areas, but more clutter
  - less line scaling: less clutter, but harder to perceive directions

Vector glyphs



3D cone glyphs

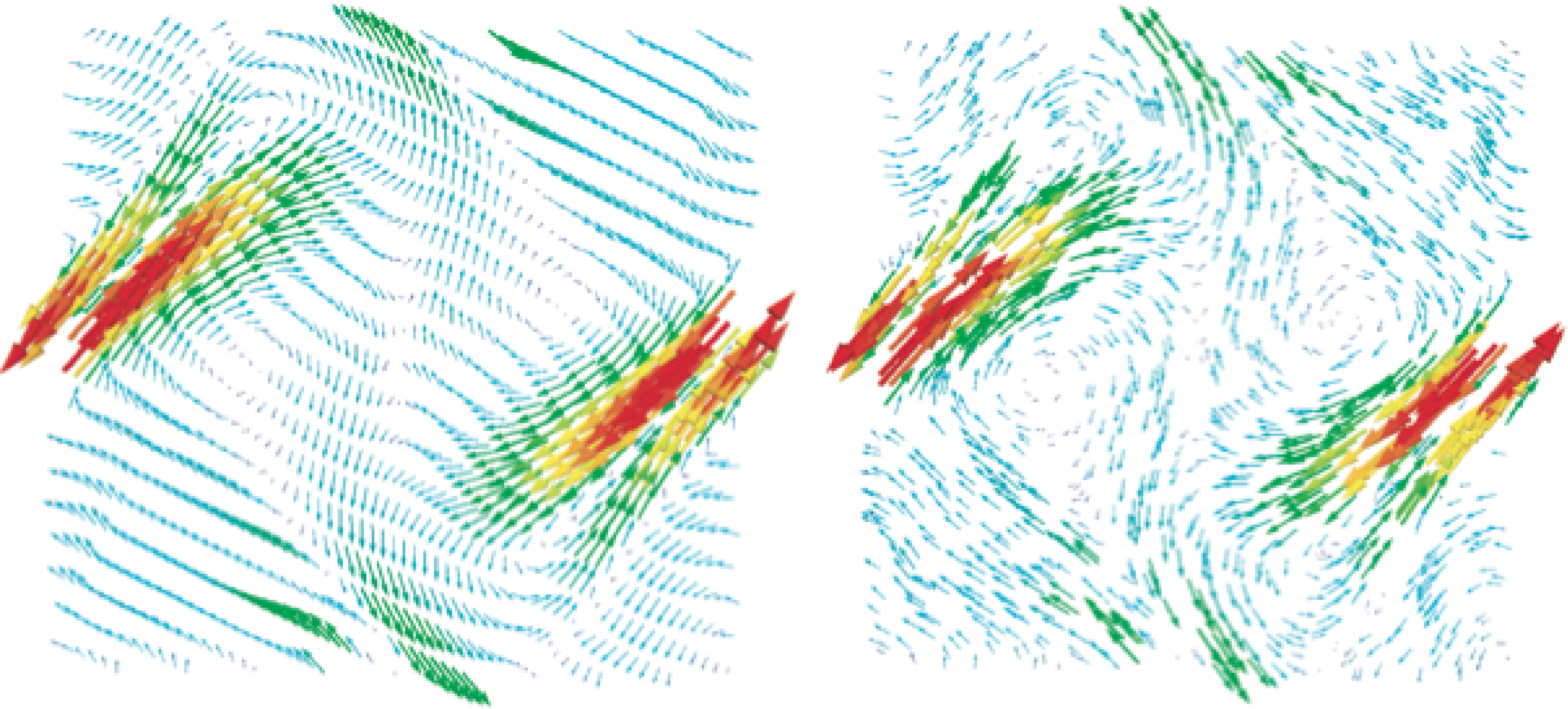
3D arrow glyphs

MHD simulation
256^2 grid

Variants

- cones, arrows, …
  - show *orientation* better than lines
  - but take more space to render
  - shading: good visual cue to separate (overlapping) glyphs

Vector glyphs



samples on a rotated grid

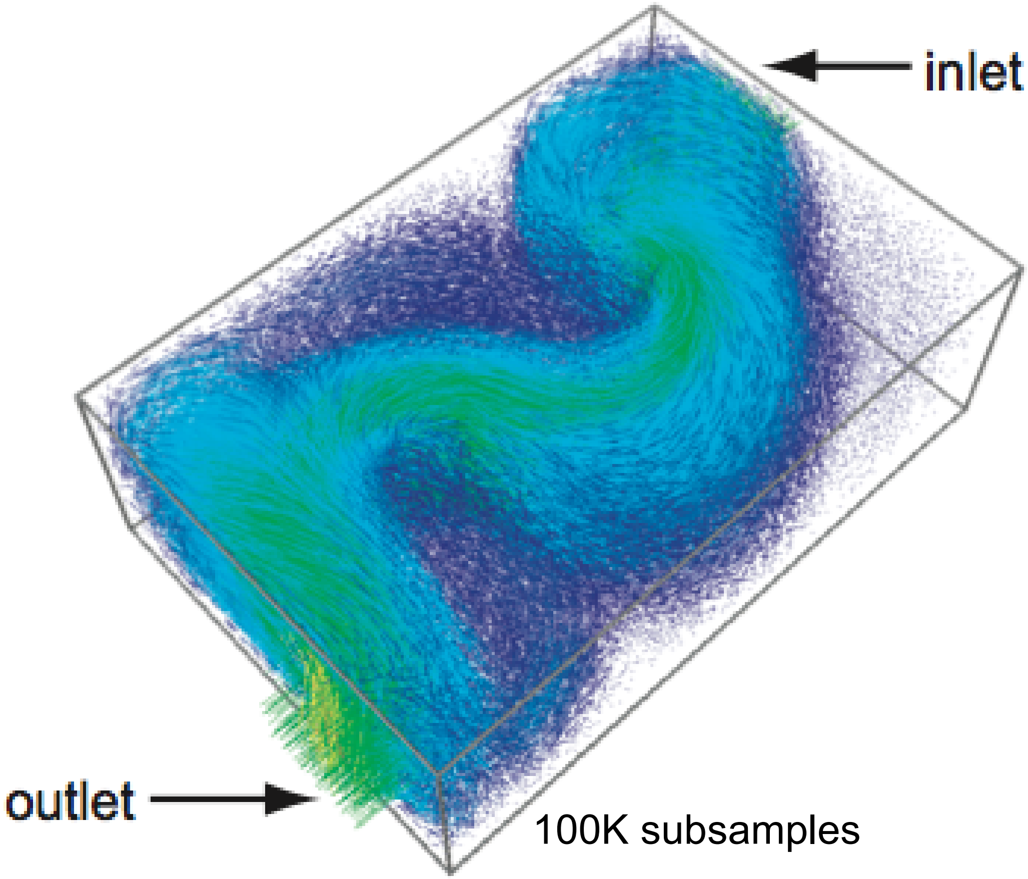
random samples, quasi-uniform density

How to choose sample points

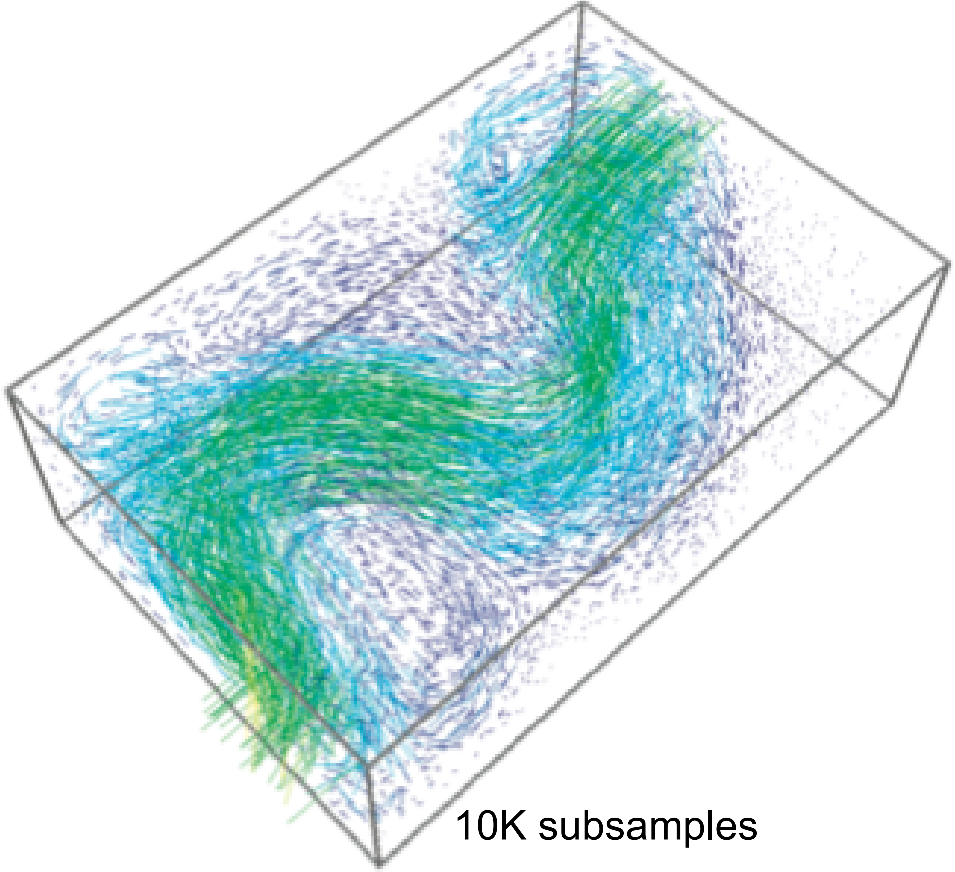
- avoid uniform grids! (why? See sampling theory, ‘beating artifacts’)
- random sampling: generally OK

