For each point in a scalar field, how many different iso-contours can go through it?
For each point in a scalar field, how many different iso-contours can go through it?

Only one!

Why?
For each point in a scalar field, how many different iso-contours can go through it?

Only one!

Why?

Different iso-contours (corresponding to different iso-values) cannot intersect with each other!!!
What if $S^*=M$?
Scalar Field Visualization – 3D

Iso-surfacing
Iso-Surfaces: Applications

A contour line is often called an iso-line, that is a line of equal value. When hiking, for example, if you could walk along a single contour line of the terrain, you would remain at the same elevation.

An iso-surface is the same idea, only in 3D. It is a surface of equal value. If you could be a fly walking on the iso-surface, you would always experience the same scalar value (e.g., temperature).

Sometimes the shapes of the iso-surfaces have a physical meaning, such as bone, skin, different layers of earth etc. (e.g., the left example above). Sometimes the shape just helps turn an abstract notion into something physical to help us gain insight (e.g. the other two examples).
This is useful, for example, if we have passed an oblique cutting plane through a 3D mesh of points and are trying to interpolate scalar values from the 3D mesh to the 2D plane.
3D Cut Planes
Reduce 3D volume to 2D slices
Here, we focus on axis-aligned cut planes, assuming the 3D volume is also axis-aligned.
Here, we focus on axis-aligned cut planes, assuming the 3D volume is also axis-aligned.

Let us concern with a uniform volume (or 3D image), which consists of individual voxels. Its dimension is \( NX \times NY \times NZ \).
Here, we focus on axis-aligned cut planes, assuming the 3D volume is also axis-aligned. Let us concern with a uniform volume (or 3D image), which consists of individual voxels. Its dimension is NX*NY*NZ. Now let us look at z dimension.
Here, we focus on axis-aligned cut planes, assuming the 3D volume is also axis-aligned.

Let us concern with a uniform volume (or 3D image), which consists of individual voxels. Its dimension is $NX \times NY \times NZ$.

Now let us look at $z$ dimension.

$z=0$ defines an $XY$ planes with all voxels whose $z$ index is zero.
Here, we focus on axis-aligned cut planes, assuming the 3D volume is also axis-aligned.

Let us concern with a uniform volume (or 3D image), which consists of individual voxels. Its dimension is $NX \times NY \times NZ$.

Now let us look at the $z$ dimension with different $z$ values (indices), you get different $XY$ planes.
Here, we focus on axis-aligned cut planes, assuming the 3D volume is also axis-aligned.

Let us concern with a uniform volume (or 3D image), which consists of individual voxels. Its dimension is \( NX \times NY \times NZ \).

Now let us look at the \( z \) dimension with different \( z \) values (indices), you get different \( XY \) planes.
Here, we focus on axis-aligned cut planes, assuming the 3D volume is also axis-aligned.

Let us concern with a uniform volume (or 3D image), which consists of individual voxels. Its dimension is $\text{NX} \times \text{NY} \times \text{NZ}$.

Now let us look at the $z$ dimension with different $z$ values (indices), you get different XY planes.
Here, we focus on axis-aligned cut planes, assuming the 3D volume is also axis-aligned.

Let us concern with a uniform volume (or 3D image), which consists of individual voxels. Its dimension is $NX \times NY \times NZ$.

Now let us look at the $z$ dimension.

This shows an $XY$ cut plane (perpendicular to $z$) can be constructed by specifying a $z$ value between $[0, NZ-1]$. 
In VTK

Use the following to get the dimension of the 3D image data or structured grid

```python
dim = reader.GetOutput().GetDimensions()
```

```python
# Create a mapper and assign it to the corresponding reader
xy_plane_Colors = vtk.vtkImageMapToColors()
xy_plane_Colors.SetInputConnection(reader.GetOutputPort())
xy_plane_Colors.SetLookupTable([you color look up table])
xy_plane_Colors.Update()
```
In VTK

