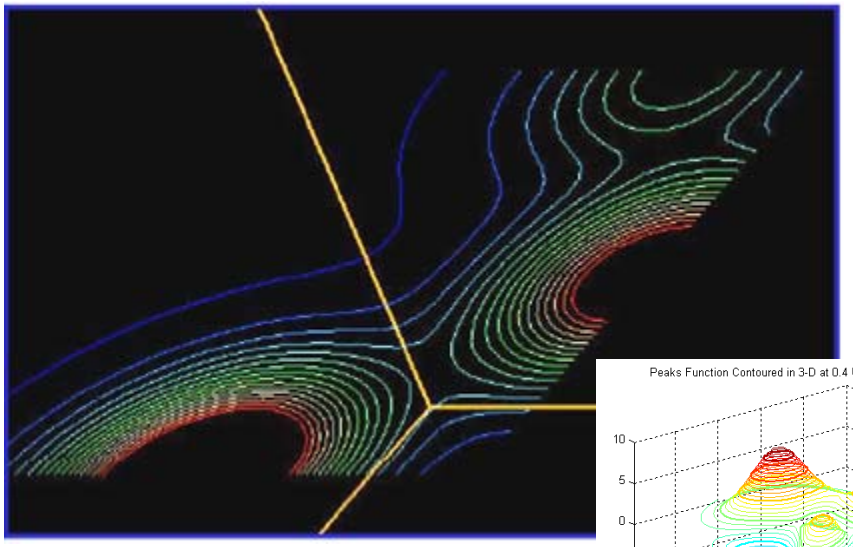
Geometric-Based Scalar Field Visualization

Iso-Contouring

Goal: know the iso-contours are defined and extracted; know the Marching Squares algorithm

2D Contour Lines

- ***Contour (iso-value) line(s)***
  - **Sub-sets of the original data that correlate all the points with the same scalar values**.
  - If the 2D scalar field is considered as a height field (2D surface), the contours are the intersections of a moving horizontal plane with this height field.