3D vector glyphs

128x85x42 volume field
456960 data points



100K subsamples

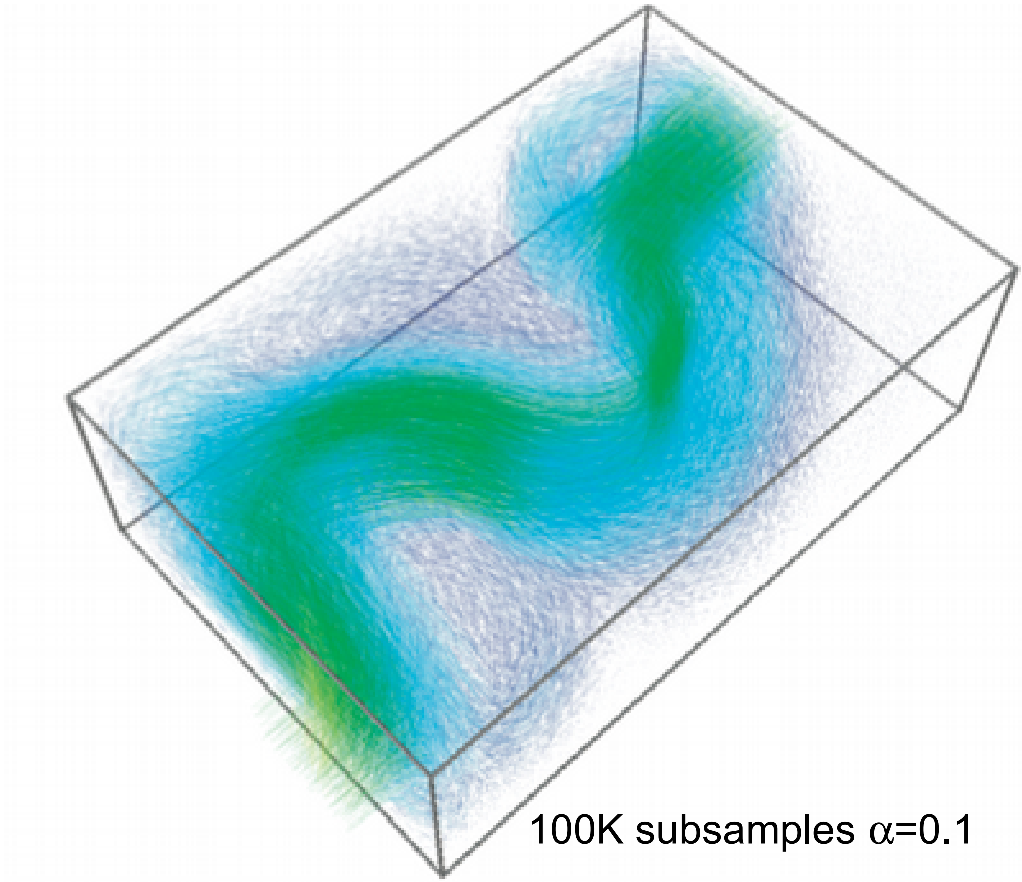


10K subsamples

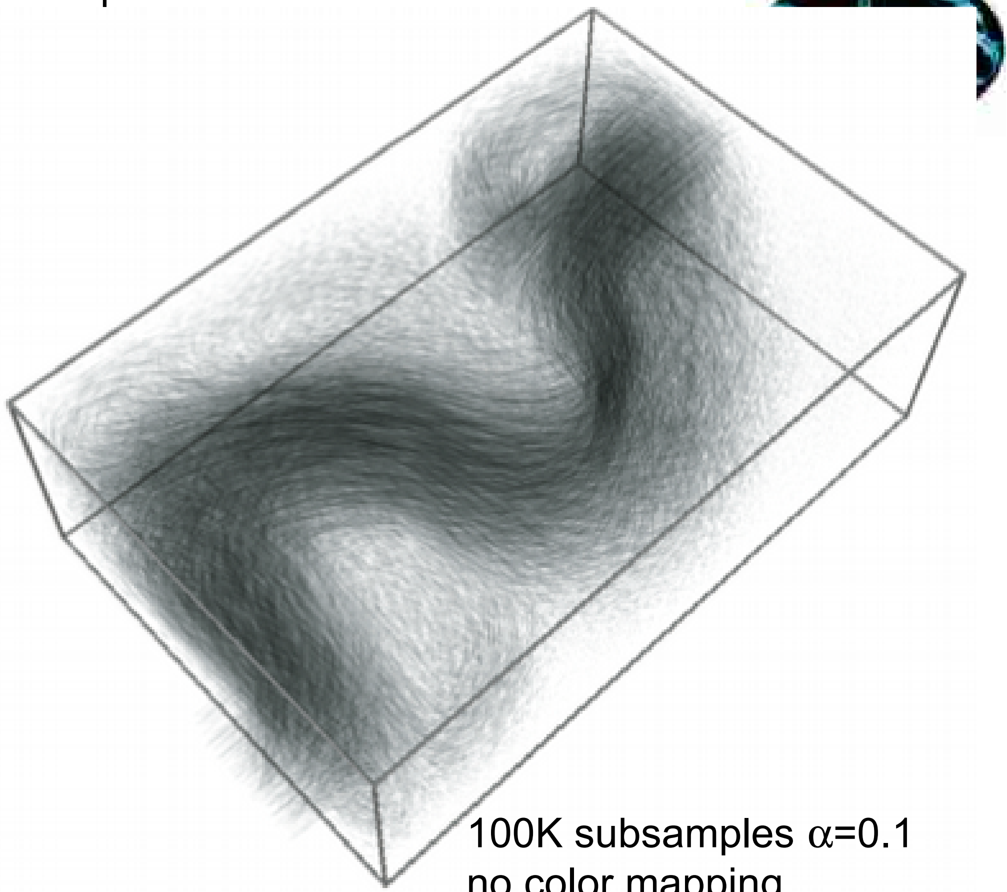
- same idea/technique as 2D vector glyphs
- 3D additional problems
  - more data, same screen space
  - occlusion
  - perspective foreshortening
  - viewpoint selection

3D vector glyphs

128x85x42 volume field
456960 data points



100K subsamples α=0.1

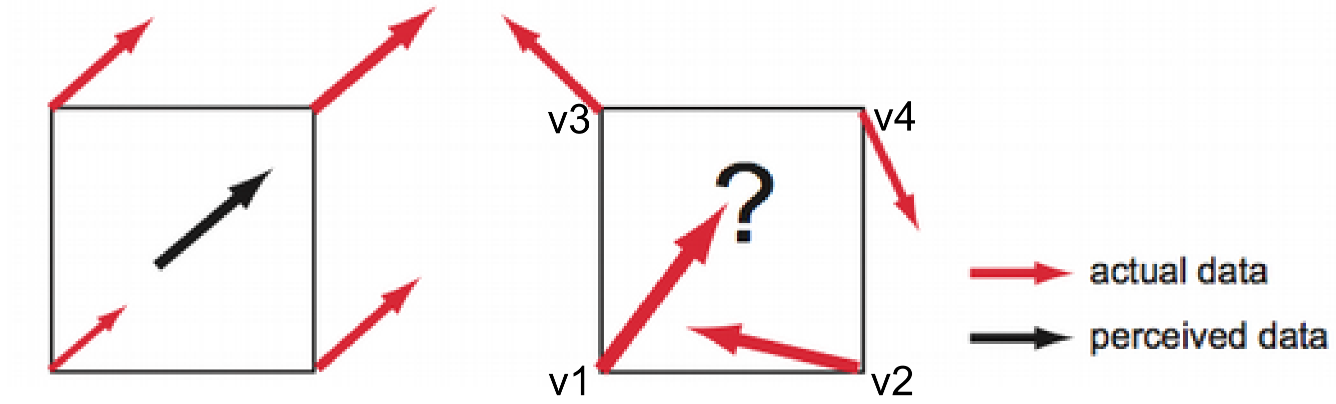


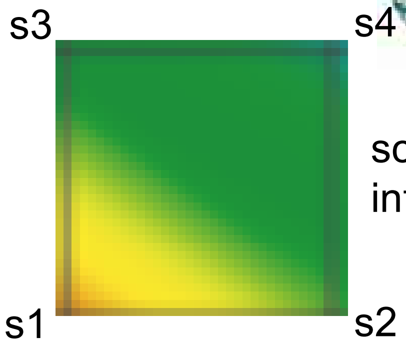
100K subsamples α=0.1
no color mapping

Alpha blending

- extremely simple and powerful tool
- reduce *perceived* occlusion
  - low-speed zones: highly transparent
  - high-speed zones: opaque and highly coherent (why?)

Glyph problem revisited





scalar plot interpolation

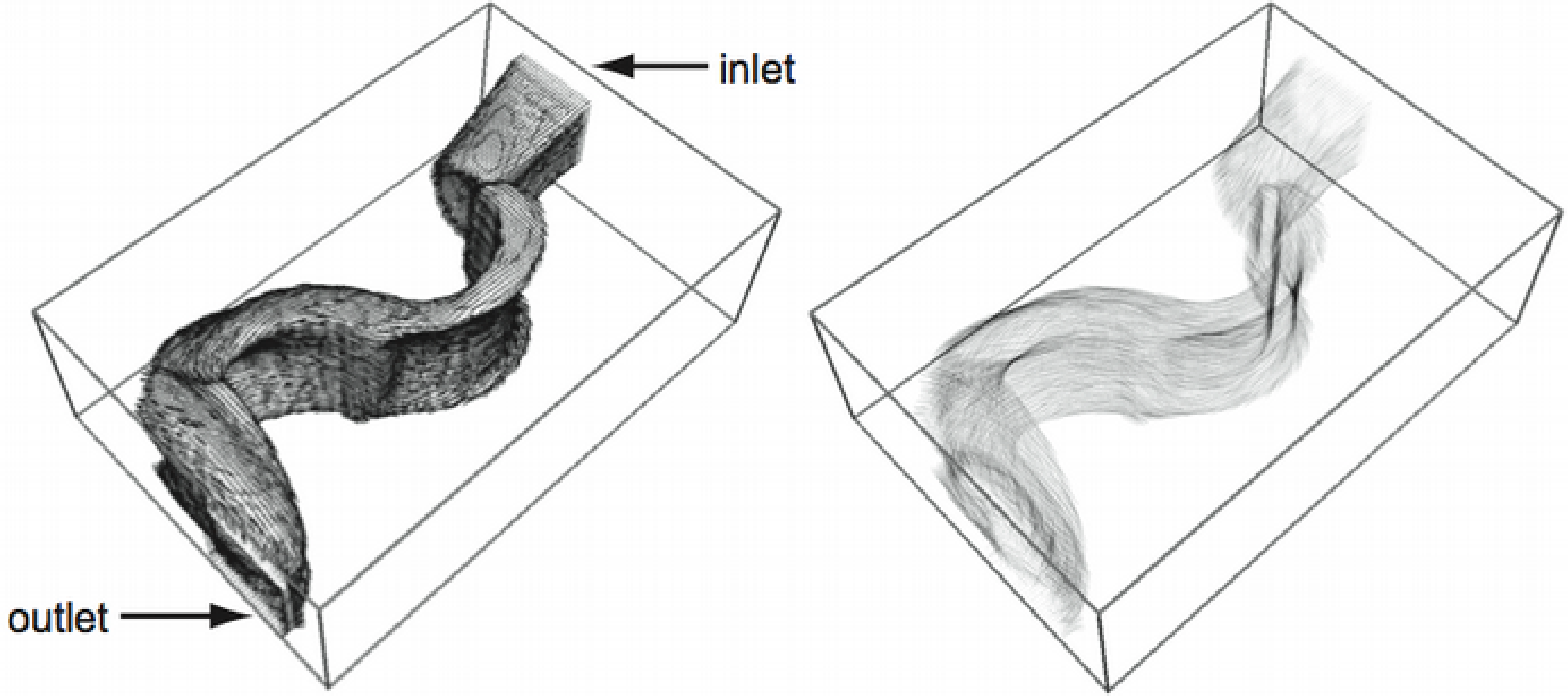
Recall the 'inverse mapping' proposal

- we render something…
- …so we can visually map it to some data/phenomenon

Glyph problems

- **no interpolation** in glyph space (unlike for scalar plots with color mapping!)
- a glyph takes more space than a pixel
- we (humans) aren't good at visually interpolating arrows…
- scalar plots are **dense**; glyph plots are **sparse**
  - this is why glyph positioning (sampling) is extra important

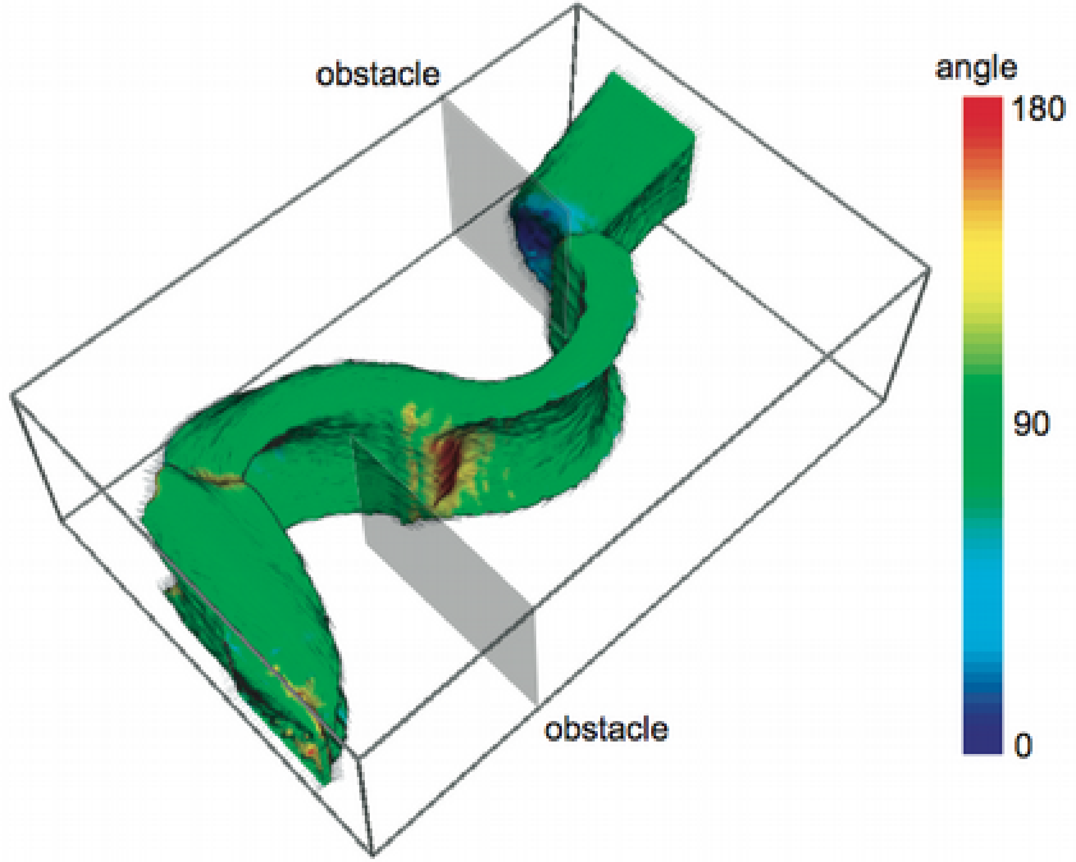
Vector glyphs on 3D surfaces



Trade-off between vector glyphs in 2D planes and in full 3D

- find interesting surface
  - e.g. **isosurface** of flow velocity
- plot 3D vector glyphs on it
- in our example, we don't use color-mapping of velocity

