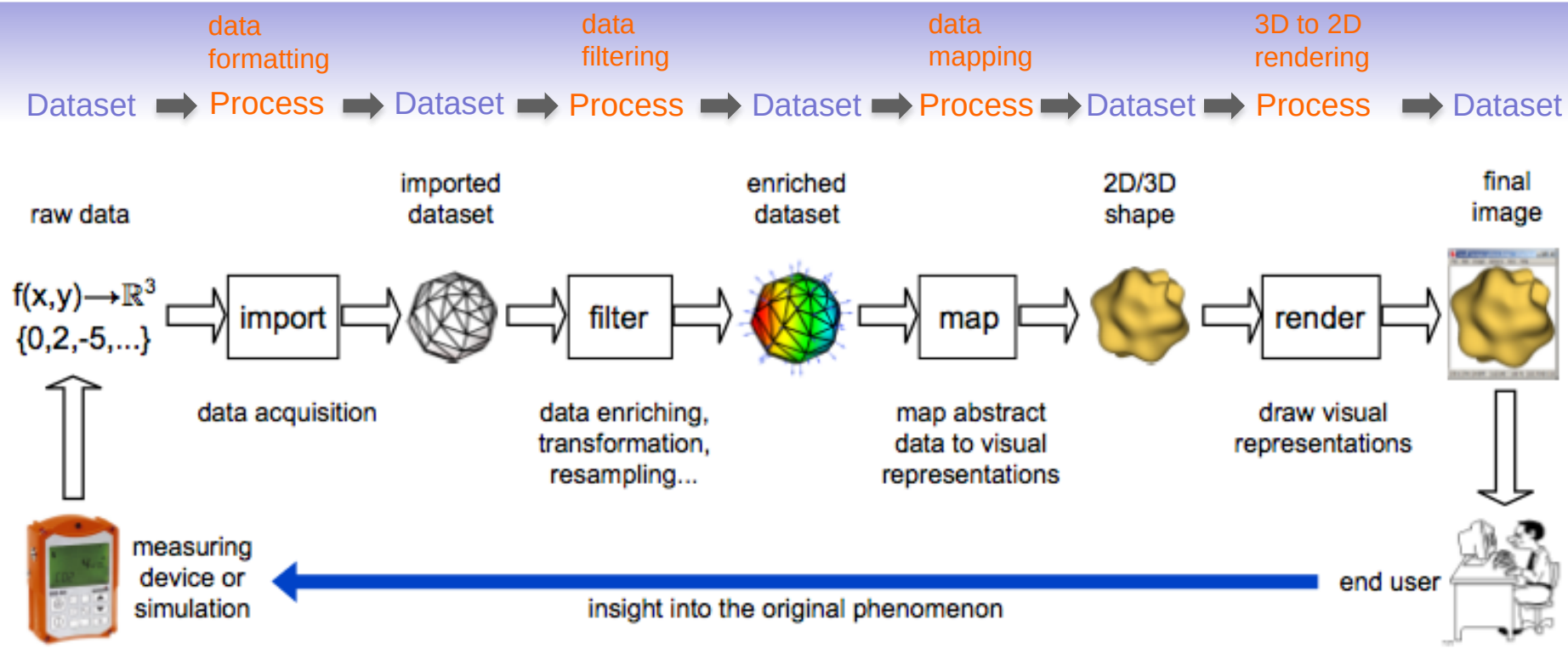
Scalar Visualization

The Visualization Pipeline



Algorithm classification

1. **Scalar algorithms**
   - operate on scalar data
   - color mapping, contouring, height plots
2. **Vector algorithms**
   - operate on vector data
   - hedgehogs, glyhps, derived quantities, stream surfaces, image-based methods
3. **Tensor algorithms**
   - operate on symmetric 3x3 tensors
   - tensor glyphs, hyperstreamlines, fiber tracing, principal component analysis
4. **Modeling algorithms**
   - change attributes and/or underlying **grid**
   - implicit functions, distance fields, cutting, selection, grid-less interpolation, grid processing

Motivation

Visualizing scalar data is frequently encountered in science, engineering, and medicine, but also in daily life. Recalling from earlier, scalar datasets, or scalar fields, represent functions $f: D \rightarrow R$, where D is usually a subset of R^2 or R^3. There exist many scalar visualization techniques, both for 2D and 3D datasets. In this chapter, we present a number of the most popular scalar visualization techniques: color mapping, contouring, and height plots.

Scalar Visualization

Color Mapping

Color mapping is a common scalar visualization technique that maps scalar data to colors, and displays the colors on the computer system. The scalar mapping is implemented by indexing into a *color lookup table*. Scalar values then serve as indices into this lookup table.

The lookup table holds an array of colors. Associated with the table is a minimum and maximum scalar range into which the scalars are mapped. Scalar values greater than the maximum are clamped to the maximum color, scalar values less than the minimum are clamped to the minimum value.

Color mapping

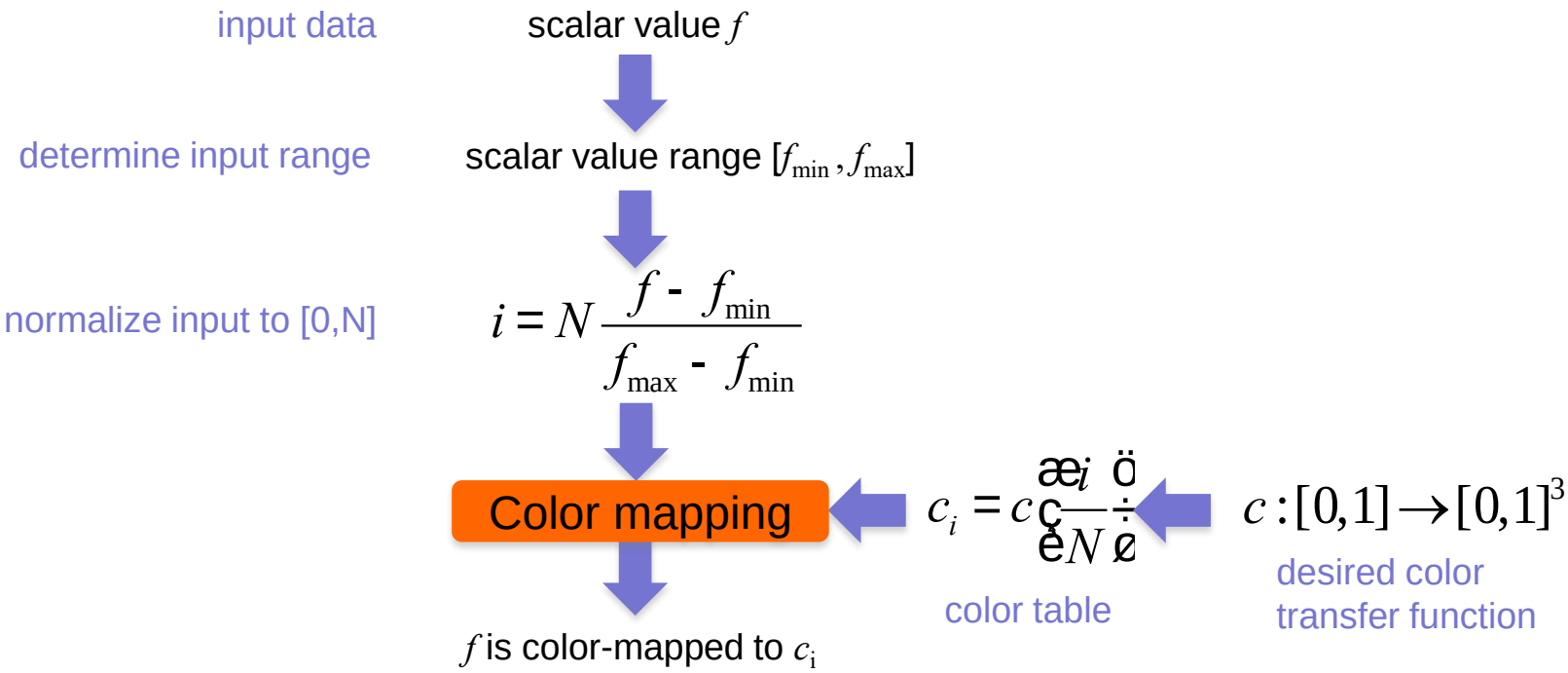
Basic idea

•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



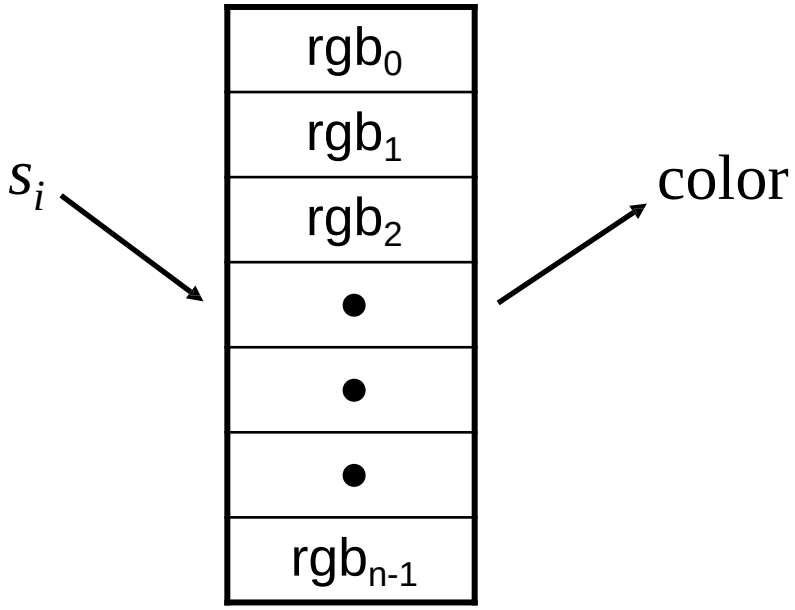
Scalar Visualization

Then, for each scalar value s_i, the index i into the color able with n entries is given as:

$$s_i < \min \quad : \quad i = 0$$

$$s_i > \max \quad : i = n - 1$$

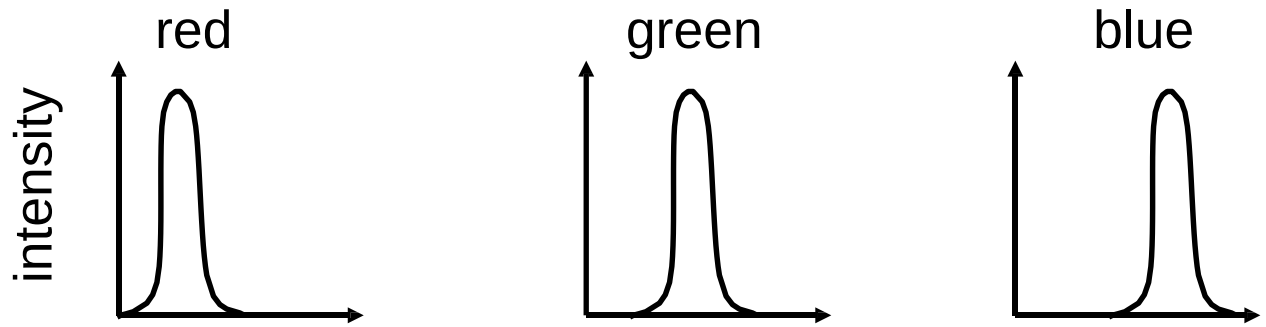
otherwise: $i = n \cdot \left(\frac{s_i - \min}{\max - \min} \right)$



Scalar Visualization

Transfer Functions

A more general form of the lookup table is called *transfer function*. A transfer function is any expression that maps scalar values into a color specification. For example, a function can be used to map scalar values into separate intensity values for the red, green, and blue components.



Scalar Visualization

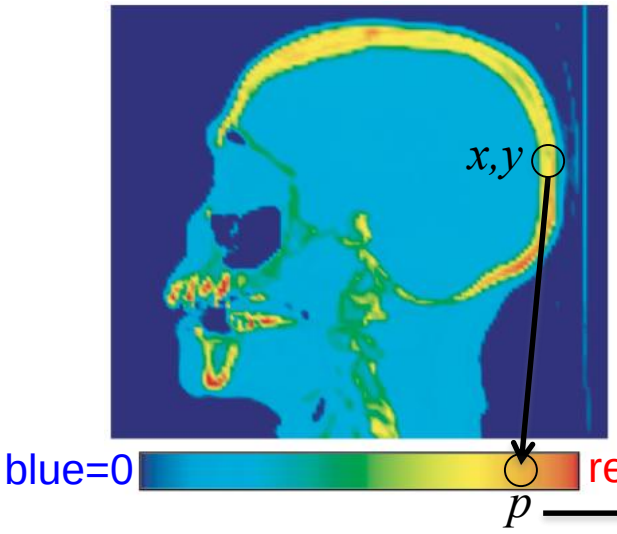
We can also use transfer functions to map scalar data into other information such as local transparency.

A lookup table is a discrete sampling of a transfer function. We can create a lookup table from any transfer function by sampling the transfer function at a set of discrete points.

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

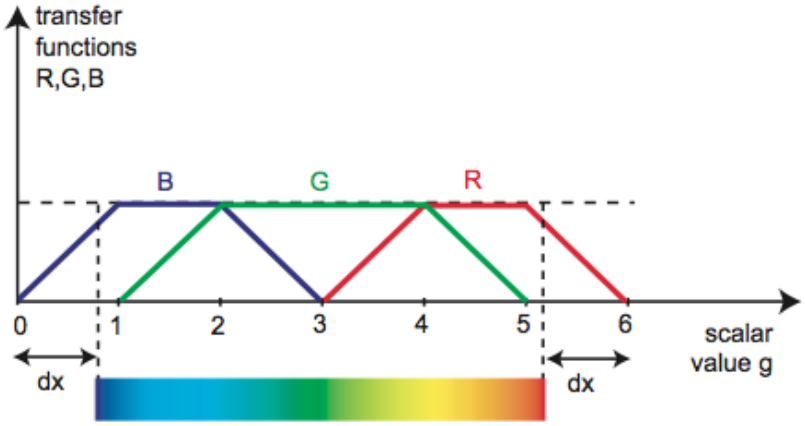
Invert mapping:

1. look at some point (x,y) in the image $\rightarrow$ 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);
}
```

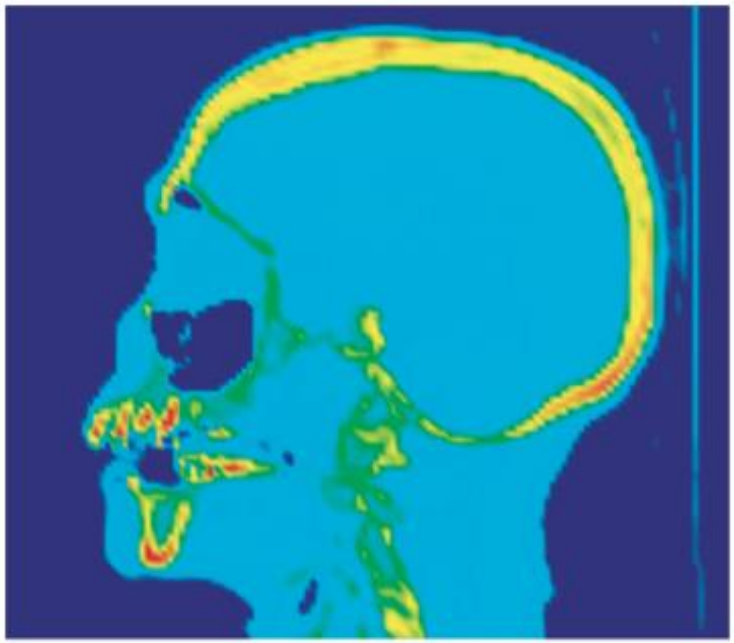
Gray-value colormap

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

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

Gray-value colormap

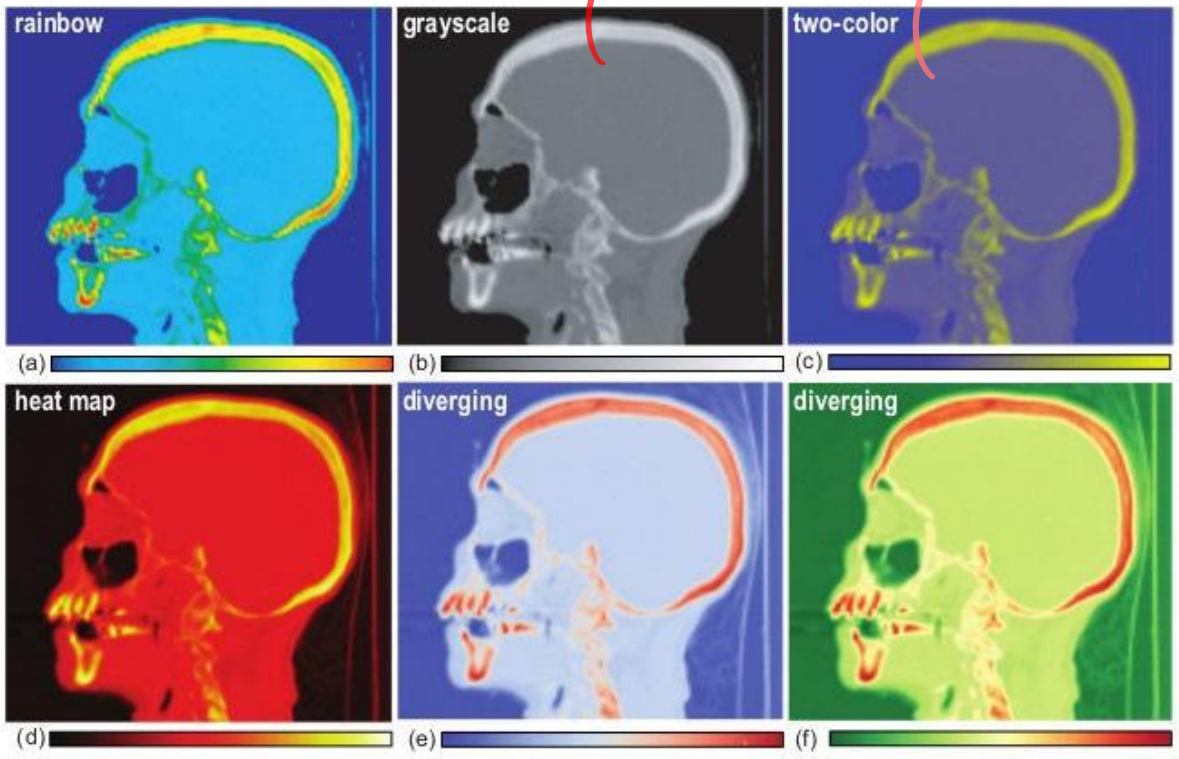
- white = hard tissues (bone)
- gray = soft tissues (flesh)
- black = air



Rainbow colormap

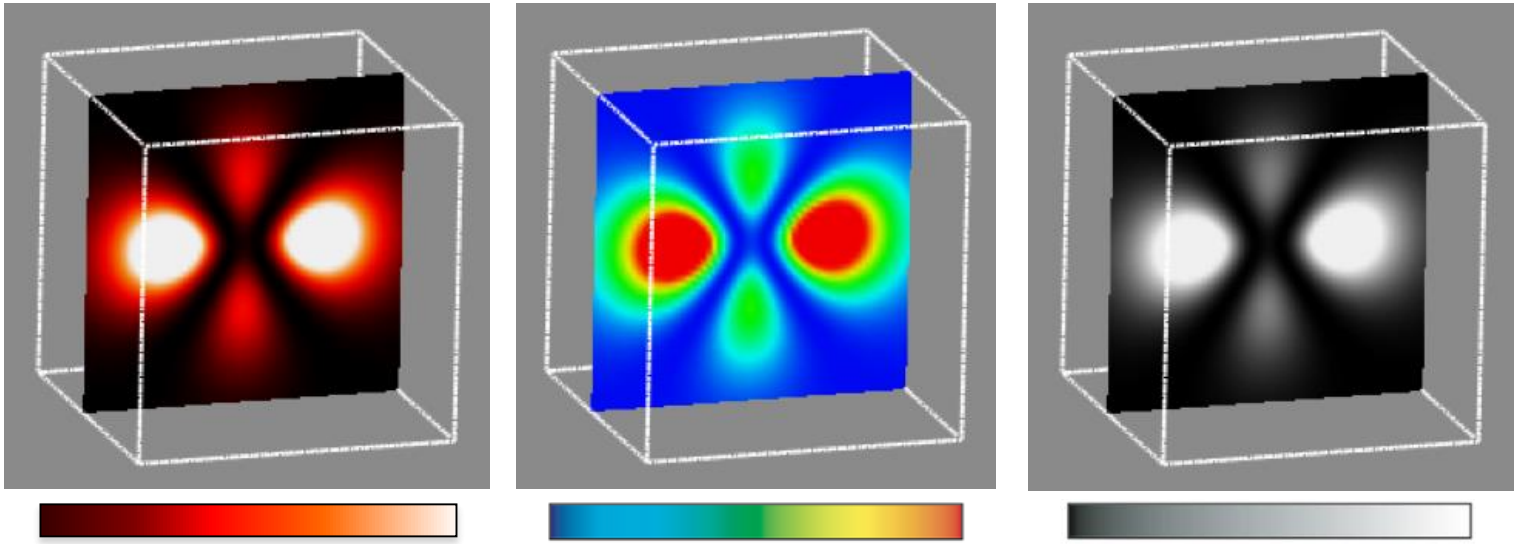
- red = hard tissues (bone)
- blue = air
- other colors = soft tissues

- Other colormap designs



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

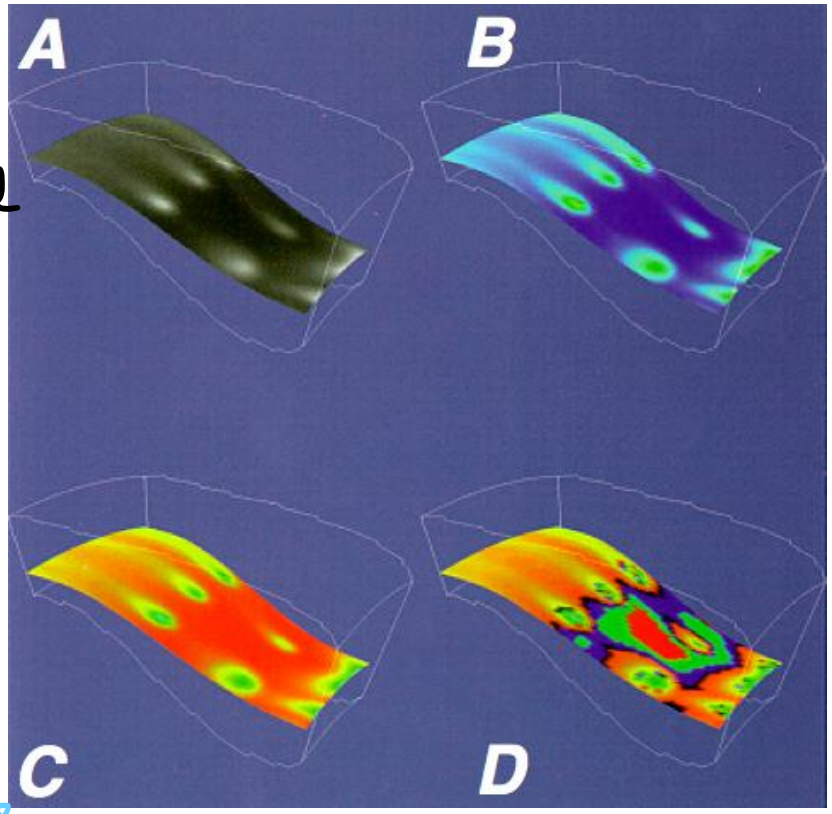
Gray-value colormap
- maxima are highlighted well
- lower values are unclear

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 values are not differentiated well.



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

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

#There may be people having color blindness

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

All the values are equally spread

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

Colormap design techniques

HSV ⇒ Hue Saturation Value

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 [brightness]

scalar value ~ H; S,V not used [hue]

scalar value ~ H,V; S not used [hue & color value]

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)

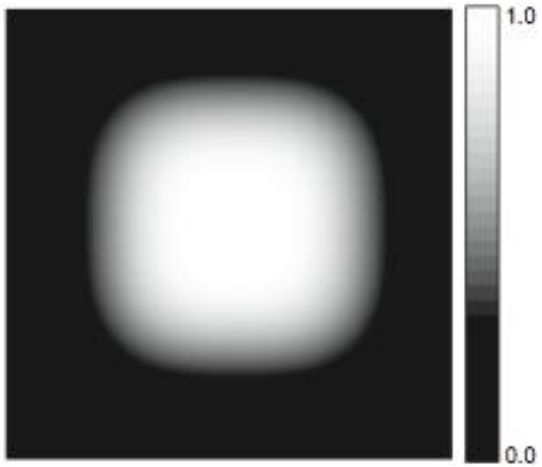
Colormap design techniques

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

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

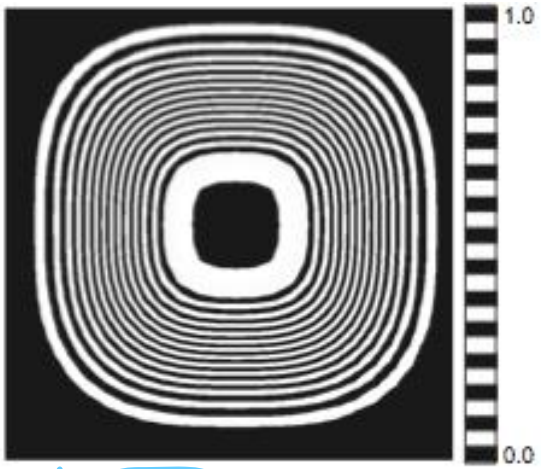