Use the following to get the dimension of the 3D image data or structured grid

```python
    dim = reader.GetOutput().GetDimensions()
```

```python
# Create a mapper and assign it to the corresponding reader
xy_plane_Colors = vtk.vtkImageMapToColors()
xxy_plane_Colors.SetInputConnection(reader.GetOutputPort())
xxy_plane_Colors.SetLookupTable([you color look up table])
xxy_plane_Colors.Update()
```

```python
# Create an image actor for the XY plane
xy_plane = vtk.vtkImageActor()
xxy_plane.GetMapper().SetInputConnection(xy_plane_Colors.GetOutputPort())
xxy_plane.SetDisplayExtent(0, dim[0]-1, 0, dim[1]-1, current_zID, current_zID)
# Current_zID is a user-input integer within the range of [0, zdim-1]
```
In VTK

Use the following to get the dimension of the 3D image data or structured grid

\[
dim = reader.GetOutput().GetDimensions()
\]

# Create a mapper and assign it to the corresponding reader
xy_plane_Colors = vtk.vtkImageMapToColors()
xy_plane_Colors.SetInputConnection(reader.GetOutputPort())
xy_plane_Colors.SetLookupTable([you color look up table])
xy_plane_Colors.Update()

# Create an image actor for the XY plane
xy_plane = vtk.vtkImageActor()
xy_plane.GetMapper().SetInputConnection(xy_plane_Colors.GetOutputPort())
xy_plane.SetDisplayExtent(0, dim[0]-1, 0, dim[1]-1, current_zID, current_zID)
# YZ and XZ cut planes can be similarly added!!!
# Current_zID is a user-input integer within the range of [0, zdim-1]
(Pseudo-) Iso-surface Construction: Wireframe Iso-surface
Here’s the situation: we have a 3D grid of data points. At each node, we have an X, Y, Z, and a scalar value S. We know the Transfer Function. We also have a particular scalar value, $S^*$, at which we want to draw the iso-surface(s).
Here’s the situation: we have a 3D grid of data points. At each node, we have an X, Y, Z, and a scalar value S. We know the Transfer Function. We also have a particular scalar value, S*, at which we want to draw the iso-surface(s).

Once you have done Marching Squares for contour lines, doing wireframe iso-surfaces is amazingly easy.

The strategy is that you pick your S*, then draw S* contours on all the parallel XY planes. Then draw S* contours on all the parallel XZ planes. Then draw S* contours on all the parallel YZ planes. And, then you’re done.

What you get looks like it is a connected surface mesh, but in fact it is just independent curves. It is easy to program (once you’ve done Marching Squares at least), and looks good. Also, it’s fast to compute.
Overall Logic for a Wireframe Iso-Surfacing

for( k = 0; k < numZ; k++ )
{
    //Compute Marching Squares on the plane XY_k
}

for( i = 0; i < numX; i++ )
{
    //Compute Marching Squares on the plane YZ_i
}

for( j = 0; j < numY; j++ )
{
    //Compute Marching Squares on the plane XZ_j
}

....

Render all the obtained contours
Iso-surface Construction: Marching Cubes
• For simplicity, we shall work with zero level \((s^*=0)\) iso-surface, and denote

positive vertices as

There are **EIGHT** vertices, each can be positive or negative - so there are \(2^8 = 256\) different cases!
These two are easy!

There is no portion of the iso-surface inside the cube!
Iso-surface Construction - One Positive Vertex - 1

Intersections with edges found by inverse linear interpolation (as in contouring)
Joining edge intersections across faces forms a triangle as part of the iso-surface.
Isosurface Construction - Positive Vertices at Opposite Corners
Iso-surface Construction: Marching Cubes

- One can work through all 256 cases in this way - although it quickly becomes apparent that many cases are similar.
Iso-surface Construction: Marching Cubes

• One can work through all 256 cases in this way - although it quickly becomes apparent that many cases are similar.

• For example:
  – 2 cases where all are positive, or all negative, give no isosurface
  – 16 cases where one vertex has opposite sign from all the rest
Iso-surface Construction: Marching Cubes

• One can work through all 256 cases in this way - although it quickly becomes apparent that many cases are similar.