Vector color coding



color = angle between vector field and normal of some given surface

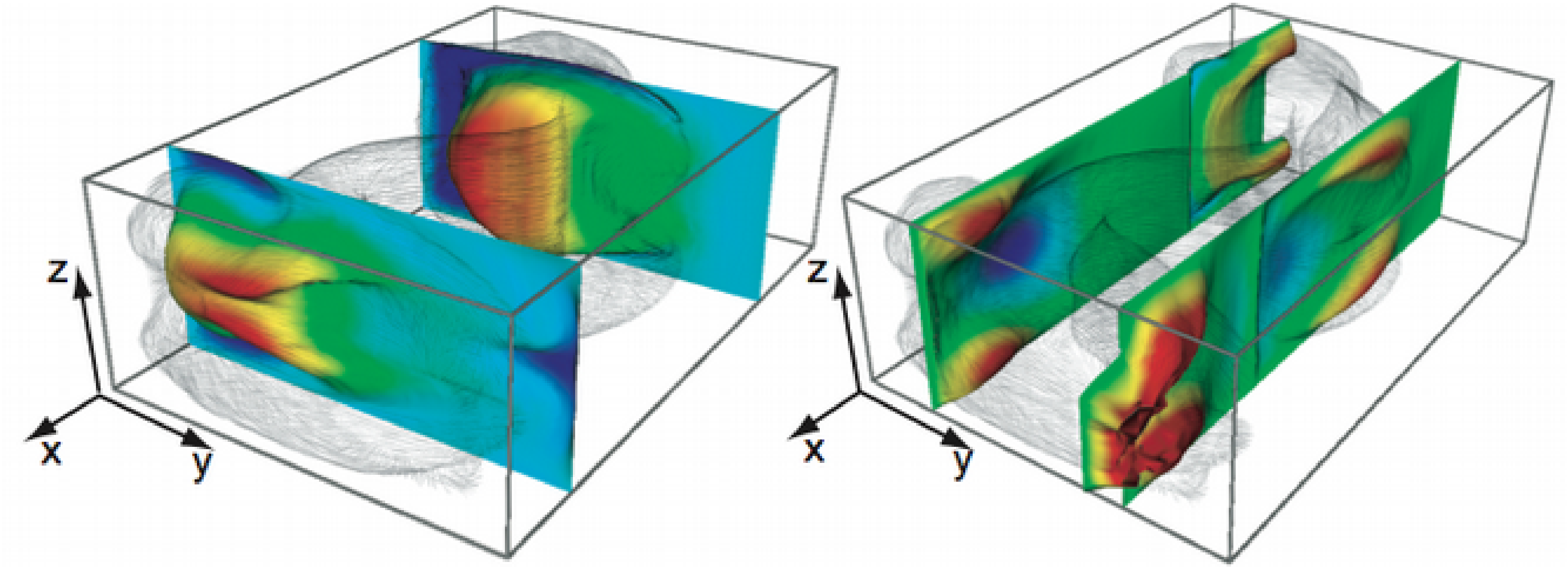
See if vectors are tangent to some given surface

- color-code angle between vector and surface normal
- easily spot
    - tangent regions (flow stays on surface, green)
    - inflow regions (flow enters surface, red)
    - outflow regions (flow exits surface, blue)

Displacement plots (also called warp plots)

Show motion of a 'probe' surface in the field

- define probe surface $S \subseteq D$
- create displaced surface $S_{displ} = \{ x + \mathbf{v}(x)\Delta t,\ \forall x \in S \}$

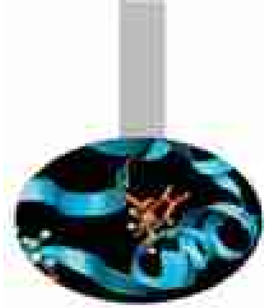


two displacement surfaces orthogonal to x axis

two displacement surfaces orthogonal to y axis

- analogy: think of a flexible sheet bent into the wind
- color can map additional scalar
- robust extension: $S_{displ} = \{ x + (\mathbf{v}(x)\mathbf{n}(x))\,\mathbf{n}(x)\Delta t,\ \forall x \in S \}$
  - removes tangential displacements

Stream objects



Main idea

- think of the vector field $\mathbf{v} : D$ as a flow field
- choose some ‘seed’ points $s \in D$
- move the seed points s in $\mathbf{v}$
- show the trajectories

Stream lines

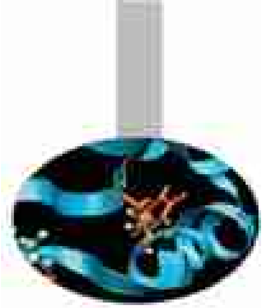
- assume that $\mathbf{v}$ is not changing in time (stationary field)
- for each seed $p_o \in D$
  - the streamline S seeded at p_o is given by

$$S = \{p(\tau), \tau \in [0, T]\}, p(\tau) = \int_{t=0}^{T} \mathbf{v}(p)dt, \quad \text{where } p(0) = p_0$$

integrate p_o in vector field $\mathbf{v}$ for time T

- if v is time dependent $\mathbf{v}=\mathbf{v}(t)$, streamlines are called particle traces

Stream objects



Practical construction

- numerically integrate

$$S = \{p(\tau), \tau \in [0, T]\}, p(\tau) = \int_{t=0}^{\tau} \mathbf{v}(p)dt, \quad \text{where } p(0) = p_0$$

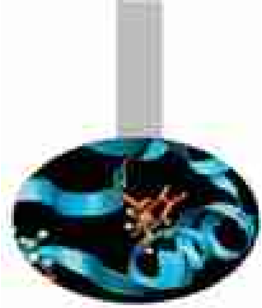
- discretizing time yields

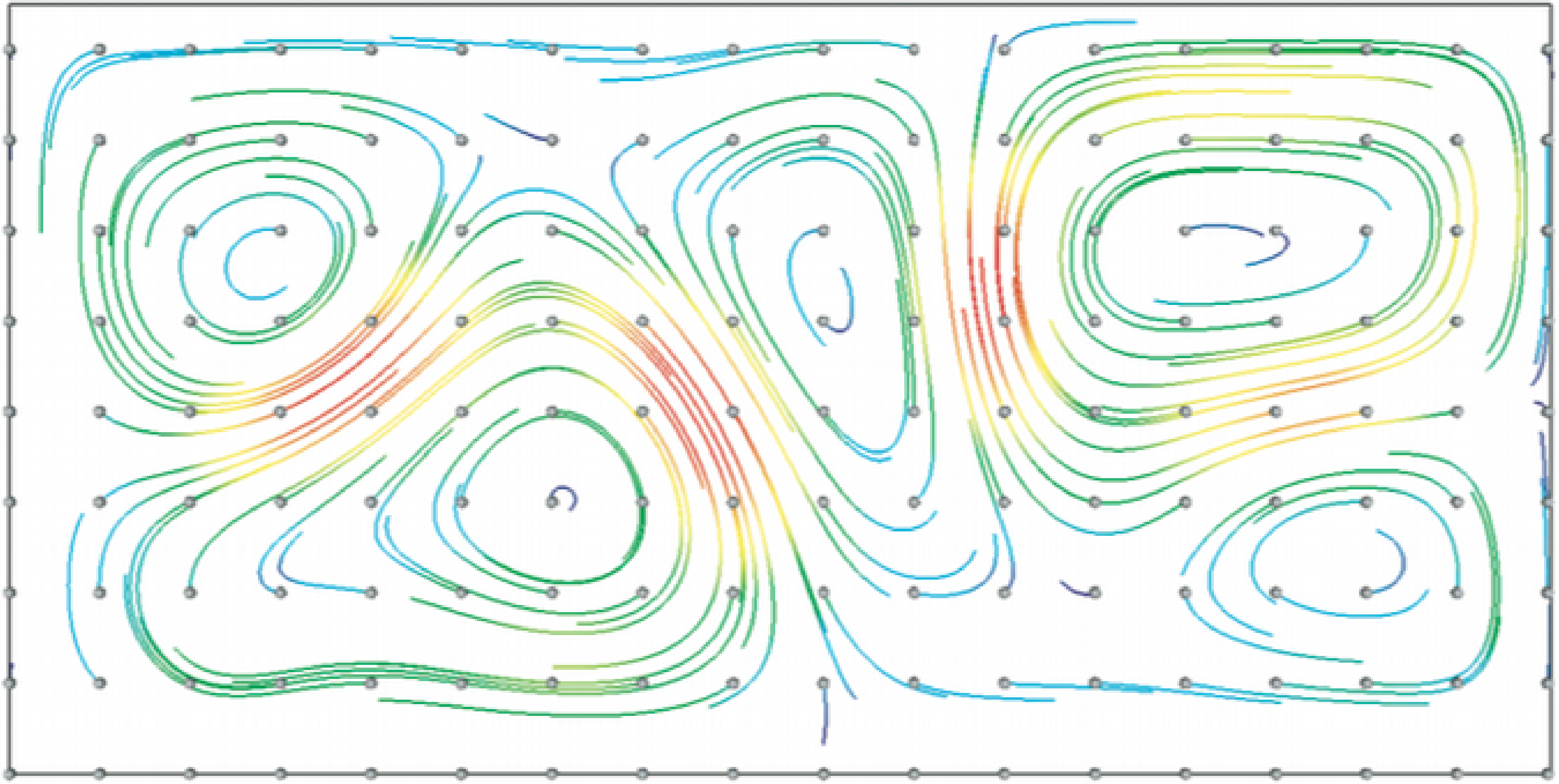
$$\int_{t=0}^{\tau} \mathbf{v}(p)dt = \sum_{i=0}^{\tau/\Delta t} \mathbf{v}(p_i)\Delta t \quad \text{where } p_i = p_{i-1} + \mathbf{v}_{i-1}\Delta t$$

(simple Euler integration)

- recall our discussion on interpolation and basis functions
- Euler integration explained
  - we consider $\mathbf{v}$ constant between two sample points p_i and p_{i+1}
  - we compute $\mathbf{v}(p)$ by linear interpolation within the cell containing p
  - variant: use $\mathbf{v}(p)/\|\mathbf{v}(p)\|$ instead of $\mathbf{v}(p)$ in integral (why better?)
  - S will be a polyline, $S = \{p_i\}$
- stop when $\tau=T$ or $\mathbf{v}(p)=0$ or $p \notin D$
  - what does $\tau=T$ mean when we use $\mathbf{v}(p)/\|\mathbf{v}(p)\|$?

Stream objects



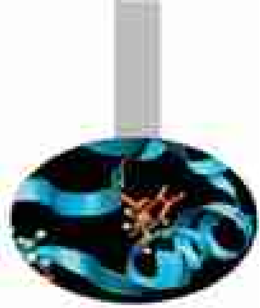


streamlines: seeds from regular grid; use un-normalized $\mathbf{v}$ for integration; color by $\|\mathbf{v}\|$

better than vector glyphs

- less intersections than for hedgehog plots as streamlines do not intersect
- the image more continuous: pixel continuity along lines

Good stream objects design



Coverage

- each dataset point should be close to a stream object
- why?
  - because we need to easily do the inverse mapping at any dataset point

Uniformity

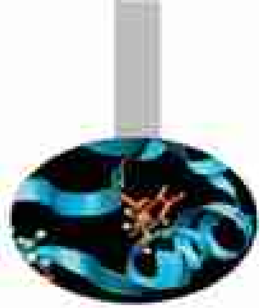
- stream object density should be quasi-uniform
- why?
  - because we want to avoid high-clutter areas *and* no-information areas

Continuity

- long stream objects preferable to short ones
- why?
  - because we can easier follow few, long, objects than many short ones

Note:

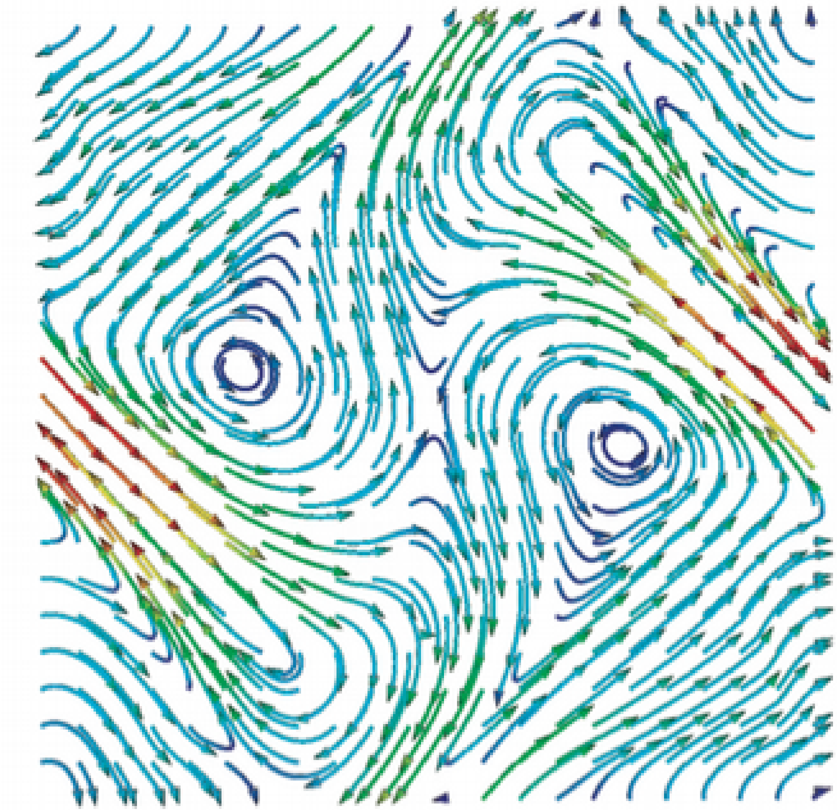
- all above can be seen as an *optimization process* on the seeds and integration time
- however, efficient and robust solutions of this optimizations are generally hard