2D function $f(x,y) = e^{-10(x^4+y^4)}$

high and low



Gray-scale colormap

- highlights plateaus
- value transitions hard to see



Variations

Zebra colormap

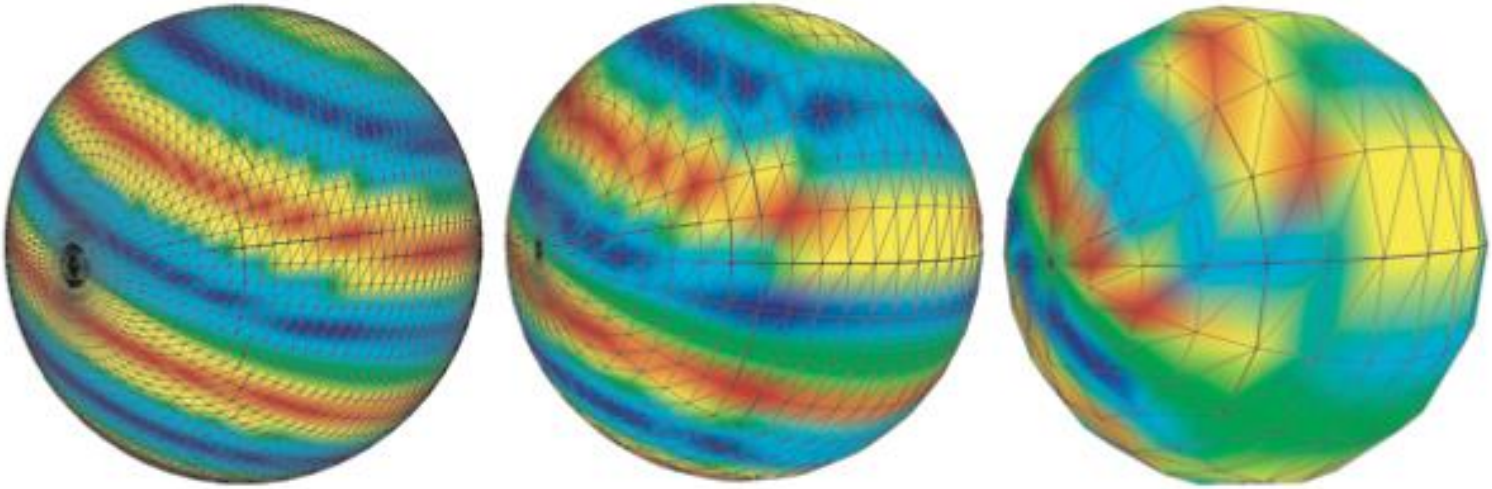
- highlights value variations (1st derivative)
- dense, thin bands: fast variation
- thick bands: slow variation

Colormap implementation details

Where to apply the colormap?

- per grid-cell vertex

2D periodic high-frequency function



64x64 points 32x32 points 16x16 points

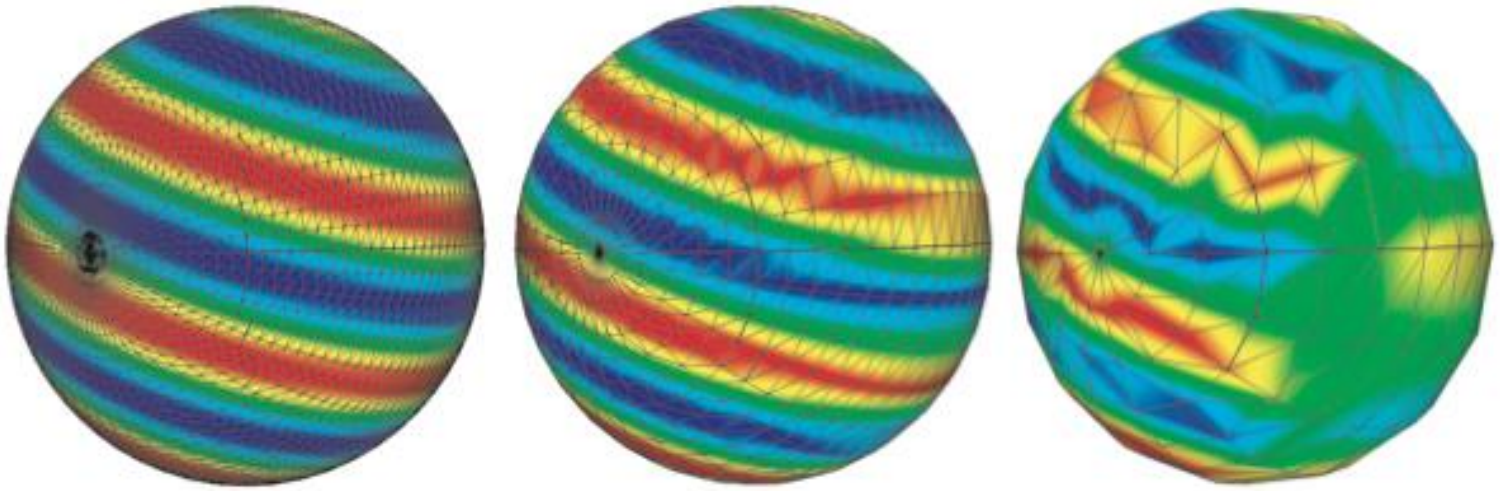
As we decrease the sampling frequency, strong colormapping artifacts appear

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

Sometimes red + blue ⇒ purple

Outside the rainbow colors

Explanation

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

sampling & applying the color mapping

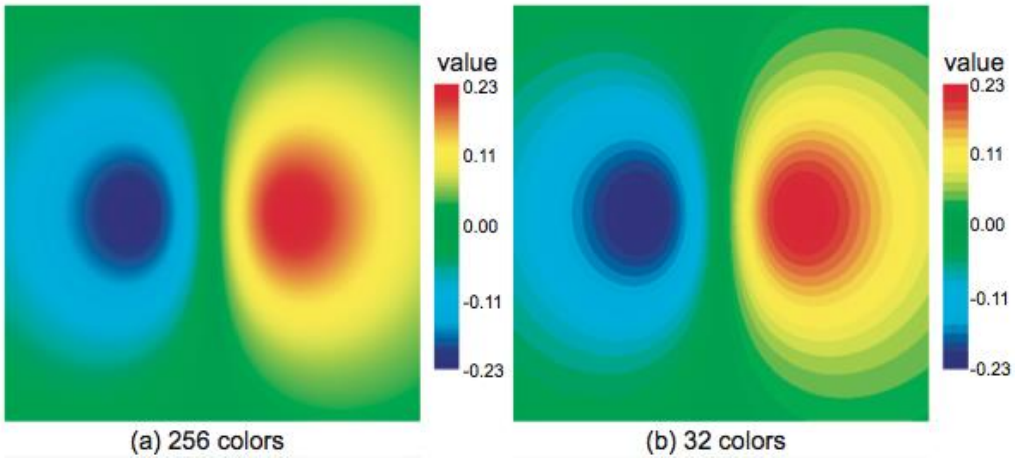
See Sec. 5.2 for details

Color banding

How many distinct colors N to use in a color table?

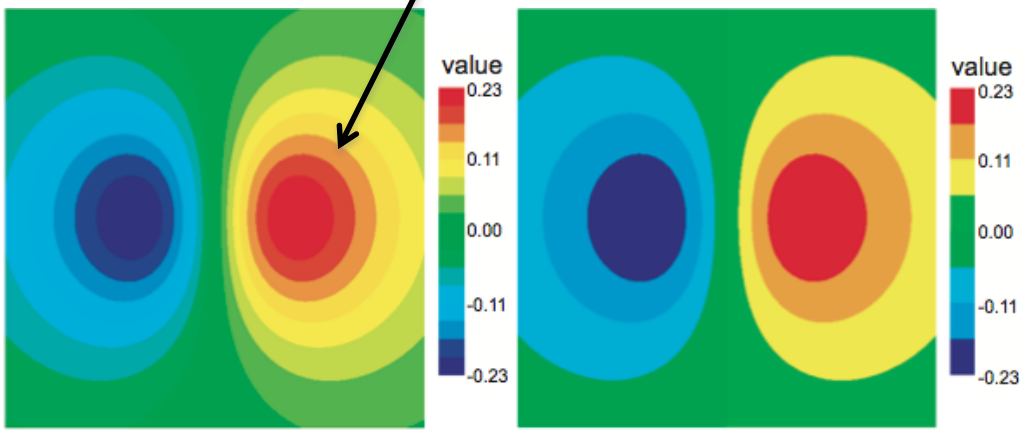
- more colors: better sampled c thus smoother results
- fewer colors: color banding appears

scalar value ⇒ continuous
then the bands color should be continuous. But here only distinct bands one seen



(a) 256 colors (b) 32 colors

color banding



(c) 16 colors (d) 8 colors

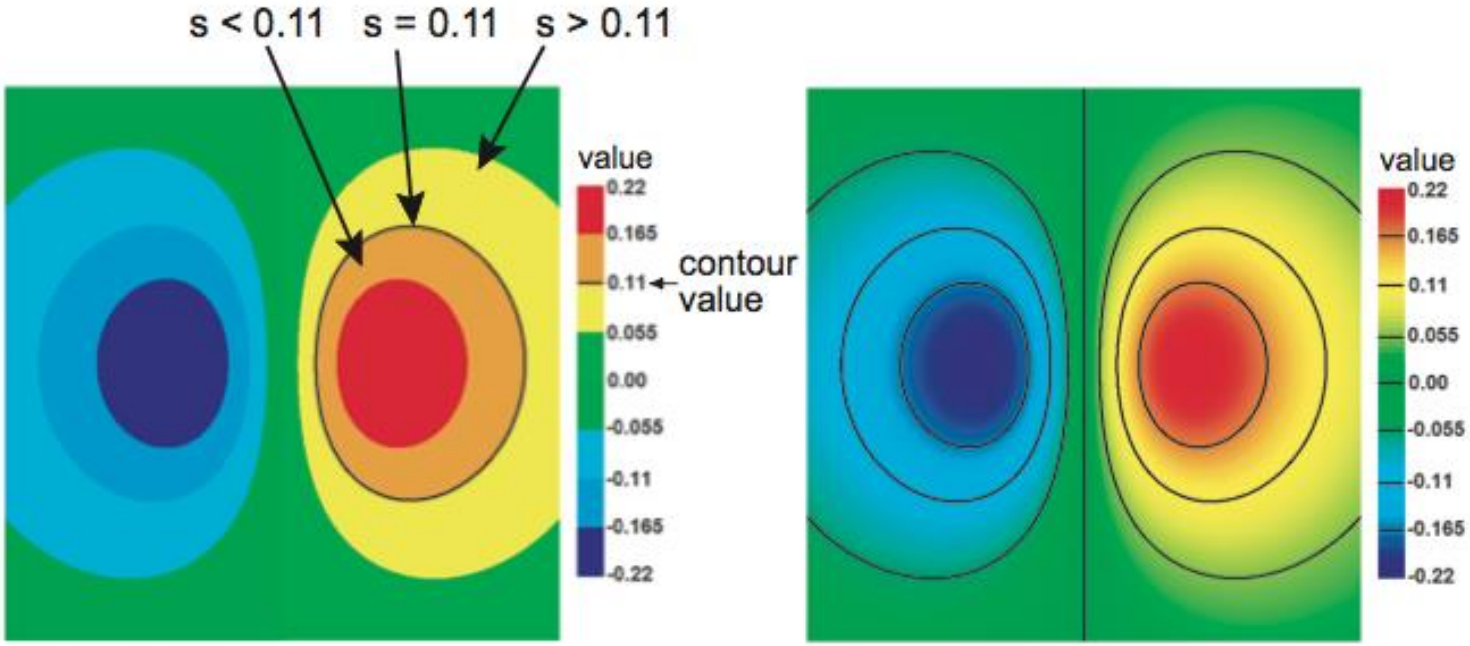
Question

- can we see sharp color banding with per-vertex colormapping? Why (not)?

Contouring

How to see where some given values appear in a dataset?

- recall color banding
- a transition separating two consecutive bands = a contour

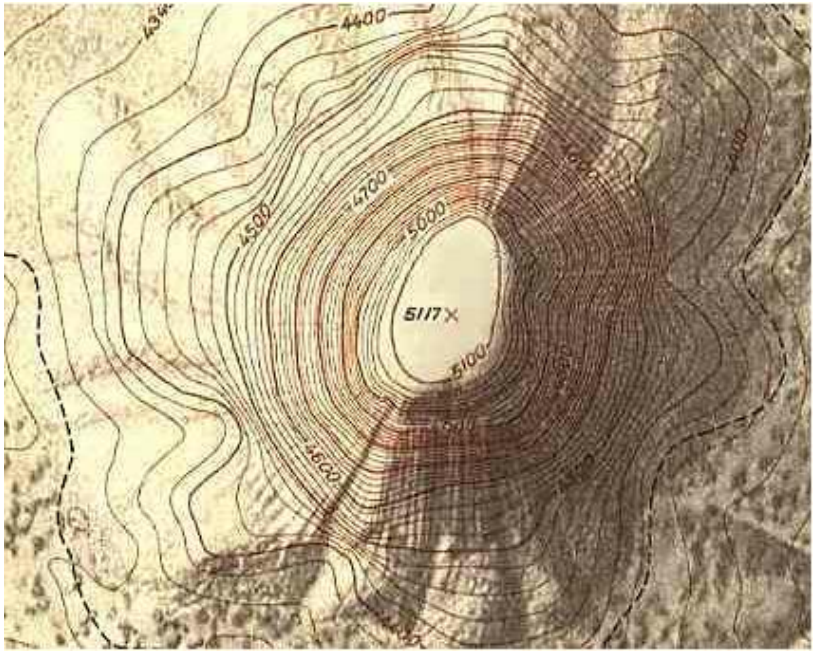


contour = all points having the scalar value $s = 0.11$