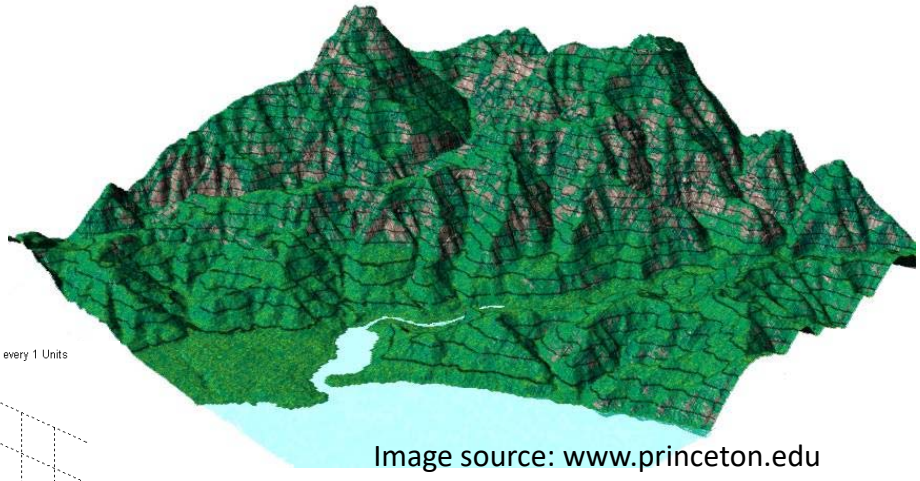


Image source: www.princeton.edu

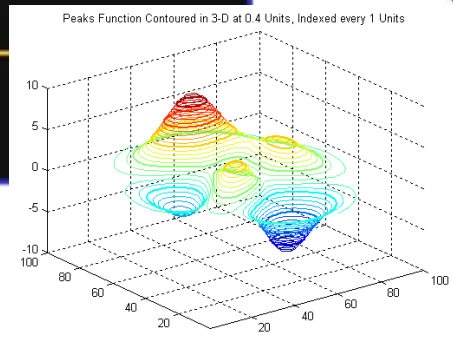
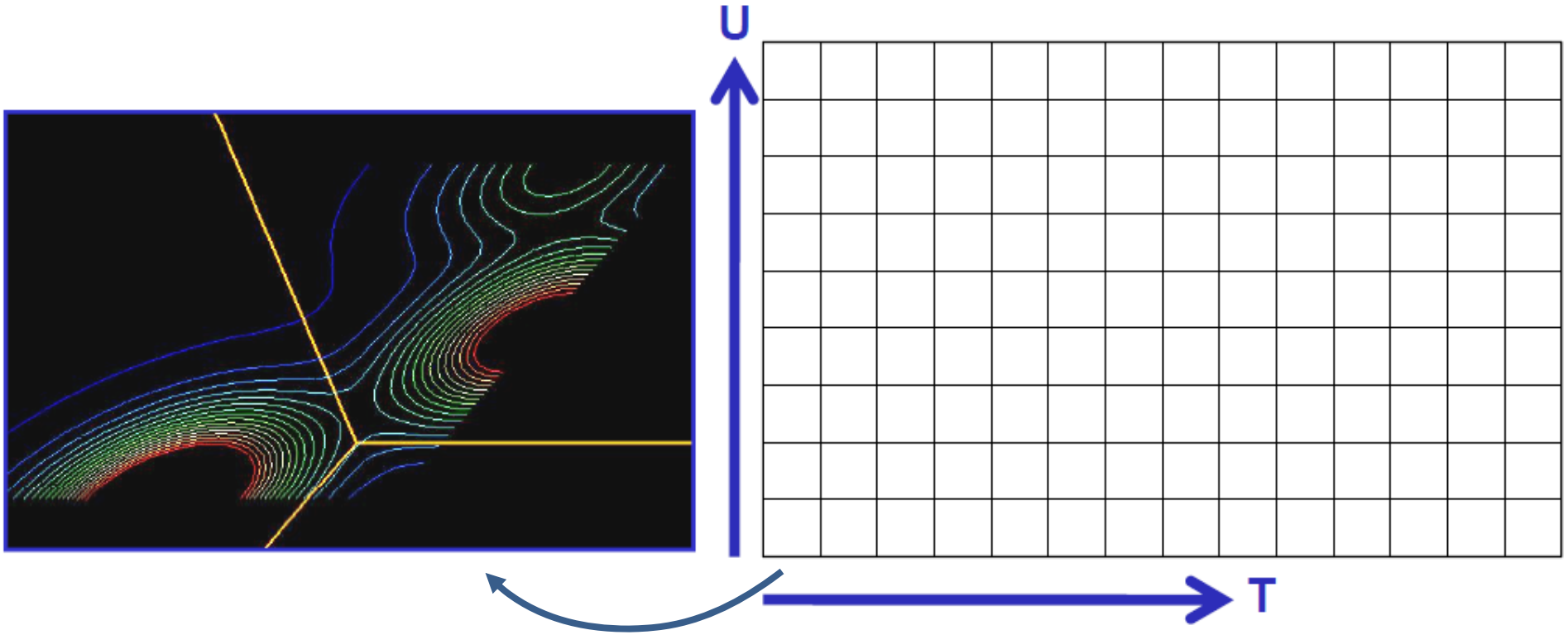