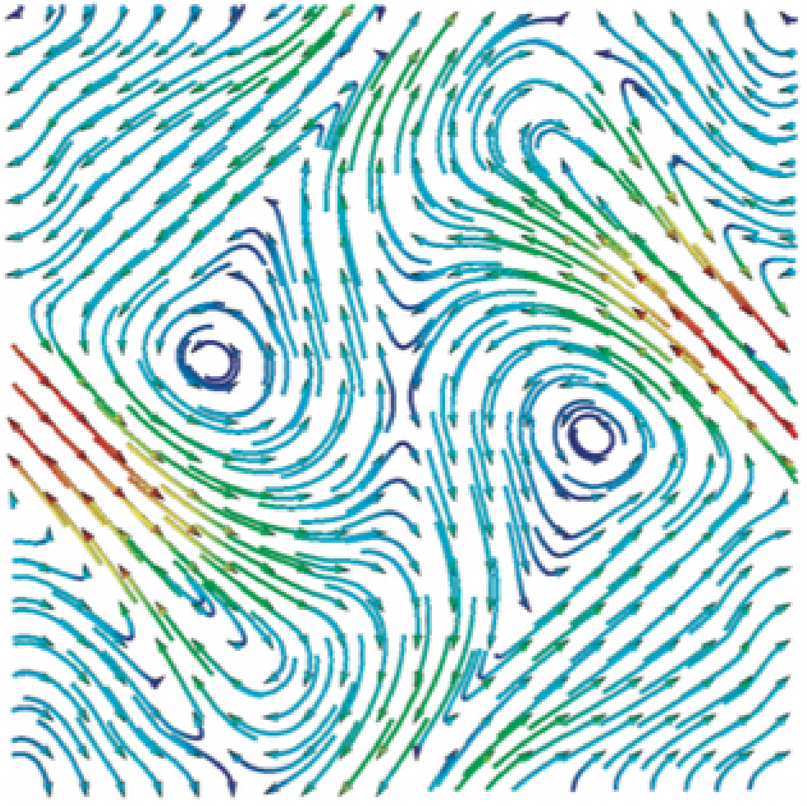
Stream tubes

Like stream objects, but 3D

- compute 1D stream objects (e.g. streamlines)
- sweep (circular) cross-section along these
- visualize result with shading



stream tubes, forward integration



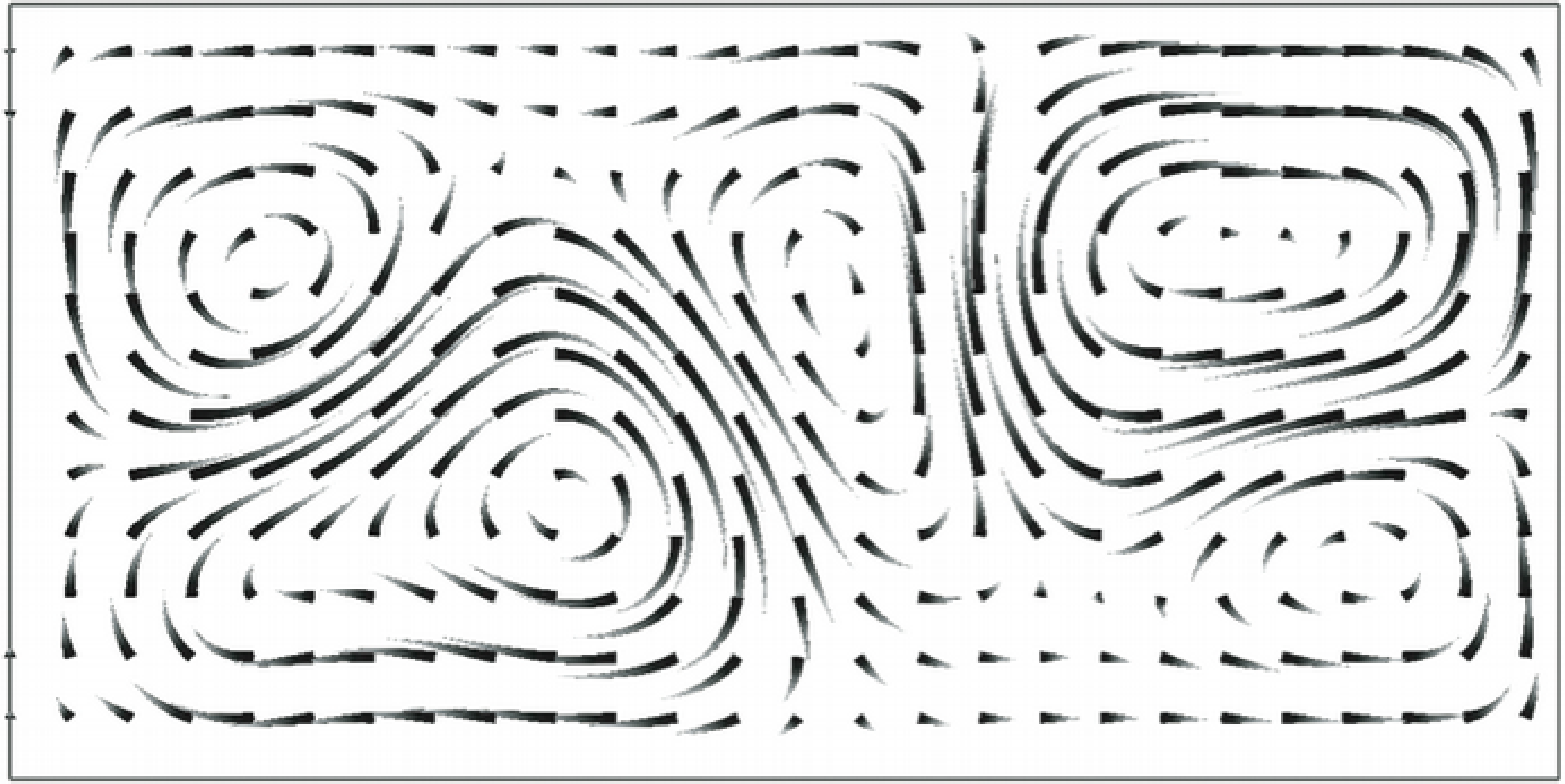
stream tubes, backward integration

- in 2D they are a nicer option than hedgehog/glyph plots

Stream tubes

Variations

- modulate tube thickness by
  - data (we’ll see this later in Module 5 – hyperstreamlines)
  - integration time – we obtain nice tapered arrows

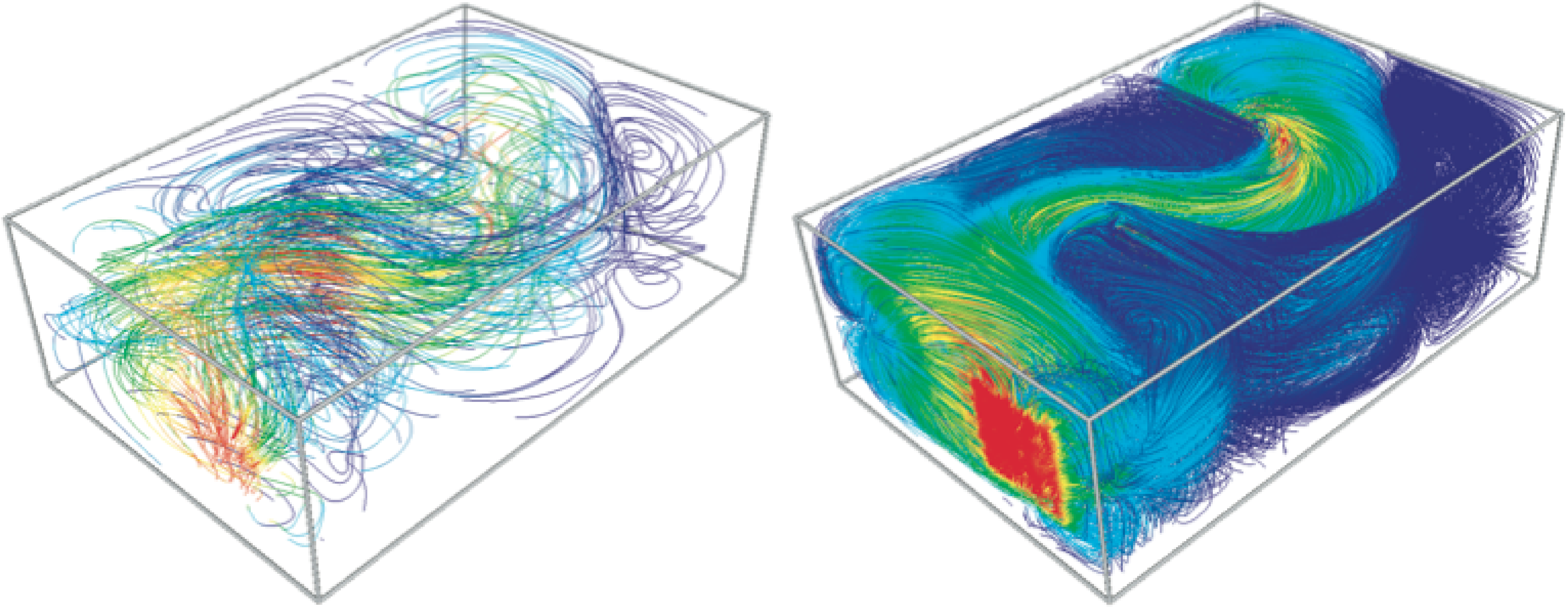


stream tubes – radius *and* opacity decrease with integration time

Stream lines in 3D

Tough problem

- more lines, so increased occlusion/clutter



undersampling 10x10x10, opacity=1

- not too much occlusion
- but little insight in the flow field

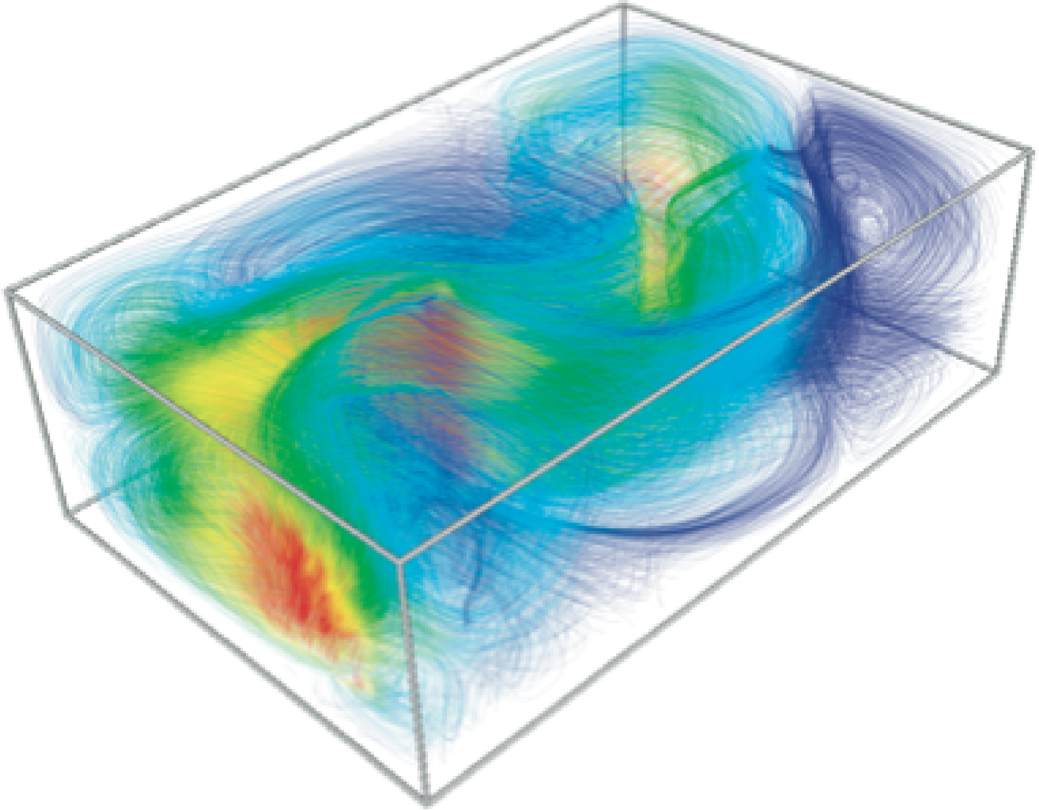
undersampling 3x3x3, opacity=1

- more local insight (better coverage)
- but too much occlusion

Stream lines in 3D

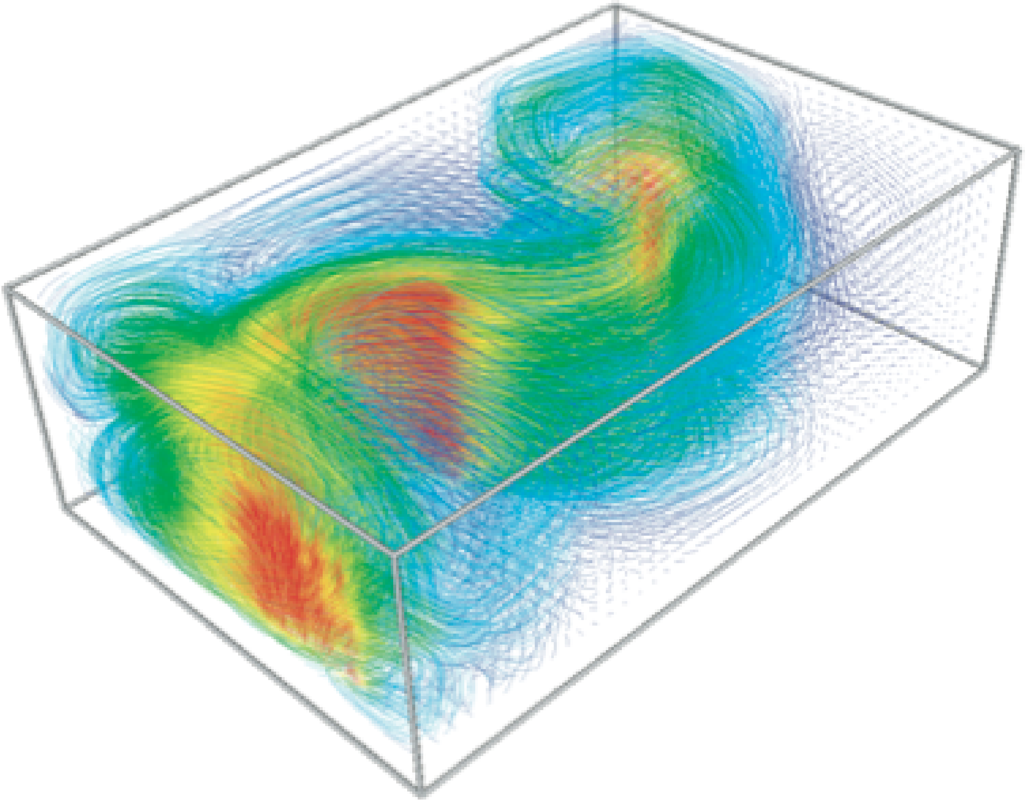
Variations

- play with opacity, seeding density, integration time



undersampling 3x3x3, opacity=0.1

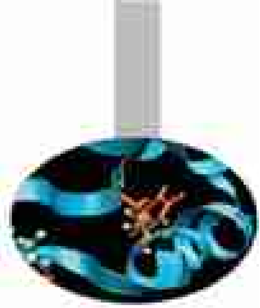
- less occlusion (see through)
- good coverage