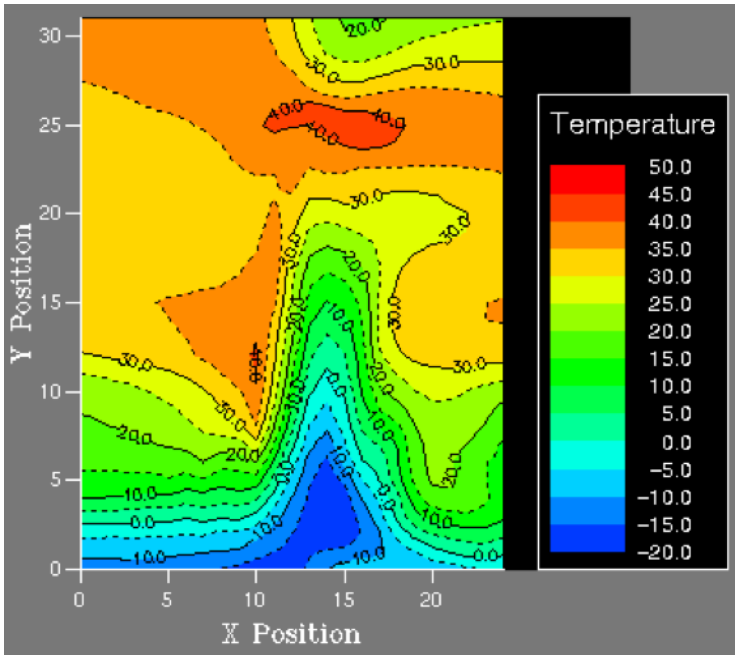
seven different contours, equidistant in value space

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

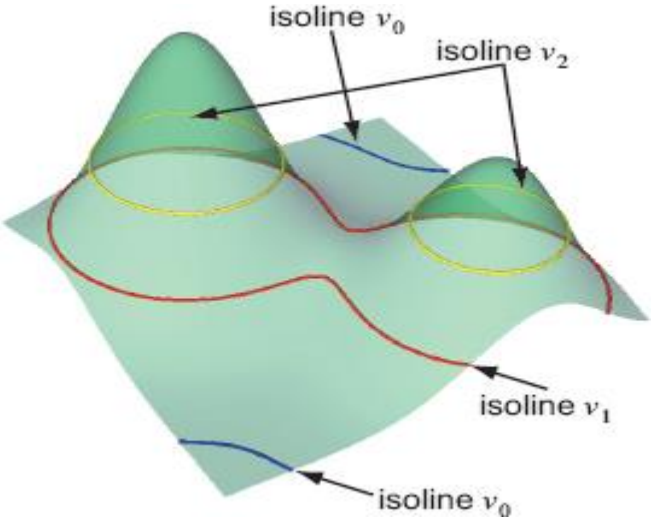
Definition

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

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

Contours never (self-)intersect, thus are nested

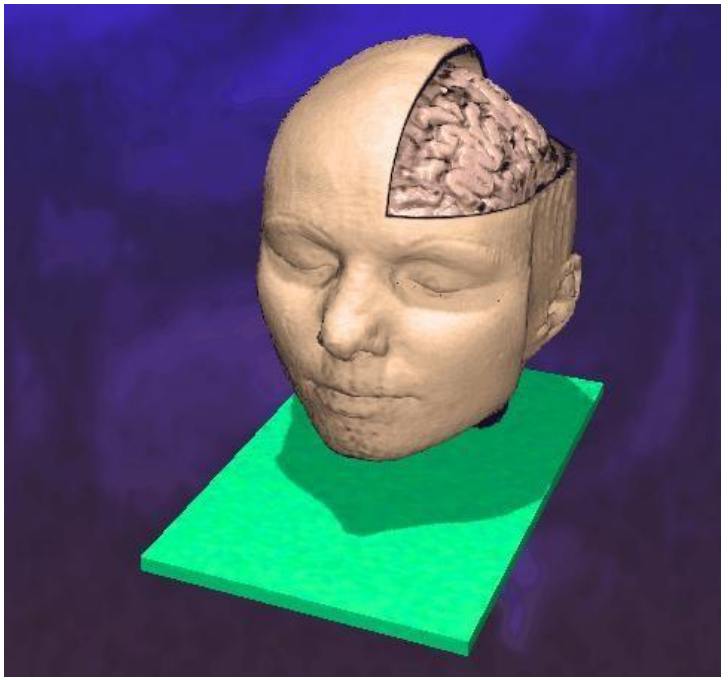
Contours cut D into values smaller resp. larger than the isovalue



these iso-lines will never intersect

Scalar Visualization

Three-dimensional contours are called *isosurfaces*, and can be approximated by many polygonal primitives. Examples of isosurfaces include constant medical image intensity corresponding to body tissues such as skin, bone, or other organs. (The corresponding isovalue for the same tissue, however, is not necessarily constant among several different scans.) Other abstract isosurfaces such as surfaces of constant pressure or temperature in fluid flow also may be created.



Contour properties

Contours are always orthogonal to the scalar value's gradient

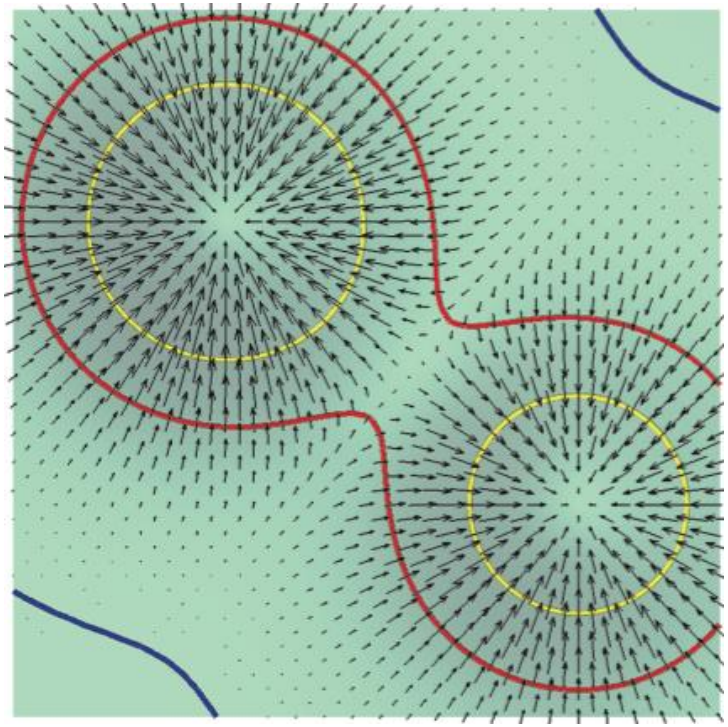
•why? Recall definitions

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

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

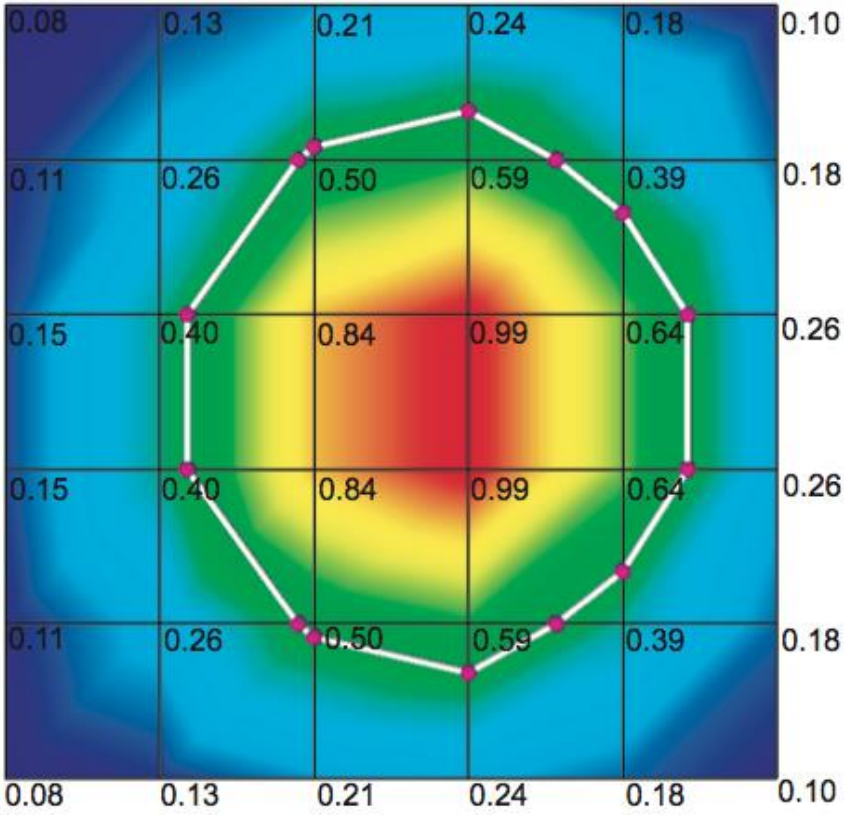


gradient of a scalar field (drawn with arrows) is orthogonal to contours

Scalar Visualization