• For example:
  – 2 cases where all are positive, or all negative, give no isosurface
  – 16 cases where one vertex has opposite sign from all the rest

• In fact, there are only 15 topologically distinct configurations
Case Table
8 Above
0 Below

1 case
7 Above

1 Below

1 case
6 Above
2 Below
3 cases
5 Above
3 Below

3 cases
7 cases

- 4 Above
- 4 Below
4 Above

4 Below

7 cases

4 connected
4 Above

4 Below

7 cases

1 opposite pairs
4 Above

4 Below

7 cases

1 vertex opposite to triplet
4 Above
4 Below
7 cases
1 isolated vertices
Marching Cubes – Look-up Table

• Connecting vertices by triangles
  – **Triangles shouldn’t intersect**
  – To be a closed manifold:
    • Each vertex used by a triangle “fan”
    • Each mesh edge used by 2 triangles (if inside grid cell) or 1 triangle (if on a grid face)
    • Each mesh edge on the grid face is shared between adjacent cells

• Look-up table
  – $2^8=256$ entries
  – For each sign configuration, it stores indices of the grid edges whose vertices make up the triangles

Sign: “0 0 0 1 0 1 0 0”
Triangles: \{\{2,8,11\}, \{4,7,10\}\}
Additional Readings

• Marching Cubes:
  • “Marching cubes: A high resolution 3D surface construction algorithm”, by Lorensen and Cline (1987)
    – >6000 citations on Google Scholar

• Dual Contouring:
  • “Dual contouring of hermite data”, by Ju et al. (2002)
    – >300 citations on Google Scholar
  • “Manifold dual contouring”, by Schaefer et al. (2007)
In VTK

use the `vtkMarchingCubes()` filter and its function `SetValue(0, iso-value)`
Polygonal Iso-Surfaces
- An Alternative

Marching Cubes is difficult, and so we will look at another approach.
Polygonal Iso-Surfaces: Data Structure
Polygonal Iso-Surfaces: Data Structure

bool FoundEdgeIntersection[12]
One entry for each of the 12 edges. 
false means S* did not intersect this edge 
true means S* did intersect this edge
Polygonal Iso-Surfaces: Data Structure

`bool FoundEdgeIntersection[12]`
One entry for each of the 12 edges.
`false` means $S^*$ did not intersect this edge
`true` means $S^*$ did intersect this edge

`Node EdgeIntersection[12]`
If an intersection did occur on edge #i, Node[i] will contain the interpolated $x$, $y$, $z$, $nx$, $ny$, and $nz$.

```c
struct Node{
  float x, y, z, nx, ny, nz;
};
```
bool FoundEdgeIntersection[12]
One entry for each of the 12 edges.
*false* means $S^*$ did not intersect this edge
*true* means $S^*$ did intersect this edge

**Node EdgeIntersection[12]**
If an intersection did occur on edge #i, Node[i] will contain the interpolated x, y, z, nx, ny, and nz.

bool FoundEdgeConnection[12][12]
A *true* in entry [i][j] or [j][i] means that **Marching Squares** has decided there needs to be a line drawn from Cube Edge #i to Cube Edge #j
Both entry [i][j] and [j][i] are filled so that it won’t matter which order you search in later.

```cpp
struct Node{
    float x, y, z, nx, ny, nz;
};
```
bool FoundEdgeIntersection[12]
One entry for each of the 12 edges.
*false* means $S^*$ did not intersect this edge
*true* means $S^*$ did intersect this edge

Node EdgeIntersection[12]
If an intersection did occur on edge #i, Node[i] will contain the interpolated x, y, z, nx, ny, and nz.

bool FoundEdgeConnection[12][12]
A *true* in entry [i][j] or [j][i] means that **Marching Squares** has decided there needs to be a line drawn from Cube Edge #i to Cube Edge #j
Both entry [i][j] and [j][i] are filled so that it won’t matter which order you search in later.

```cpp
struct Node{
    float x, y, z, nx, ny, nz;
};
```
Strategy in ProcessCube():

1. Use ProcessCubeEdge() 12 times to find which cube edges have S* intersections.

2. Return if no intersections were found anywhere.

3. Call ProcessCubeQuad() 6 times to decide which cube edges will need to be connected. This is Marching Squares like we did it before, but it doesn’t need to re-compute intersections on the cube edges in common. ProcessCubeEdge() already did that. This leaves us with the FoundEdgeConnection[][][] array filled.

4. Call DrawCubeTriangles() to create triangles from the connected edges.
Polygonal Iso-Surfaces: Algorithm

Strategy in \texttt{DrawCubeTriangles}():

1. Look through the \texttt{FoundEdgeConnection[][]} array for a Cube Edge \#A and a Cube Edge \#B that have a connection between them.