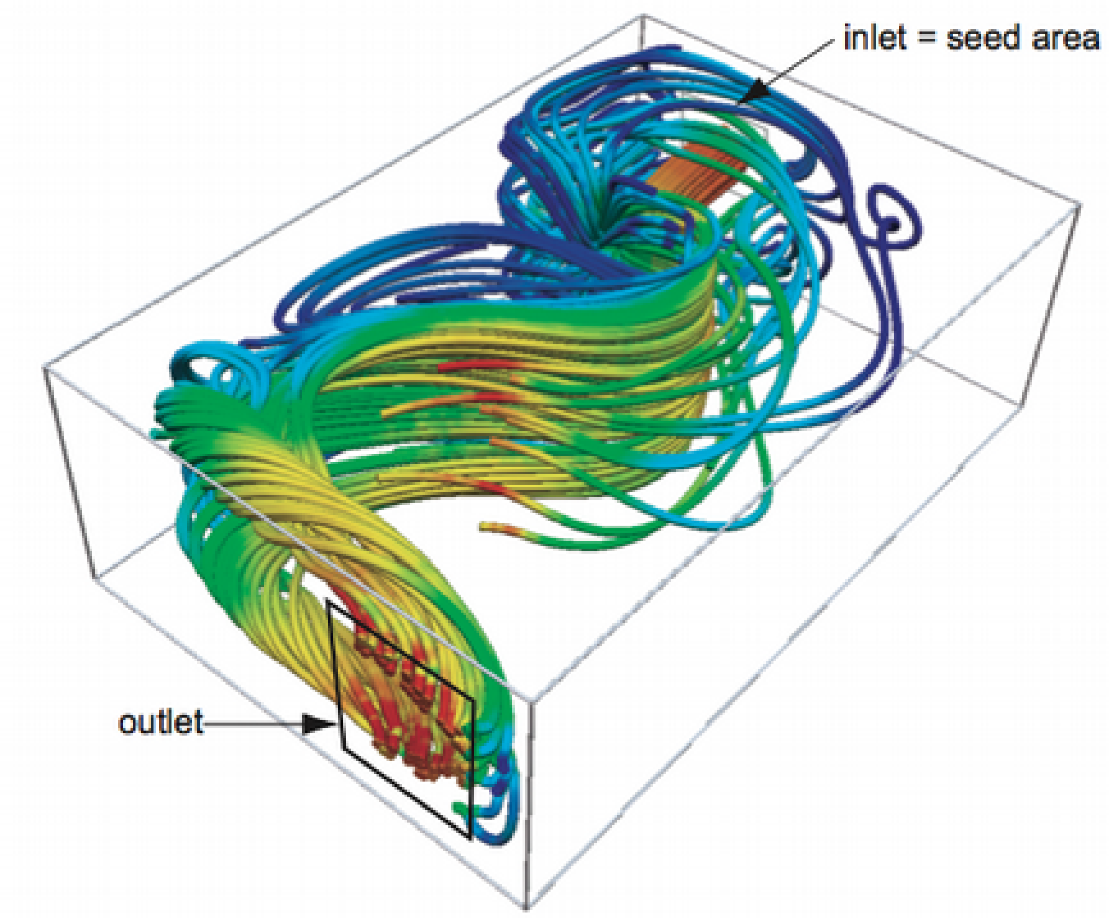
undersampling 3x3x3, shorter time

- more local insight (better coverage)
- even less occlusion
- but less continuity

Stream tubes in 3D



- even higher occlusion problem than for 3D streamlines
- must reduce number of seeds



stream tubes traced from inlet to outlet

- show where incoming flow arrives at
- color by flow velocity
- shade for extra occlusion cues

Image-based vector field visualization

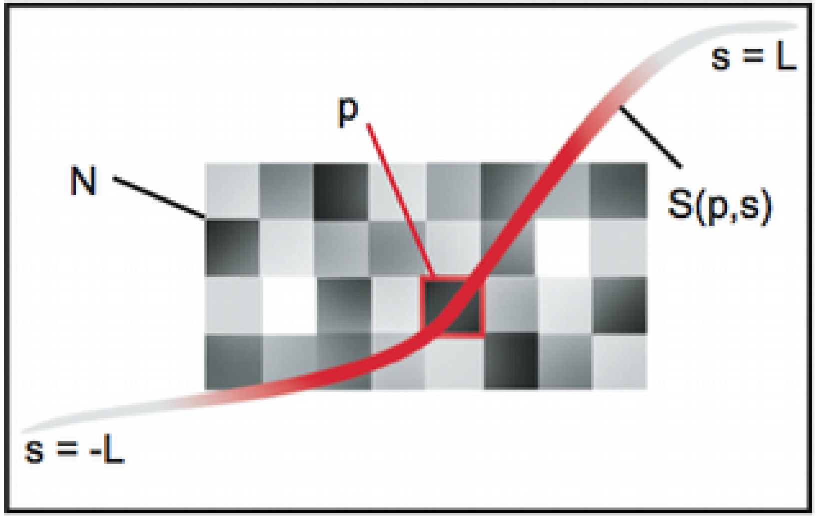
So far

- we had discrete visualizations (glyphs, streamlines, stream ribbons, warp plots)

Now

- we want a dense, pixel-filling, continuous, vector field visualization

Principle

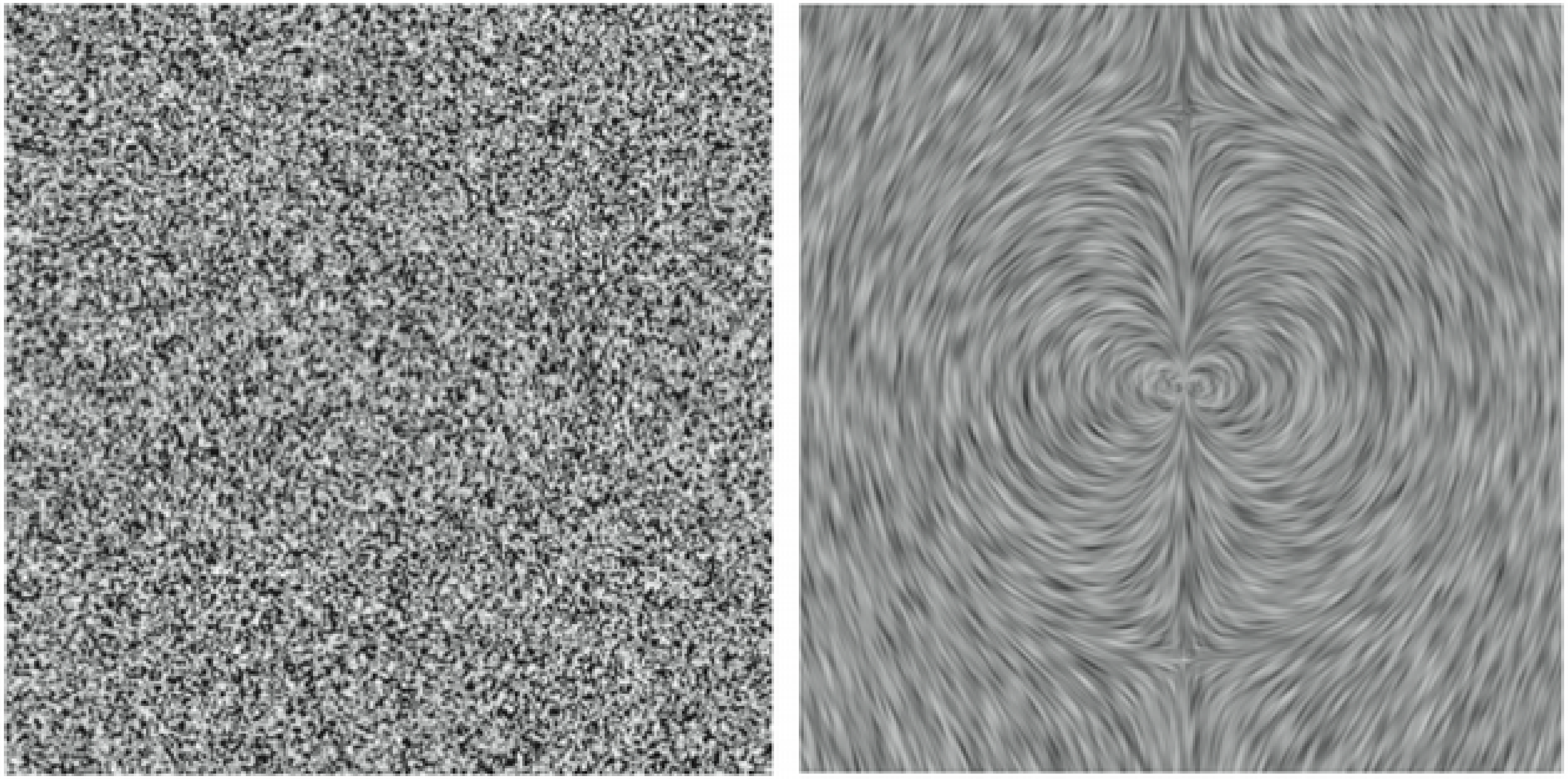


$$T(p) = \frac{\int_{-L}^{L} N(S(p,s))k(s)ds}{\int_{-L}^{L} k(s)ds}$$

gray value at pixel p
N = noise texture

- take each pixel
- trace a streamli al)
- blend all stream
  - multiplied by a random grayscale value at p
  - with opacity decreasing (exponentially) on distance-along-streamline from p
- identical to *blurring* (convolving) noise along the streamlines of $\mathbf{v}$

Image-based vector field visualization



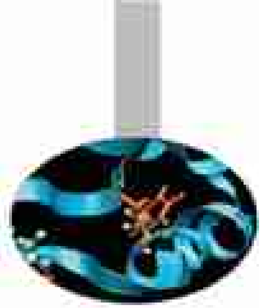
noise texture

line integral convolution (LIC)

Line integral convolution

- highly coherent images along streamlines (why? because of **v**-oriented blurring)
- highly contrasting images across streamlines (why? because of random noise)
- easy to interpret images

Image-based animated flow visualization



Main idea

- extend LIC with animation
- dynamics help seeing *orientation* and *speed* (not shown by LIC)