First, we will focus on 2-D contours and how to generate such an isocontour for a given isovalue. Consider a regular grid with scalar values assigned to the grid nodes. Contouring always begins by selecting a scalar value (the isovalue or contourvalue) that corresponds to the contour lines or surface generated. Assuming linear interpolation on the regular grid, we can identify those locations on the edges of the regular grid where the data assumes the isovalue. For example, if an edge has scalar values 10 and 0 at its two end points, and if we are trying to generate a contour line of value 5, then the contour passes through the midpoint of that edge.

Basic contouring algorithm



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

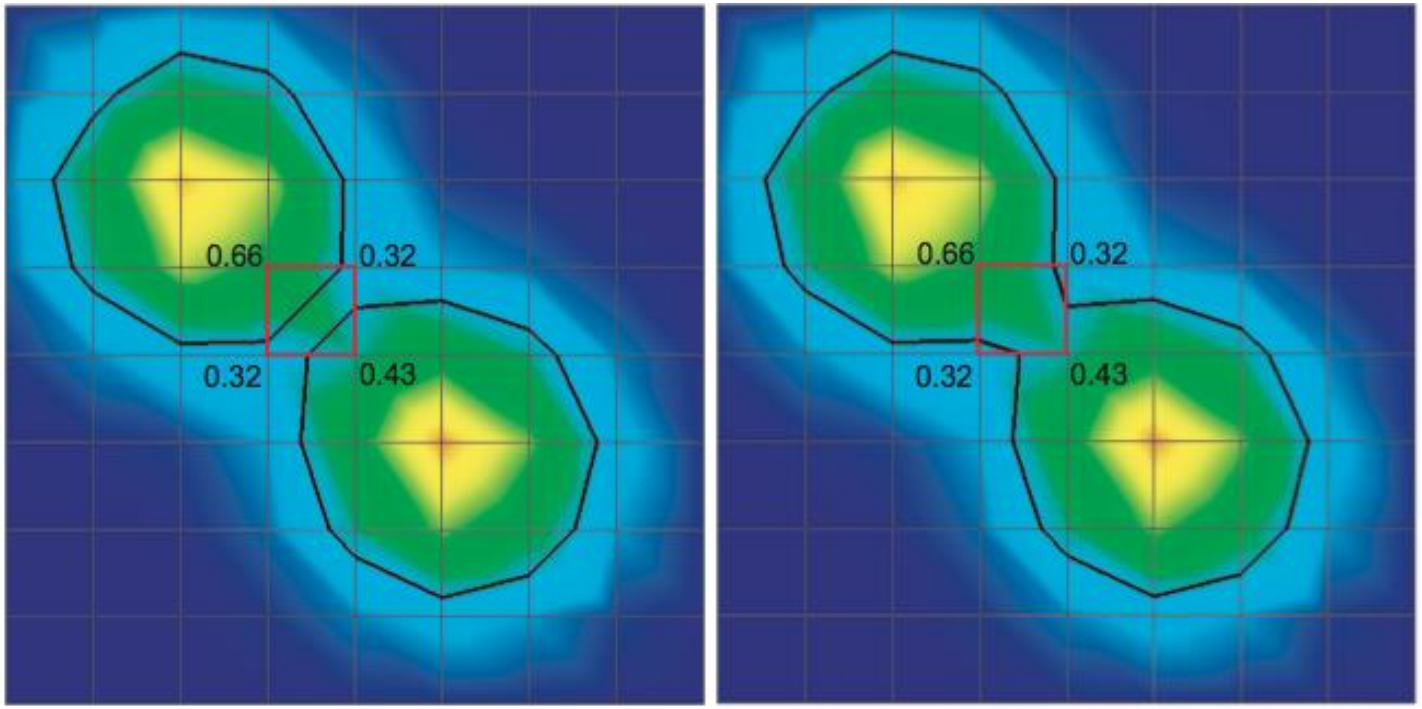
Works OK but it is

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

Contouring ambiguity

Each edge of the red cell intersects the contour

- which is the right contour result?

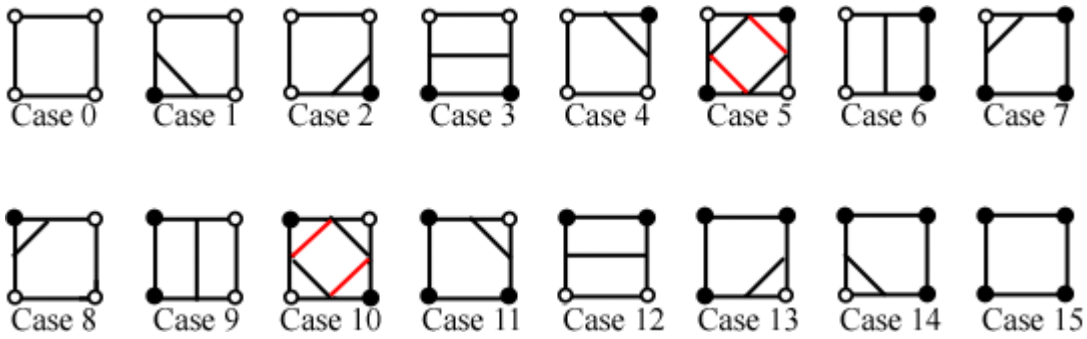


- we could discriminate only if we had higher-level information (e.g. topology)
- at cell level, we cannot determine more

Scalar Visualization

Marching Squares

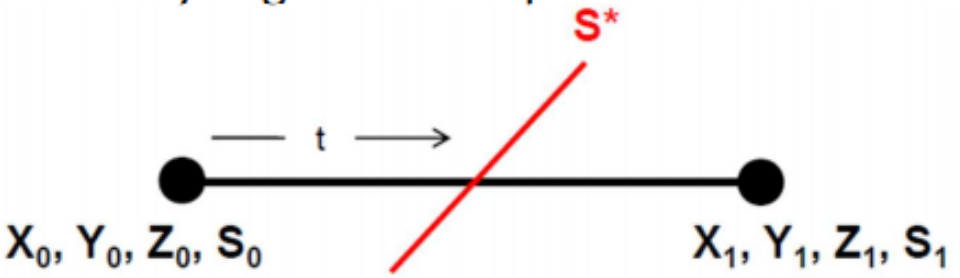
Another approach uses a divide and conquer technique, treating cells independently. This *marching squares* algorithm assumes that a contour can only pass through a cell in a finite number of ways due to the linear interpolation used. A case table is constructed that enumerates all possible topological states of a cell, given combinations of scalar values at the cell points.



(dark vertices indicate scalar value is above isovalue)

Marching Squares

Does S* cross any edges of this square?



Linearly interpolating the scalar value from node 0 to node 1 gives:

$$S = (1-t)S_0 + tS_1 = S_0 + t(S_1 - S_0) \quad \textit{where } 0. \leq t \leq 1.$$