2. If can’t find one, then you are done with this cube.
Polygonal Iso-Surfaces: Algorithm

Strategy in `DrawCubeTriangles()`:

1. Look through the `FoundEdgeConnection[][]` array for a Cube Edge #A and a Cube Edge #B that have a connection between them.
2. If can’t find one, then you are done with this cube.
3. Now look through the `FoundEdgeConnection[][]` array for a Cube Edge #C that is connected to Cube Edge #B. *If you can’t find one, something is wrong.*
4. Draw a triangle using the `EdgeIntersection[]` nodes from Cube Edges #A, #B, and #C. Be sure to use `glNormal3f()` in addition to `glVertex3f()`.
Strategy in DrawCubeTriangles():

1. Look through the FoundEdgeConnection[][] array for a Cube Edge #A and a Cube Edge #B that have a connection between them.
2. If can’t find one, then you are done with this cube.
3. Now look through the FoundEdgeConnection[][] array for a Cube Edge #C that is connected to Cube Edge #B. If you can’t find one, something is wrong.
4. Draw a triangle using the EdgeIntersection[] nodes from Cube Edges #A, #B, and #C. Be sure to use glNormal3f() in addition to glVertex3f().
5. Turn to false the FoundEdgeConnection[][] entries from Cube Edge #A to Cube Edge #B.
6. Turn to false the FoundEdgeConnection[][] entries from Cube Edge #B to Cube Edge #C.
Polygonal Iso-Surfaces: Algorithm

Strategy in DrawCubeTriangles():

1. Look through the FoundEdgeConnection[]][[] array for a Cube Edge #A and a Cube Edge #B that have a connection between them.
2. If can’t find one, then you are done with this cube.
3. Now look through the FoundEdgeConnection[]][[] array for a Cube Edge #C that is connected to Cube Edge #B. *If you can’t find one, something is wrong.*
4. **Draw a triangle** using the EdgeIntersection[] nodes from Cube Edges #A, #B, and #C. Be sure to use `glNormal3f()` in addition to `glVertex3f()`.
5. Turn to `false` the FoundEdgeConnection[]][[] entries from Cube Edge #A to Cube Edge #B.
6. Turn to `false` the FoundEdgeConnection[]][[] entries from Cube Edge #B to Cube Edge #C.
7. **Toggle** the FoundEdgeConnection[]][[] entries from Cube Edge #C to Cube Edge #A. If this connection was there before, we don’t need it anymore. If it was not there before, then we just invented it and we will need it again.
Polygonal Iso-Surfaces: Why Does this work?

Take this case as an example. The intersection points A, B, C, and D were found and the lines AB, BC, CD, and DA were found because Marching Squares will have been performed on each of the cube’s 6 faces.
Take this case as an example. The intersection points A, B, C, and D were found and the lines AB, BC, CD, and DA were found because Marching Squares will have been performed on each of the cube’s 6 faces.

At this point, we could just draw the quadrilateral ABCD, but this will likely go wrong because it is surely non-planar. So, starting at A, we break out a triangle from the edges AB and BC (which exist) and the edge CA (which doesn’t exist, but we need it anyway to complete the triangle).
Polygonal Iso-Surfaces: Why Does this work?

Take this case as an example. The intersection points A, B, C, and D were found and the lines AB, BC, CD, and DA were found because Marching Squares will have been performed on each of the cube’s 6 faces.

At this point, we could just draw the quadrilateral ABCD, but this will likely go wrong because it is surely non-planar. So, starting at A, we break out a triangle from the edges AB and BC (which exist) and the edge CA (which doesn’t exist, but we need it anyway to complete the triangle).

When we toggle the `FoundEdgeConnection[][][]` entries for AB and BC, they turn from `true` to `false`.
Polygonal Iso-Surfaces: Why Does this work?

Take this case as an example. The intersection points A, B, C, and D were found and the lines AB, BC, CD, and DA were found because Marching Squares will have been performed on each of the cube's 6 faces.

At this point, we could just draw the quadrilateral ABCD, but this will likely go wrong because it is surely non-planar. So, starting at A, we break out a triangle from the edges AB and BC (which exist) and the edge CA (which doesn’t exist, but we need it anyway to complete the triangle).