Algorithm

- consider a time-and-space dependent property $I : D \times \mathbf{R}_+ \rightarrow \mathbf{R}_+$ (e.g. gray value)
- advect I in time over D

$$I(x + \mathbf{v}(x,t)\Delta t, t + \Delta t) = I(x,t)$$

- …and also inject some noise at each point of D

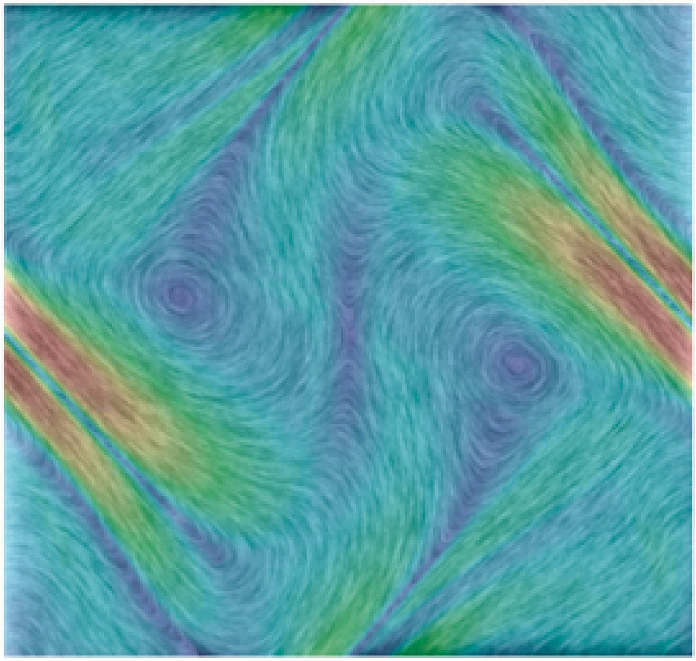
$$I(x+\mathbf{v}(x,t)\Delta t, t+\Delta t) = (1-\alpha)I(x,t) + \alpha N(x+\mathbf{v}(x,t)\Delta t, t+\Delta t)$$

advected term

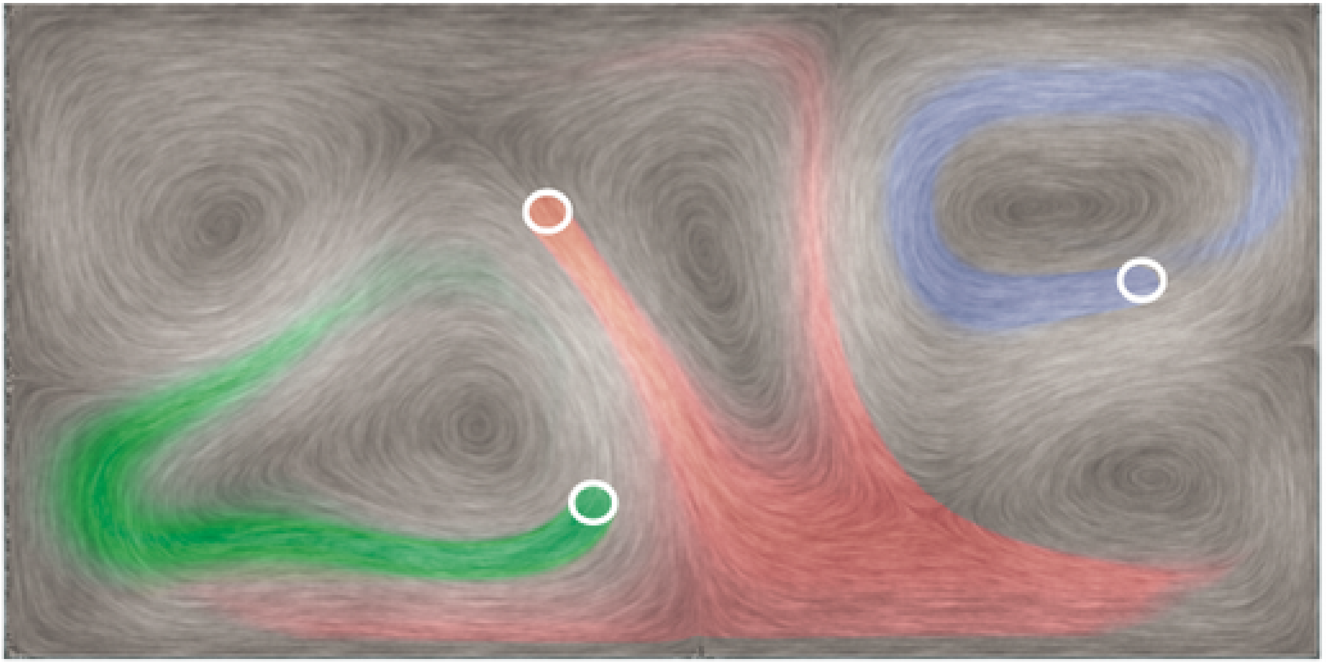
injected noise term

balance between advection
and noise injection

Image-based flow visualization (IBFV)



IBFV, velocity color-coded



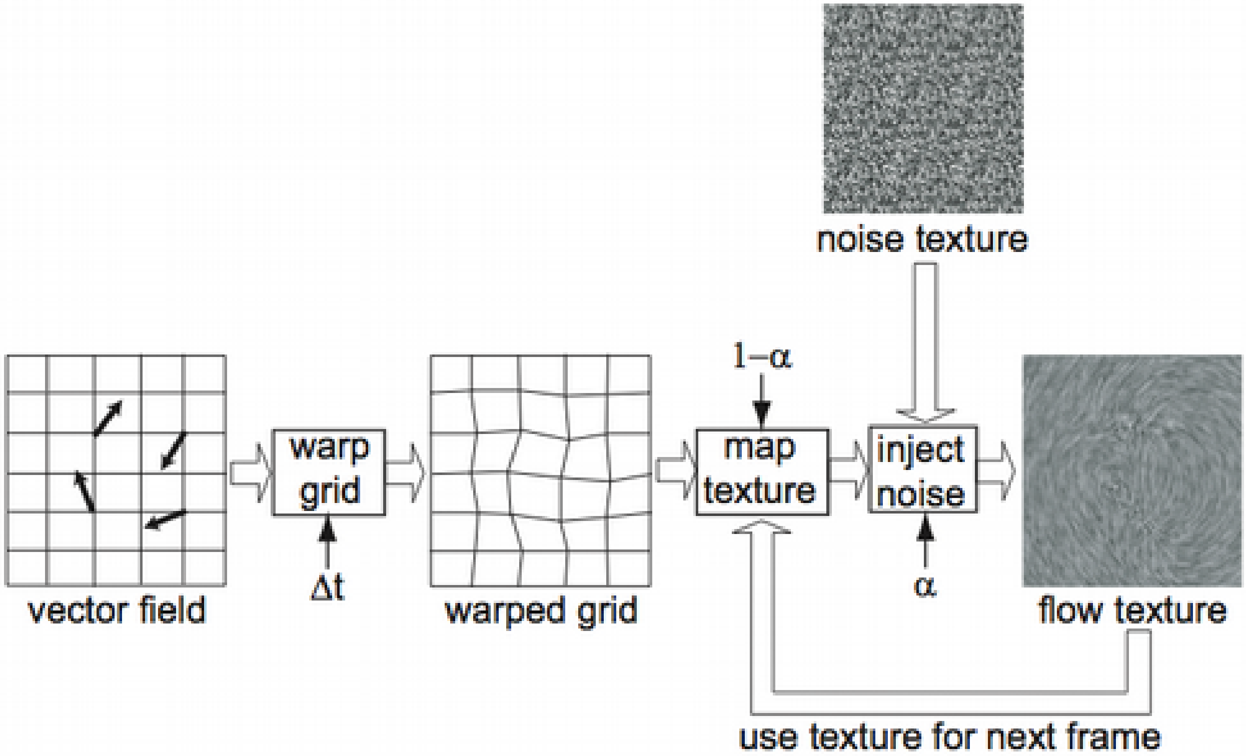
IBFV, with user-placed colored ink seeds
and luminance-coded velocity magnitude

Implementation

- sounds complex, but it's really easy^^ (200 LOC C with OpenGL)
  - see next slide for details
- real-time (hundreds of frames per second) even for modest graphics cards
- naturally handles time-dependent vector fields

Image-based flow visualization (IBFV)

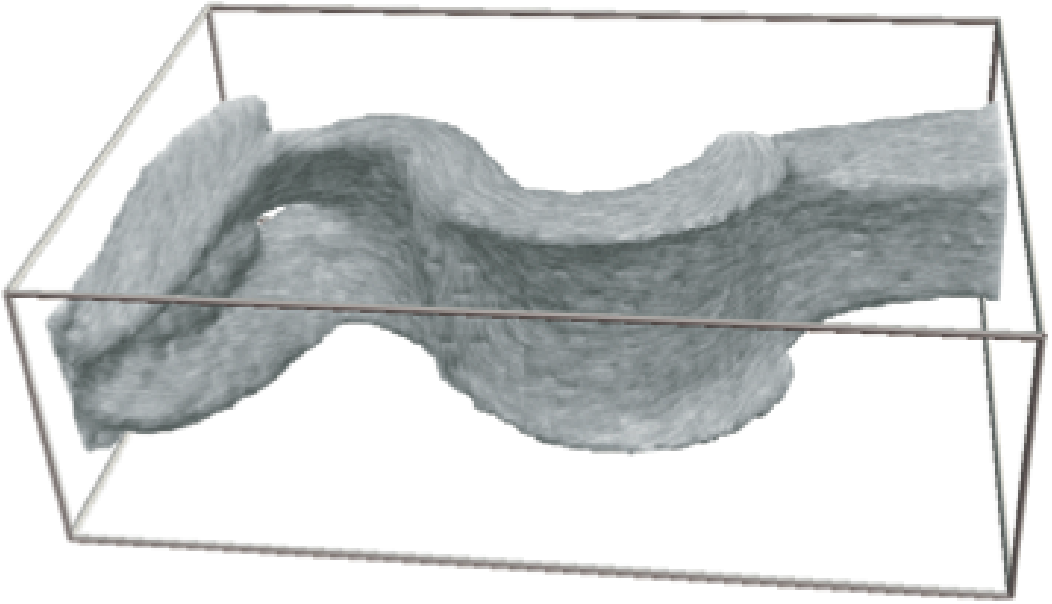
Implementation



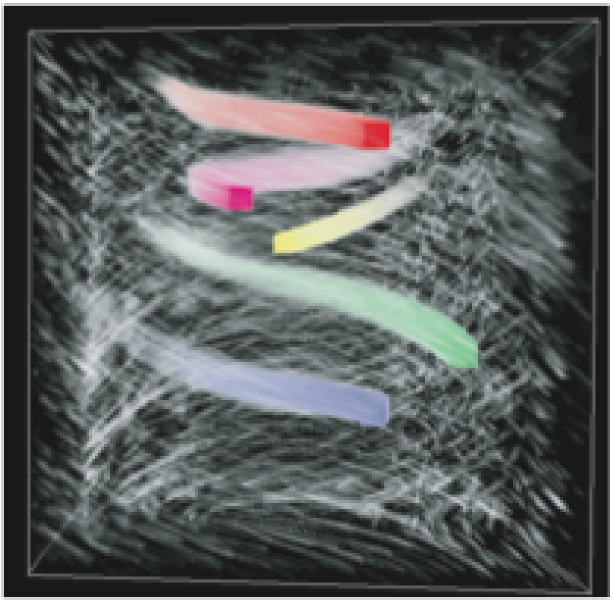
- define grid on 2D flow domain D
- warp grid D along $\mathbf{v}$ into D_{warp}
- forever
    - read current frame buffer into I
    - draw D_{warp} textured with I (advection) with opacity $1\text{-}\alpha$
    - blend noise texture N' atop of I (injection) with opacity α

Image-based flow visualization (IBFV)

Variants on 3D curved surfaces and 3D volumes



IBFV on curved surfaces



IBFV in 3D volumes

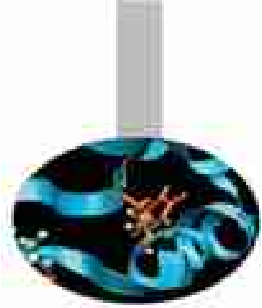
Curved surfaces

- basically same as in planar 2D, just some implementation details different

3D volumes

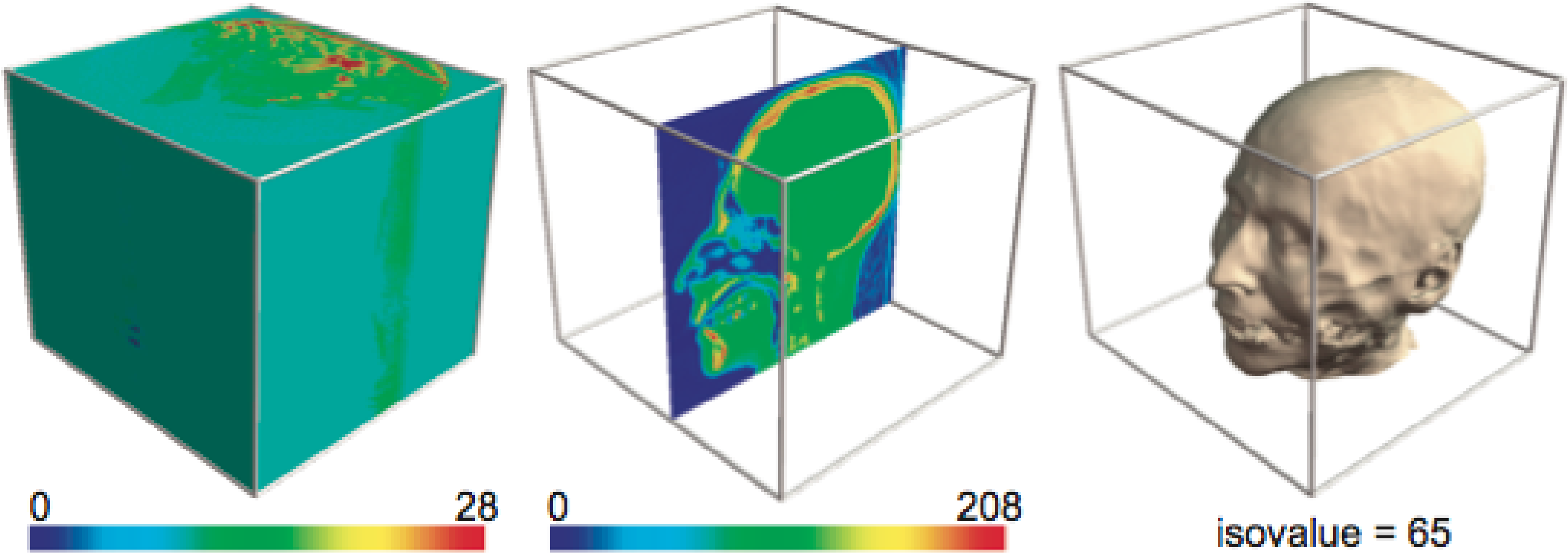
- must do something to ‘see through’ the volume
- use an ‘opacity noise’ (similarly injected as grayvalue noise)
- effect: similar to snowflakes drifting in wind on a black background