Image source: www.mathworks.com

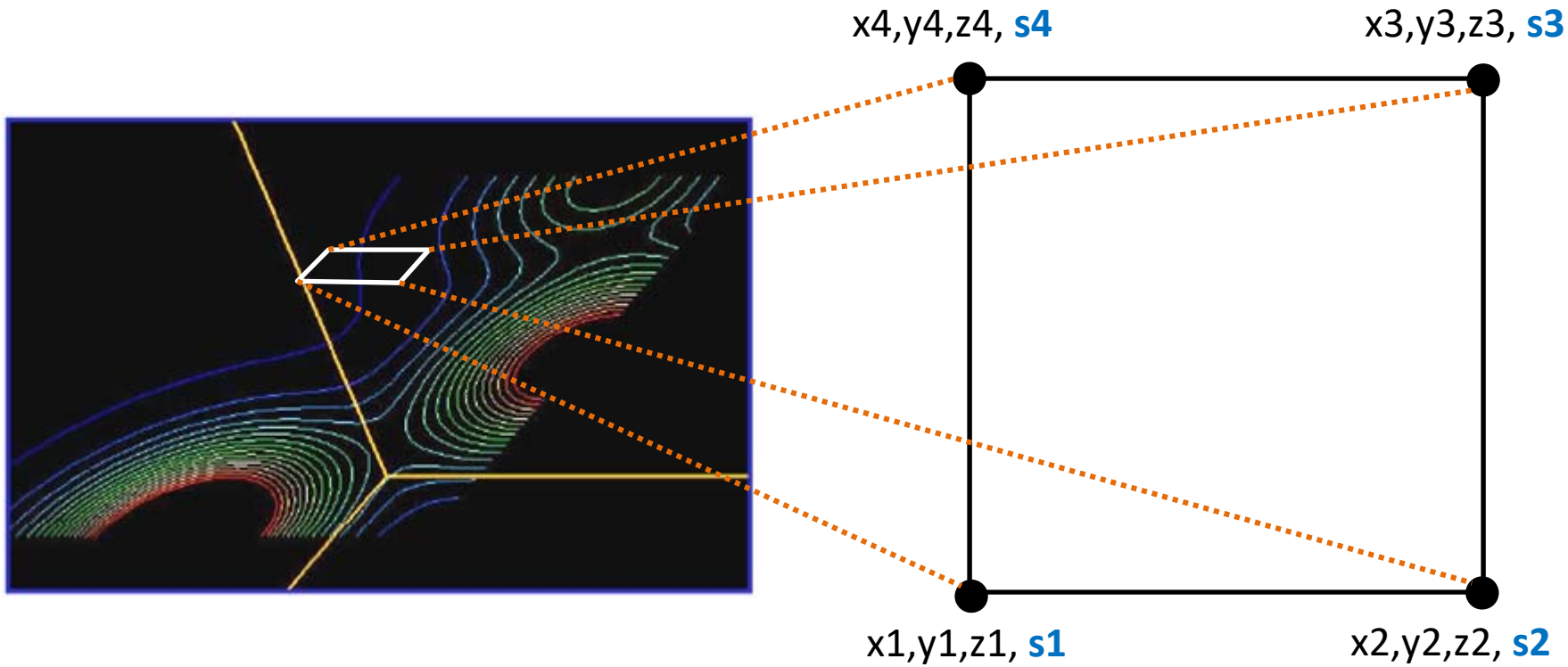
2D Contour Lines

- **Here's the situation**: we have a 2D 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 **contour (iso-value) line(s).***