When we toggle the `FoundEdgeConnection[][][]` entries for AB and BC, they turn from `true` to `false`. When we toggle the `FoundEdgeConnection[][][]` for CA, it turns from `false` to `true`.
Polygonal Iso-Surfaces: Why Does this work?

Take this case as an example. The intersection points A, B, C, and D were found and the lines AB, BC, CD, and DA were found because Marching Squares will have been performed on each of the cube’s 6 faces.

At this point, we could just draw the quadrilateral ABCD, but this will likely go wrong because it is surely non-planar. So, starting at A, we break out a triangle from the edges AB and BC (which exist) and the edge CA (which doesn’t exist, but we need it anyway to complete the triangle).

When we toggle the `FoundEdgeConnection[][]` entries for AB and BC, they turn from *true* to *false*. When we toggle the `FoundEdgeConnection[][]` for CA, it turns from *false* to *true*.

This leaves `FoundEdgeConnection[][]` for CA, CD, and AD all set to *true*, which will cause the algorithm to find them and connect them into a triangle next.
Take this case as an example. The intersection points A, B, C, and D were found and the lines AB, BC, CD, and DA were found because Marching Squares will have been performed on each of the cube’s 6 faces.

At this point, we could just draw the quadrilateral ABCD, but this will likely go wrong because it is surely non-planar. So, starting at A, we break out a triangle from the edges AB and BC (which exist) and the edge CA (which doesn’t exist, but we need it anyway to complete the triangle).

When we toggle the `FoundEdgeConnection[][]` entries for AB and BC, they turn from `true` to `false`. When we toggle the `FoundEdgeConnection[][]` for CA, it turns from `false` to `true`.

This leaves `FoundEdgeConnection[][]` for CA, CD, and AD all set to `true`, which will cause the algorithm to find them and connect them into a triangle next.

Similarly, you can process the intersections at edges E, F, and G, and edge segments EF, FG, and GE to form a triangle.
Polygonal Iso-Surfaces: Why Does this work?

Take this case as an example. The intersection points A, B, C, and D were found and the lines AB, BC, CD, and DA were found because Marching Squares will have been performed on each of the cube’s 6 faces.

At this point, we could just draw the quadrilateral ABCD, but this will likely go wrong because it is surely non-planar. So, starting at A, we break out a triangle from the edges AB and BC (which exist) and the edge CA (which doesn’t exist, but we need it anyway to complete the triangle).

When we toggle the `FoundEdgeConnection[][]` entries for AB and BC, they turn from `true` to `false`. When we toggle the `FoundEdgeConnection[][]` for CA, it turns from `false` to `true`.

This leaves `FoundEdgeConnection[][]` for CA, CD, and AD all set to `true`, which will cause the algorithm to find them and connect them into a triangle next.

Similarly, you can process the intersections at edges E, F, and G, and edge segments EF, FG, and GE to form a triangle.
Polygonal Iso-Surfaces: Surface Normals

We would very much like to use lighting when displaying polygonal iso-surfaces, but we need surface normals at all the triangle vertices. Because there really isn’t a surface there, this would seem difficult, but it’s not.

The normal is a vector that shows how the iso-surface is changing. How “something is changing” is called the gradient. So, the surface normal to a volume is:

\[
\vec{n} = \left( \frac{dS}{dx}, \frac{dS}{dy}, \frac{dS}{dz} \right) = \nabla S
\]

\[
\frac{dS}{dx} = \frac{S_{i+1,j,k} - S_{i-1,j,k}}{2\Delta x}
\]
Polygonal Iso-Surfaces: Surface Normals

We would very much like to use lighting when displaying polygonal iso-surfaces, but we need surface normals at all the triangle vertices. Because there really isn’t a surface there, this would seem difficult, but it’s not.

The normal is a vector that shows how the iso-surface is changing. How “something is changing” is called the gradient. So, the surface normal to a volume is:

\[
\vec{n} = \left( \frac{dS}{dx}, \frac{dS}{dy}, \frac{dS}{dz} \right) = \nabla S
\]

Prior to the iso-surface calculation, you compute the surface normals for all the nodes in the 3D mesh. You then interpolate them along the cube edges when you create the iso-surface triangle vertices.
Acknowledgment

• Thanks for materials from
  – Dr. Mike Bailey
Open discussion: What can be the issue(s) of Marching Cubes or the aforementioned polygonal iso-surfacing approach?