Volume visualization: Motivation



Scalar volume $s : \mathbf{R}^3 \to \mathbf{R}$

How to visualize this?



direct color mapping

- see only outer surface

slicing

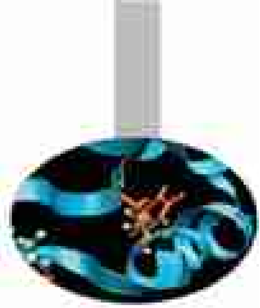
- all details on slice
- no info outside slice

contouring

- all details on contour
- no info outside contour

How to visualize this so we see through the volume

www.cs.rug.nl/svcg



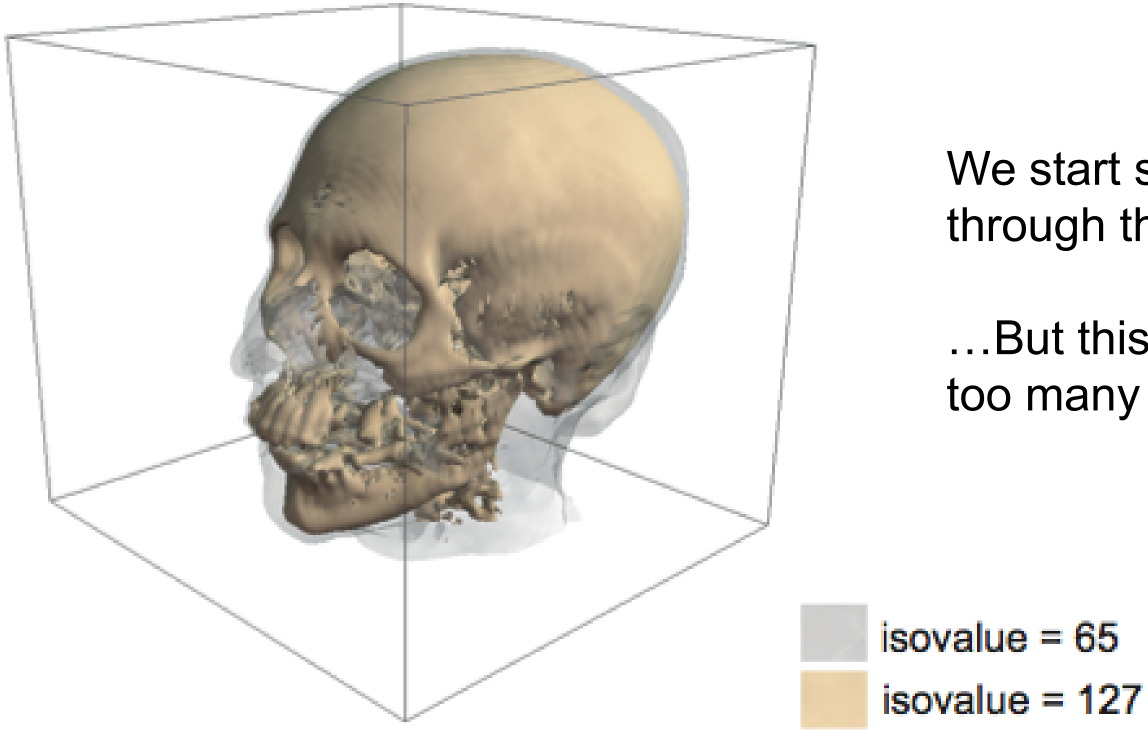
Seeing through a volume

Idea

- use known techniques (slices and contours)
- use transparency

First try

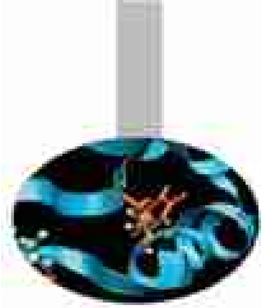
- draw several contours C_i for several values s_i
- transparency α_i proportional to scalar value s



We start seeing a little bit through the volume…

…But this won't work for too many contours

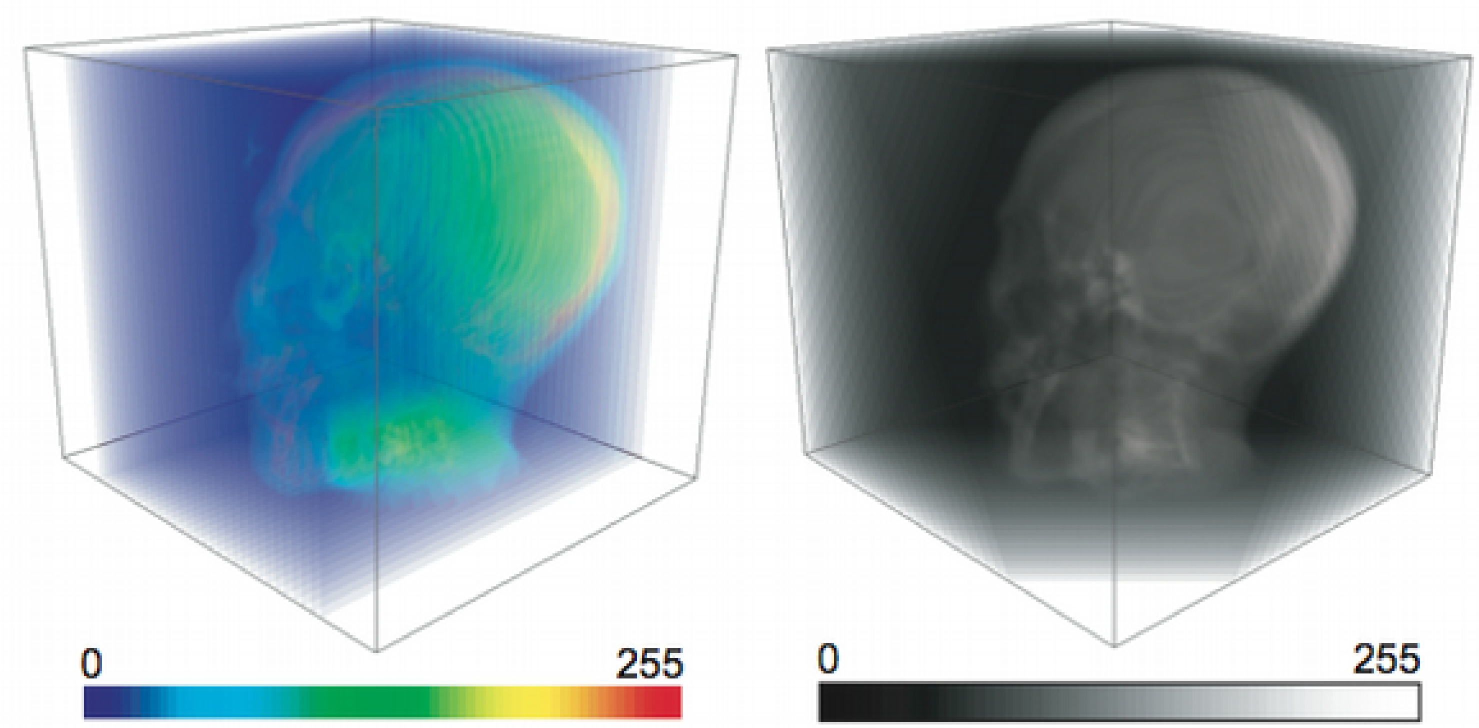
Seeing through a volume



Second try

- draw several parallel slices S_i
- transparency α_i inversely proportional to number of slices

$$\alpha_i = \frac{1}{\| S \|}$$



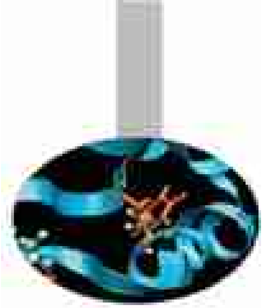
axis-aligned slices

- not OK if we view volume across slicing direction

view direction-aligned slices

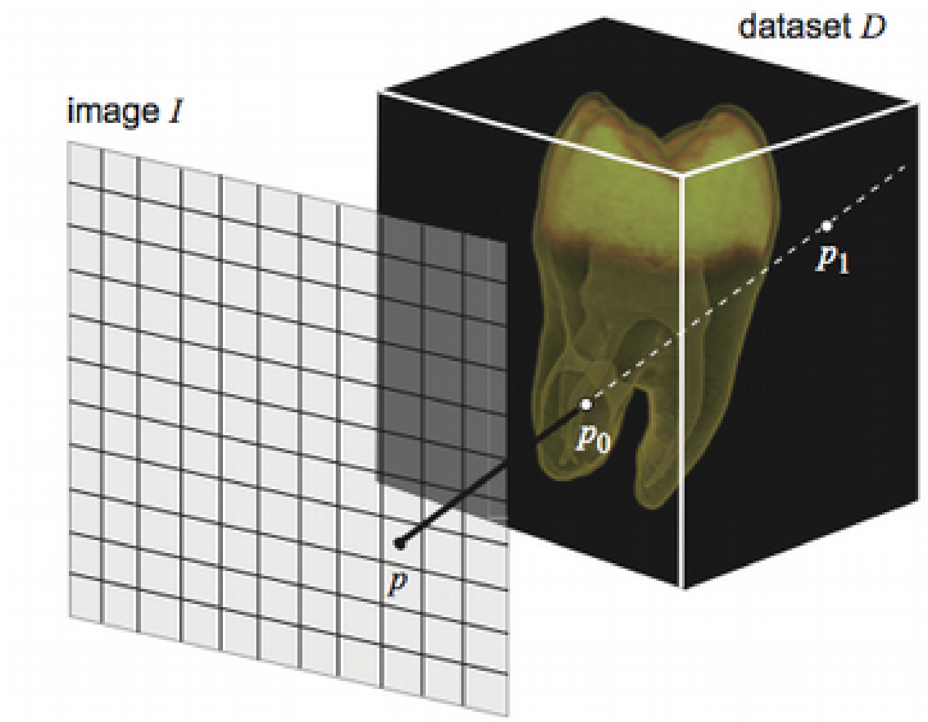
- any viewing direction OK
- must reslice when changing viewpoint

Volume rendering basics



Main idea

- consider a scalar signal $s : D \to \mathbf{R}$ to be drawn on the screen image I
- for each pixel $p \in I$
  - construct a ray $\mathbf{r}$ orthogonal to I passing through p
  - compute intersection points p_0 and p_1 of $\mathbf{r}$ with D
  - express $I(p)$ as function of s along $\mathbf{r}$ between p_0 and p_1



1. Parameterize ray

$$p(t) = (1-t)p_0 + tp_1, \quad t \in [0,1]$$

1. Compute pixel color

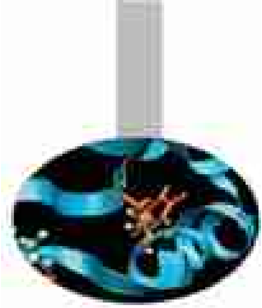
$$I(p) = f(F(s(t)), \quad t \in [0,1]$$

transfer function

ray function

www.cs.rug.nl/svcg

Volume rendering



Define a ray function

$$F : \{s(t) \mid t \in [0,1]\} \rightarrow \mathbf{R}$$

all scalar values along ray

a single resulting scalar value

The ray function ’aggregates’ all scalar values along a ray

Next, define a transfer function

$$f : \mathbf{R} \rightarrow [0,1]^4$$

a single scalar value

an RGBA color

- same concept as color mapping (see Module 2)

Idea

- ray function: says how to combine all scalar values along a ray into a single value
- transfer function: says how to map a single scalar value to a color
- The process of computing all rays for an image I is called ray casting