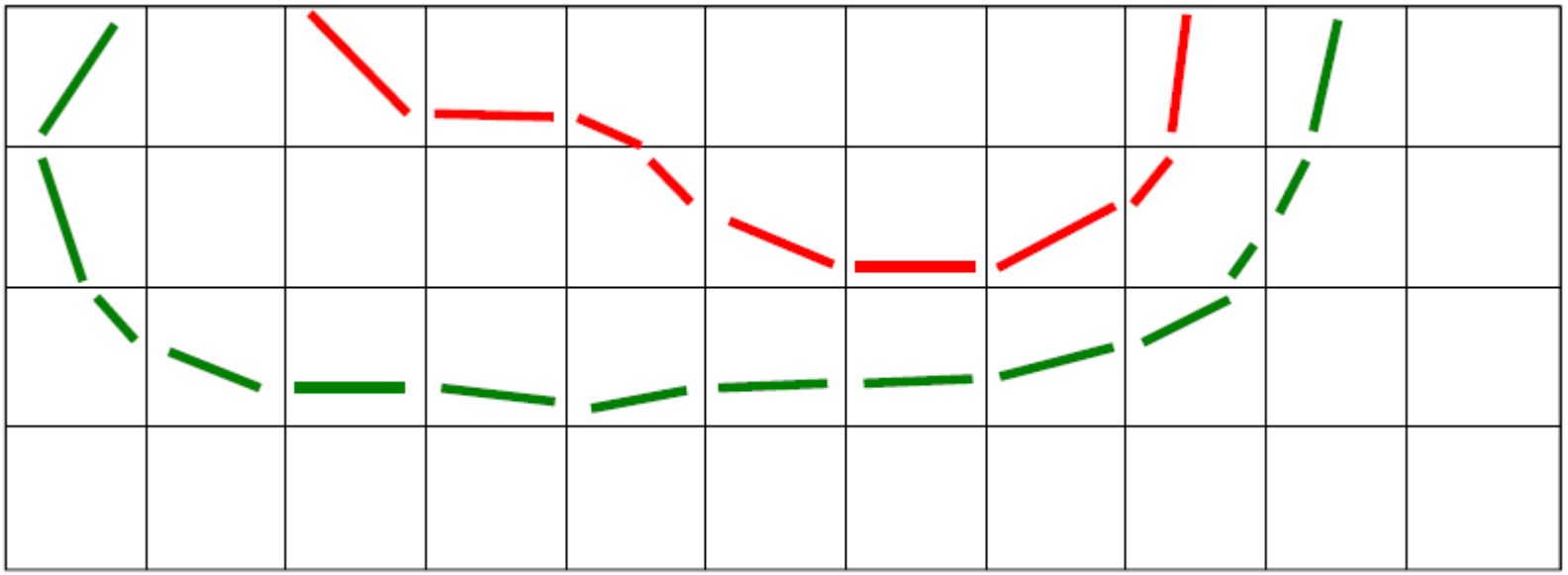
2D Contour Lines: Marching Squares

- Instead of dealing with the entire grid, we once again look at one square at a time, then march through them all in order. For this reason, this method is called the ***Marching Squares.***



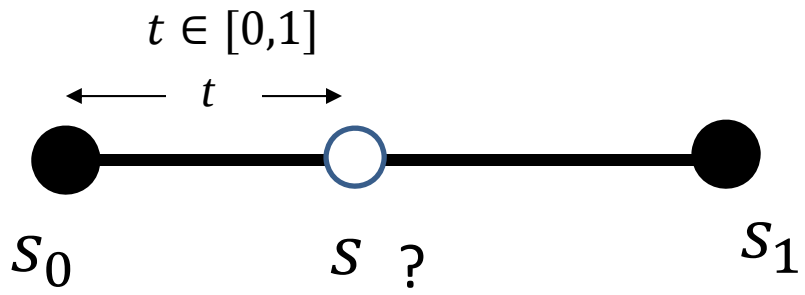
Marching Squares

- What's really going to happen is that we are not creating contours by connecting points into a complete curve all at once. Instead, **we are creating contours by drawing a collection of 2-point line segments**, assuming that those line segments will align across square boundaries.



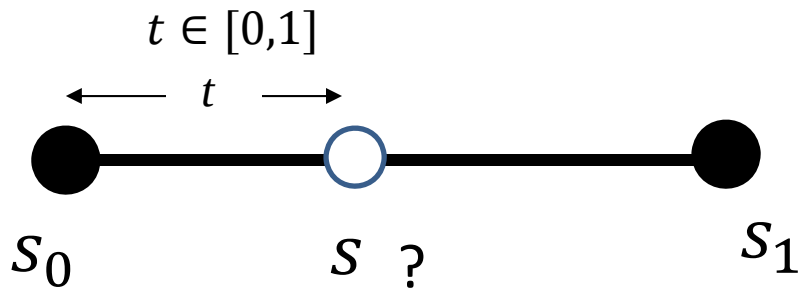
We need to check the relation between the contour and an edge of a square/quad

Linear Interpolation on an Edge



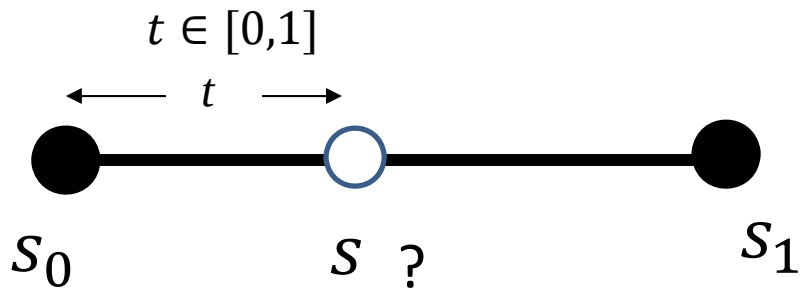
s0, *s1*, and *s*, are scalar values defined on an edge

Linear Interpolation on an Edge



s has to be between s_0 and s_1 **because of the linear interpolation**.

Linear Interpolation on an Edge



s has to be between s_0 and s_1 **because of the linear interpolation**.