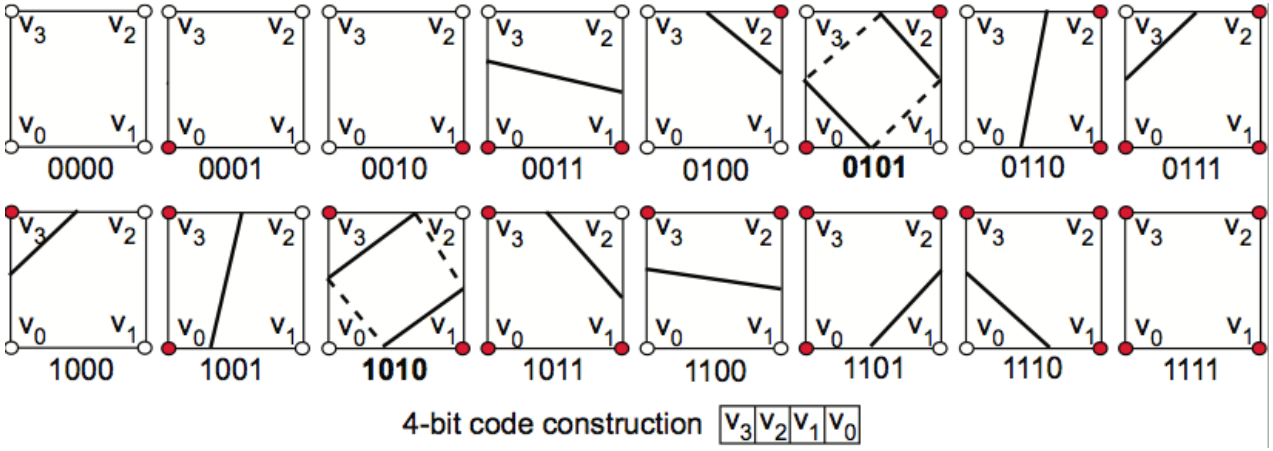
Setting this interpolated S equal to S* and solving for t gives:

$$t^* = \frac{S^* - S_0}{S_1 - S_0}$$

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

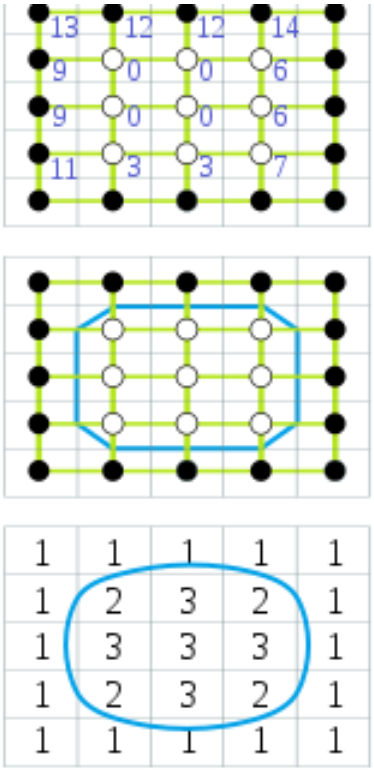


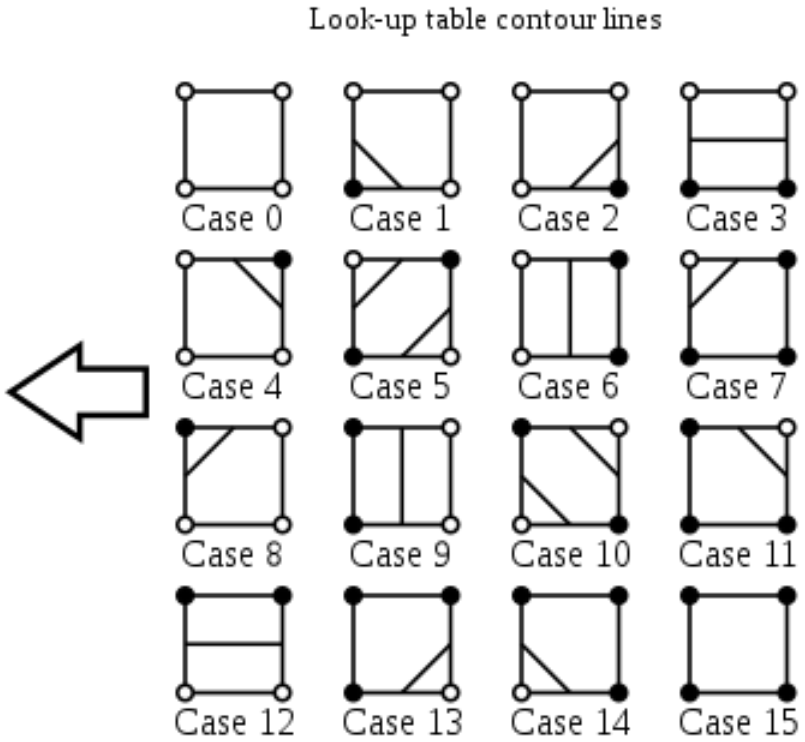
e.g.
inside: $f > f_0$
outside: $f \leq f_0$

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')


13 12 12 14
9 0 0 6
9 0 0 6
11 3 3 7
1 1 1 1 1
1 2 3 2 1
1 3 3 3 1
1 2 3 2 1
1 1 1 1 1


Look-up table contour lines
Case 0
Case 1
Case 2
Case 3
Case 4
Case 5
Case 6
Case 7
Case 8
Case 9
Case 10
Case 11
Case 12
Case 13
Case 14
Case 15

for each cell c_i of the dataset

{

int *index* = 0 ;

for (each vertex v_j of c_i)

store the inside/outside state of v_j in bit j of *index;*

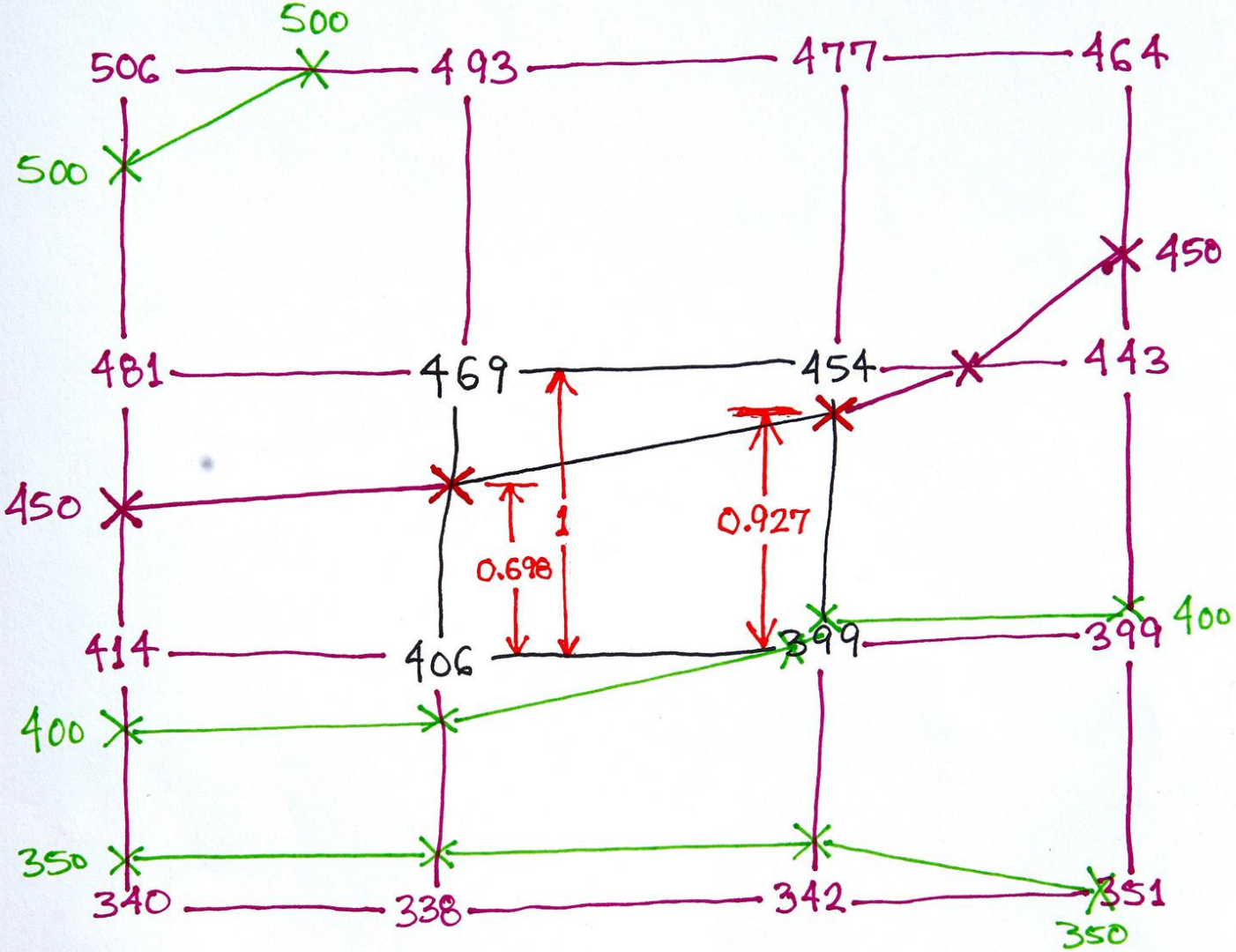
select the optimized code from the case table using *index;*

for (all cell edges e_j of the selected case)

intersect e_j with isovalue v using Equation $\frac{s_i - s_{\min}}{s_{\max} - s_{\min}}$

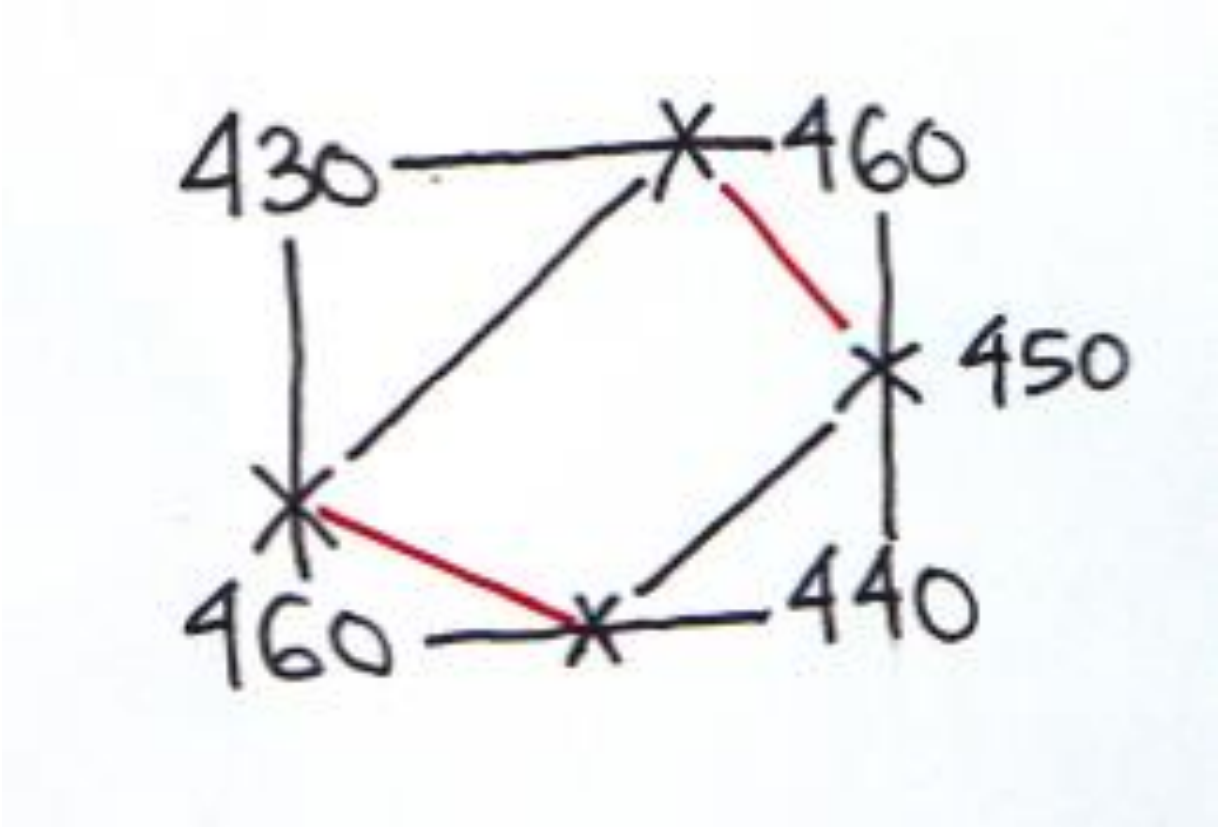
construct line segments from these intersections;

}



Ex: $\frac{450-406}{469-406} = \frac{44}{63} = 0.698$

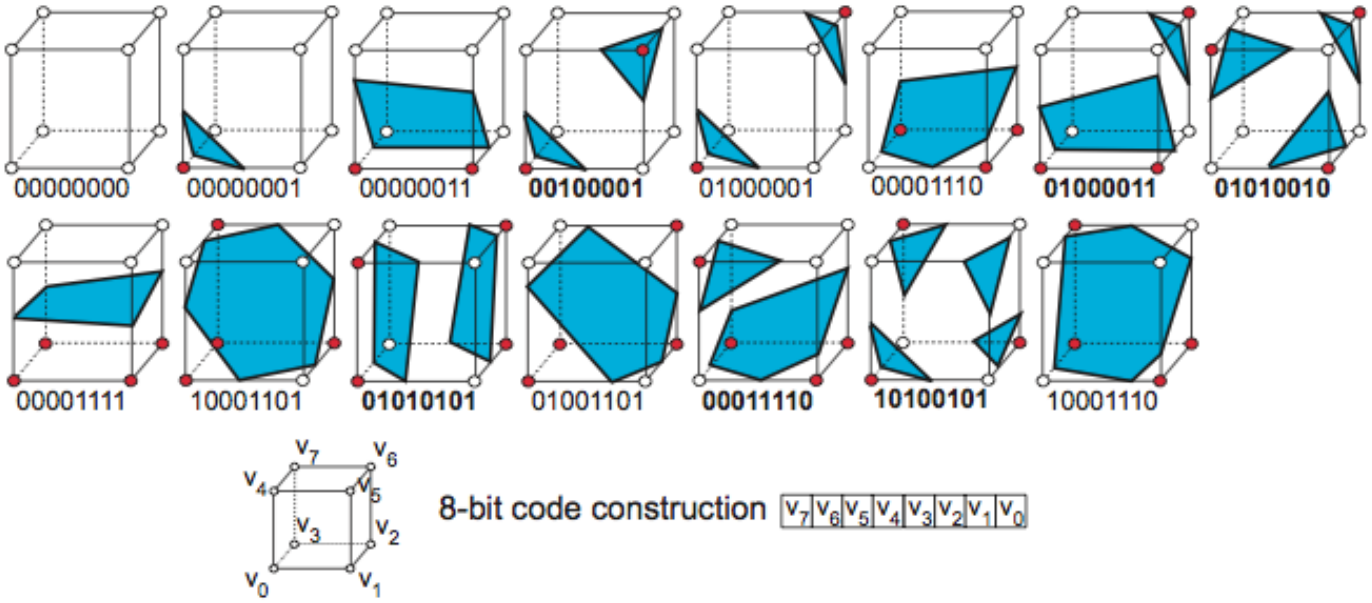
Contouring ambiguity



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: $f \leq f_0$

2. Process all dataset cells

- for each cell, use codes as pointers into a jump-table with 15 cases (reduce the 2^8=256 cases to 8 by symmetry considerations)

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
- directly from data

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

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