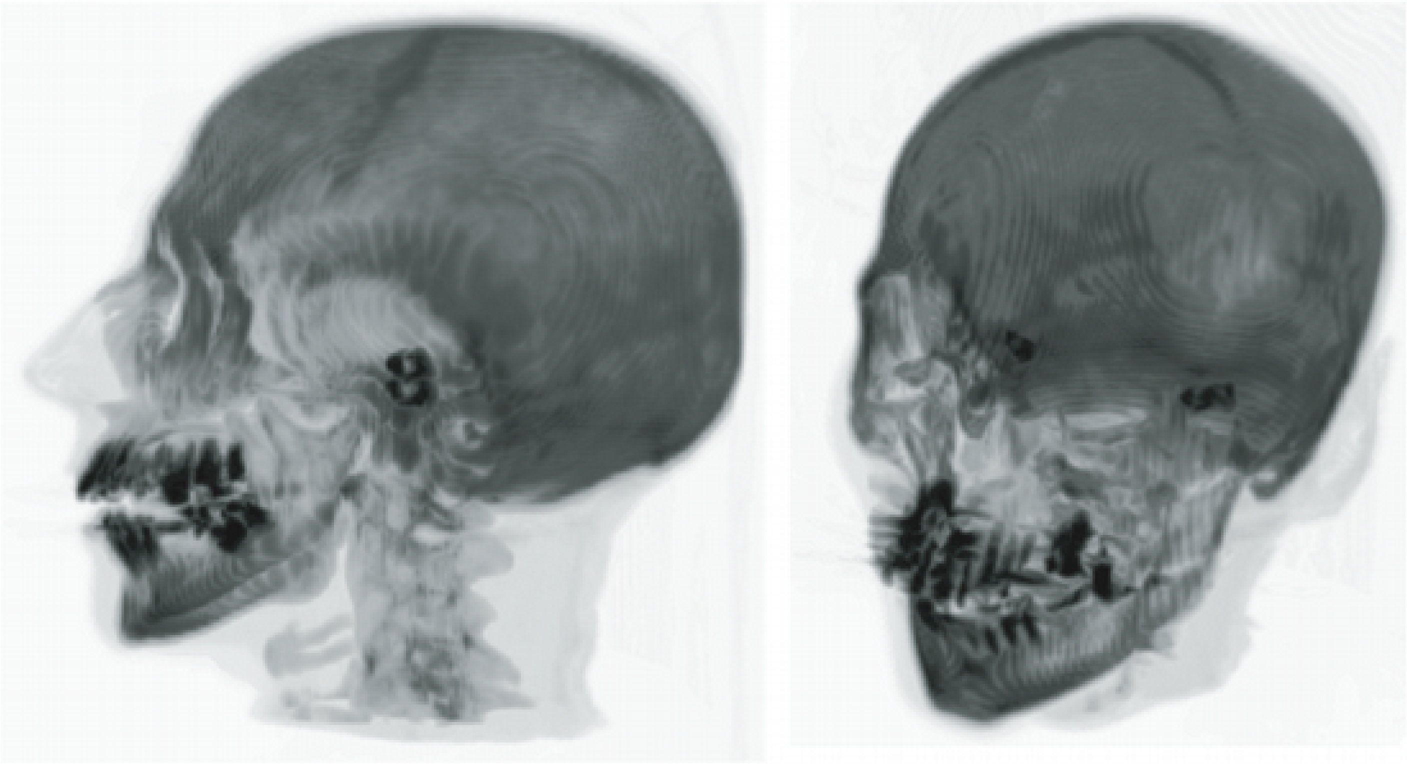
Maximum intensity projection (MIP)

First example of ray function

- find maximum scalar along ray, then apply transfer function to its value

$$I(p) = f\left(\max_{t\in[0,T]} s(t)\right)$$

- useful to emphasize high-value points in the volume



Example
MIP of human head CT

- white = low density (air)
- black = high density (bone)

OK, but gives no depth cues

www.cs.rug.nl/svcg

Average intensity projection

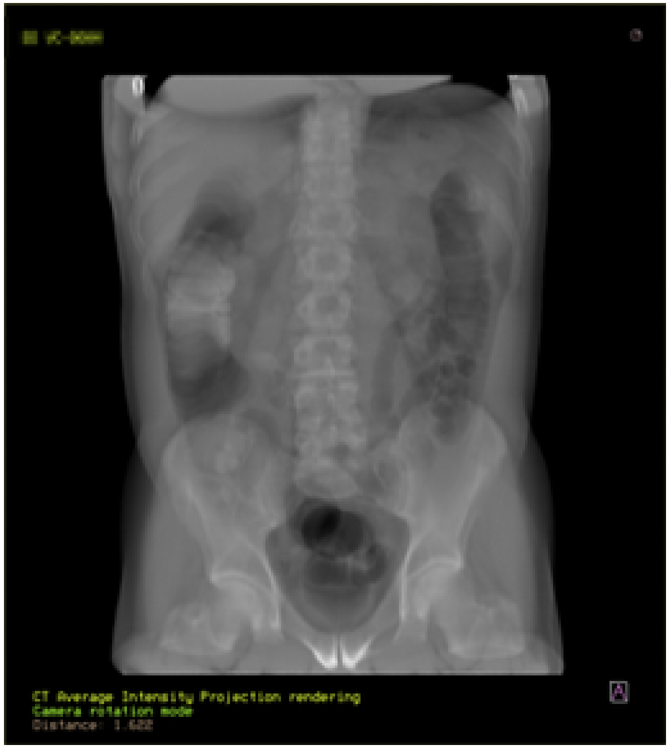
Second example of ray function

- compute average scalar along ray, then map it to color

$$I(p) = f\left(\frac{\int_{t=0}^{T} s(t)dt}{T}\right)$$

- useful to emphasize average tissue type (e.g. density in a CT scan)

maximum intensity projection



average intensity projection

Example
Human torso CT

- black = low density (air)
- white = high density (bone)

Average intensity projection is equivalent to an X-ray

Distance to value function

Third example of ray function

- compute distance along ray until a specific scalar value σ
- $I(p) = f\left(\min_{t\in[0,T],s(t)\geq\sigma} t\right)$ some specific tissue is located

distance to value 20

distance to value 50

Example
Human head CT

- black = low distance
- white = high distance

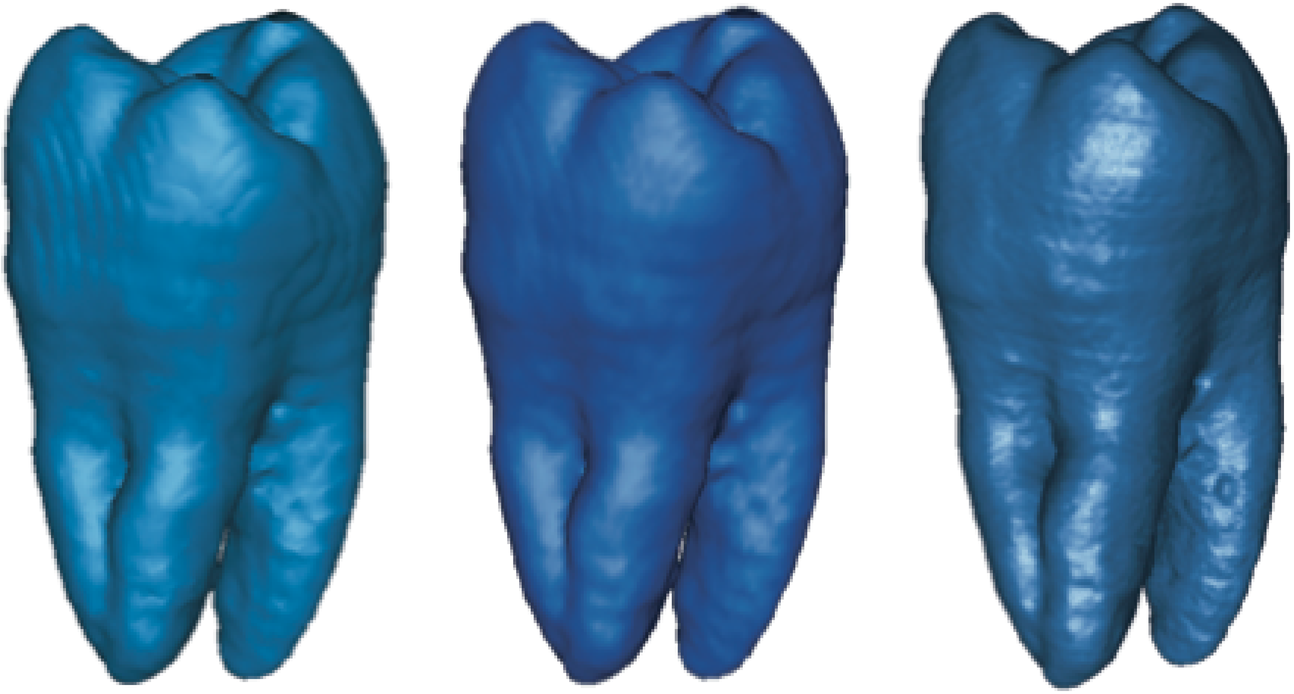
Isosurface function

Fourth example of ray function

- compute whether a given isovalue σ exists along ray

$$I(p) = \begin{cases} f(\sigma), & \exists t \in [0,T], s(t) = \sigma \\ I_0, & \text{otherwise.} \end{cases}$$

- produces same result as marching cubes, but with a higher accuracy

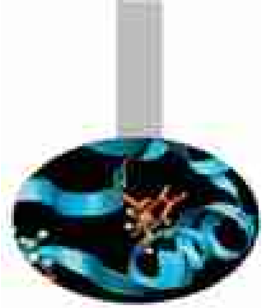


isosurface (marching cubes)

isosurface (software ray casting)

isosurface (hardware ray casting)

Composite function



Fifth example of ray function

- compute a color at each point along the ray (apply transfer function *first*)
- blend (compose) all colors to get the final pixel color (ray function=alpha blending)

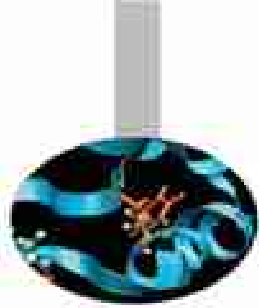
$$I(p) = F(\{ f(s(t)) \mid t \in [0,1]\})$$

transfer function (applied to all pixels along ray)

ray function (blends all colors produced by transfer function along ray)

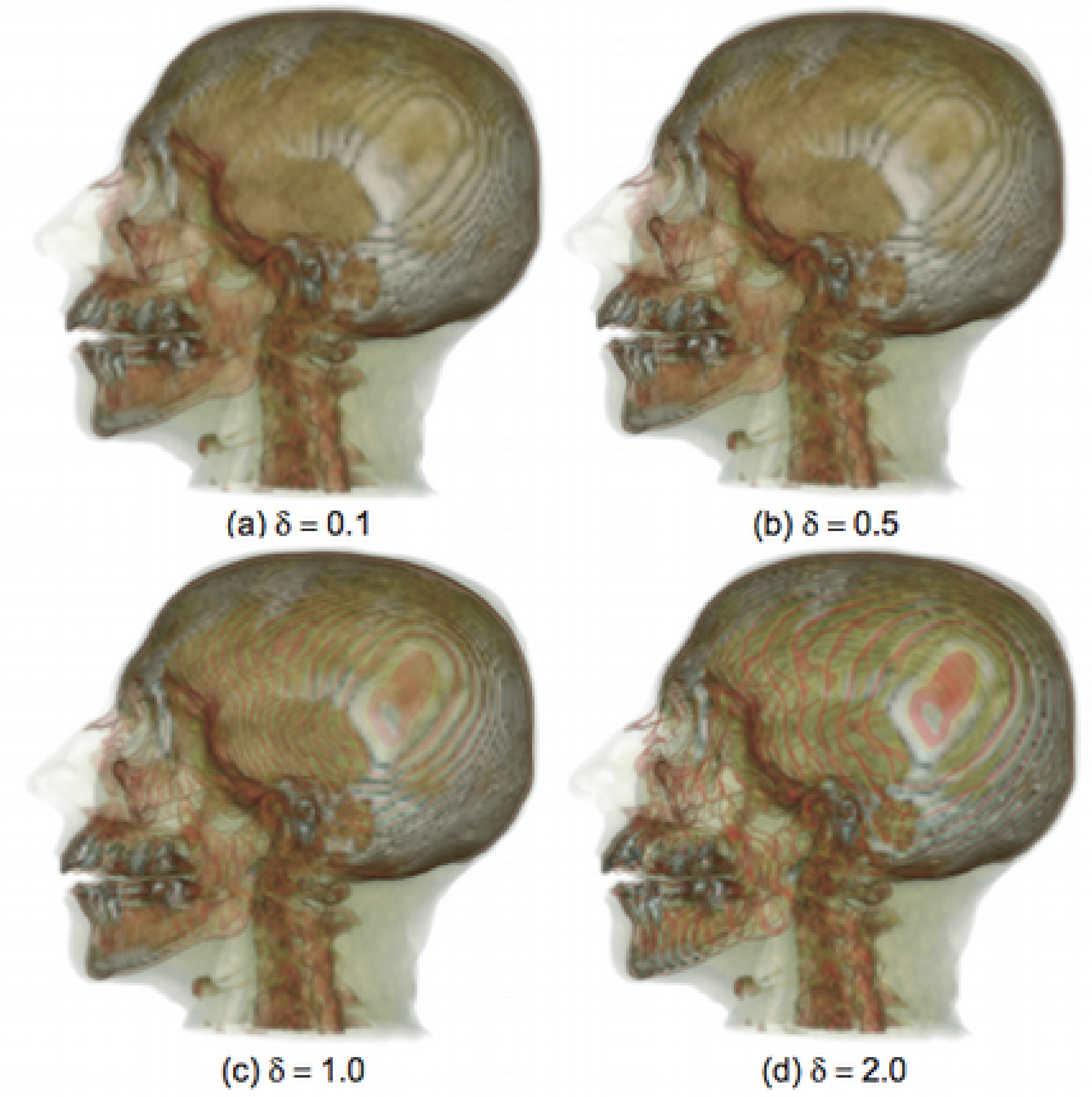
- transfer function: controls color+transparency of all material types
- ray function: blends together all material colors+transparencies along ray
- most powerful (but most computationally expensive) ray function
- allows huge range of effects (depending on type of transfer function)
- designing ‘good’ transfer functions is however non-trivial:
- **let the user change it interactively**

Implementation issues



Sampling density

- recall the ray parameterization $$q(t) = (1-t)q_0 + tq_1, \quad t \in [0,1]$$
- we need to sample along the ray (e.g. integrate, compute min/max, etc)
- how small should we take the sampling step $\delta = dt$?


(a) δ = 0.1
(b) δ = 0.5
(c) δ = 1.0
(d) δ = 2.0

Human head CT, four different δ values

- smaller δ: more accuracy
- too small δ: slow rendering

Practical guideline

- δ should never exceed a voxel size (otherwise we skip voxels while traversing the ray…)