Scalar Visualization

Height Plots

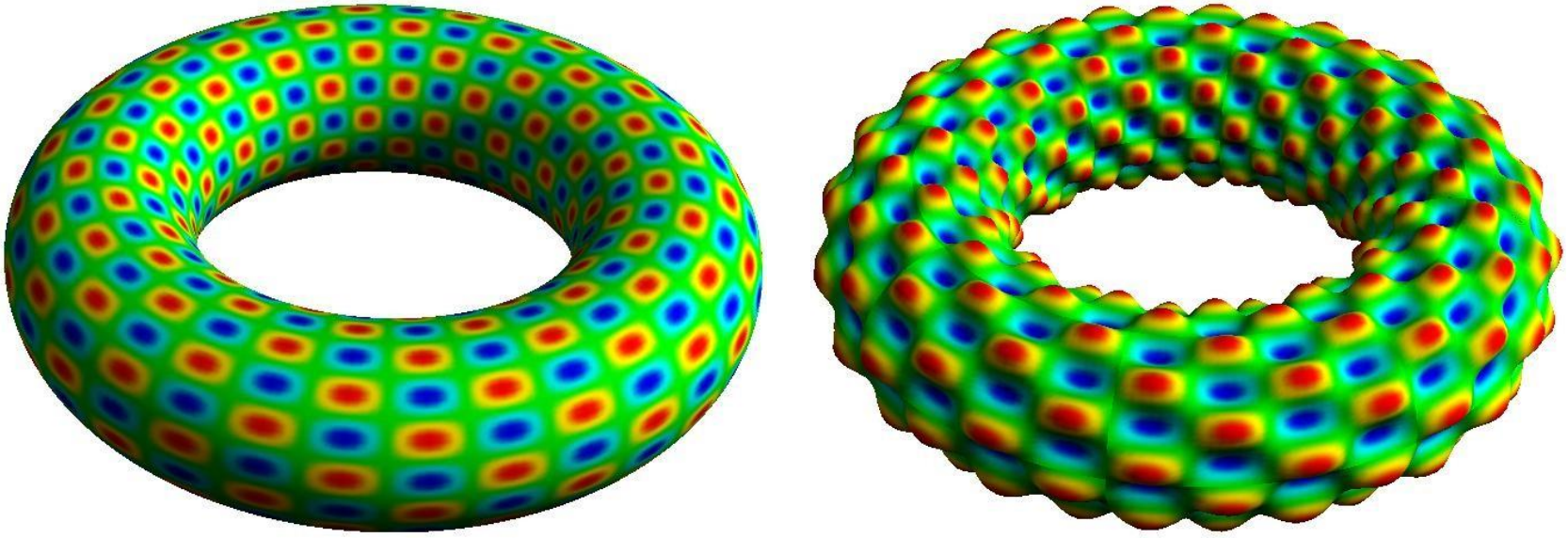
Height plots, also called elevation or carpet plots, were introduced by our first example during the introduction. Given a two-dimensional surface $D_s \in D$, part of a scalar dataset D, height plots cam be described by the mapping operation

$$m: D_s \rightarrow D,\ m(x)=x+s(x)n(x),\ \forall x \in D_s$$

Where $s(x)$ is the scalar value of D at the point x and $n(x)$ is the normal to the surface D_s at x. In other words, the height plot mapping operation “warps” a given surface D_s included in the data set along the surface normal with a factor proportional to the scalar values.

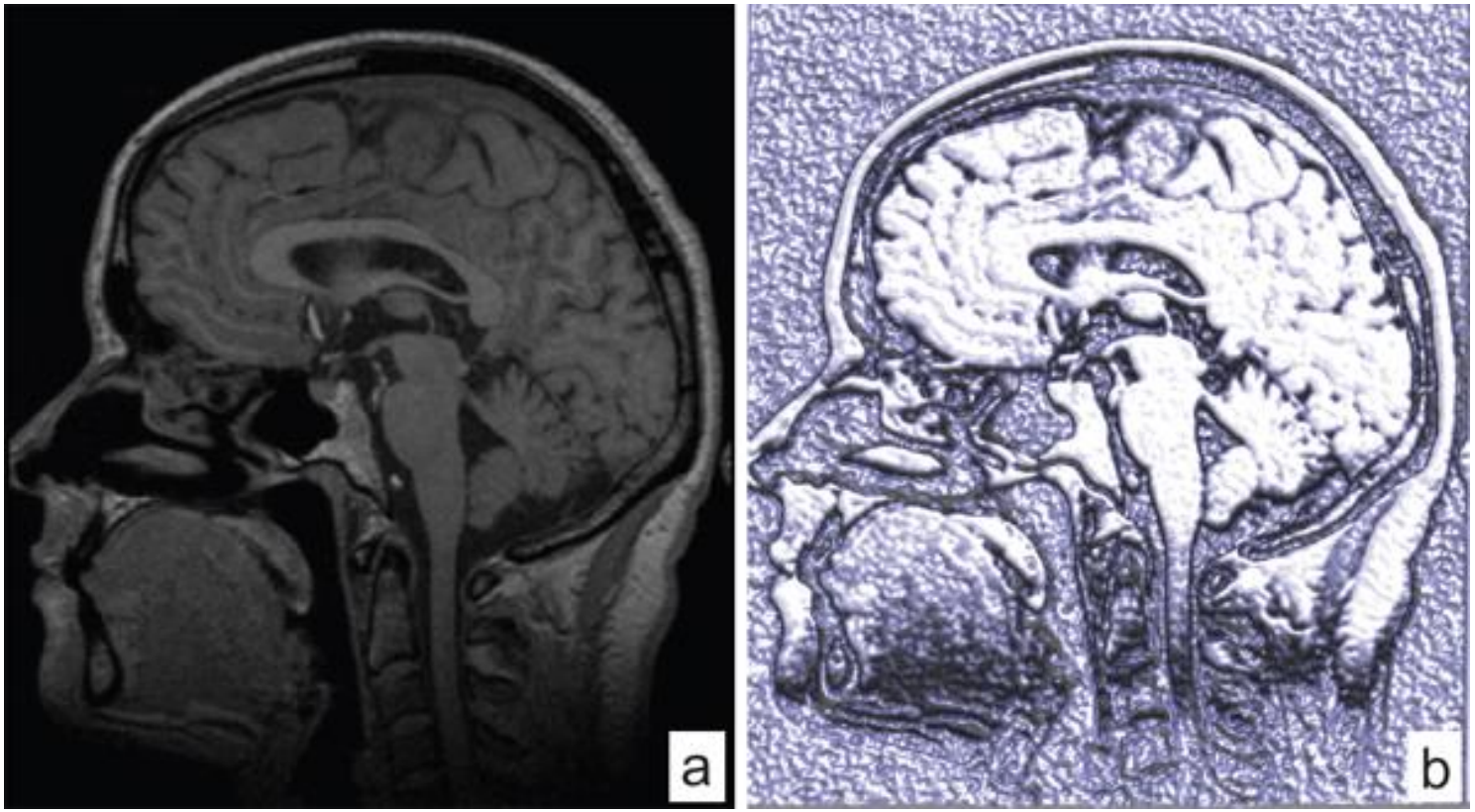
Scalar Visualization

Example: a torus surface (a) and its “warped” variant with the height corresponding to the scalar value.



(a) (b)

Images courtesy of Alexandru Telea


a
b

Conclusion

- Visualizing scalar data
  - Color mapping
    - Assign a color as a function of the scalar value at each point of a given domain
  - Contouring
    - Displaying all points with a given 2D or 3D domain that have a given scalar value
  - Height plots
    - Deform the scalar dataset domain in a given direction as a function of the scalar data
- Advantage
  - Produce intuitive results
  - Easily understood by the vast majority of users
  - Simple to implement
- Disadvantage
  - A number of restrictions
    - One or two dimensional scalar dataset
    - We want to visualize the scalar values of ALL, not just a few of the data points of a 3D dataset
- Other scalar visualization method
  - Specific method for data over images or volumes