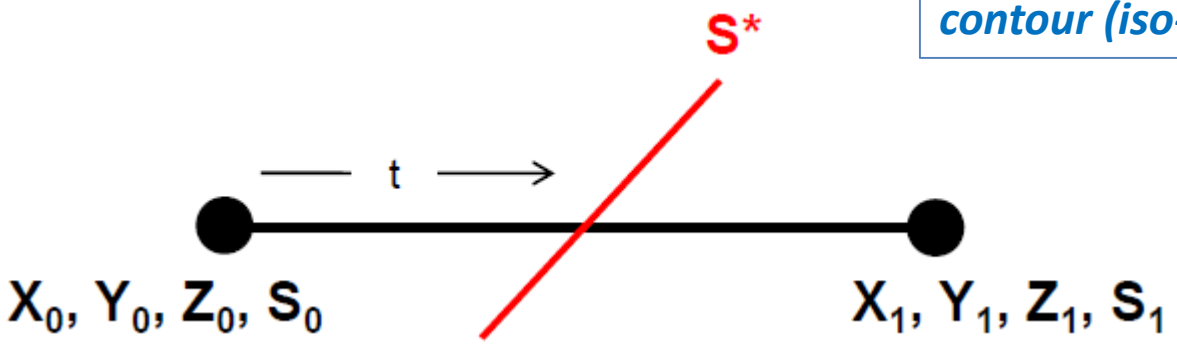
$$S = (1-t)S_0 + tS_1 = S_0 + t(S_1 - S_0) \quad \textit{where } 0. \leq t \leq 1.$$

Recall: do you know how to linearly interpolate two colors?

Marching Squares

Does S^* cross any edges of this square?

*We have a particular scalar value, **S***, at which we want to draw the **contour (iso-value) line(s).***



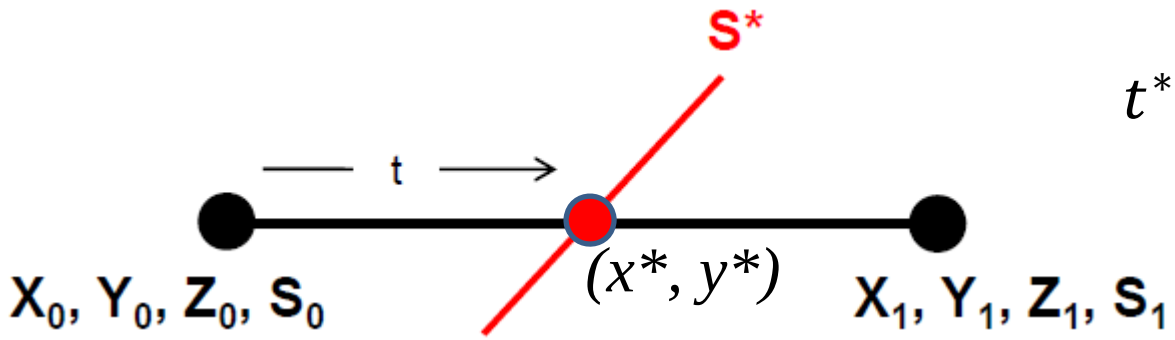
Linearly interpolating any scalar value from node0 to node1 gives:

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

Substituting this interpolated S with S* and solving for t* gives:

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

Marching Squares

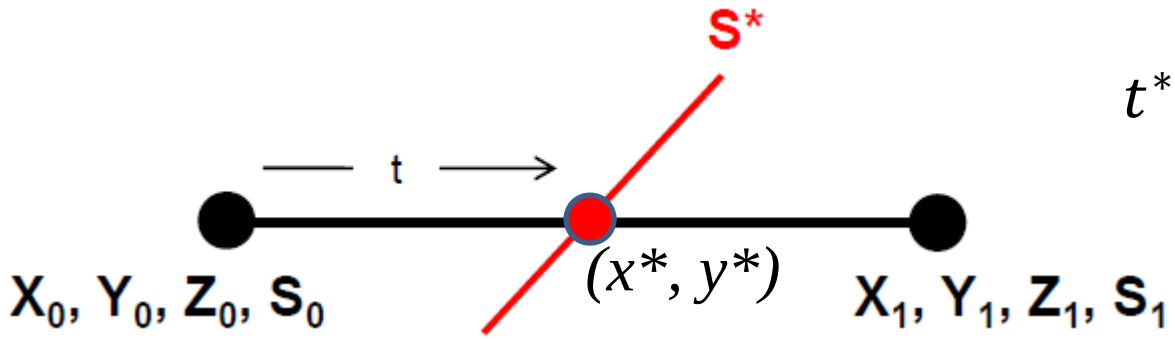


$$t^* = \frac{S^* - S_0}{S_1 - S_0}$$

If 0. ≤ t* ≤ 1., then S* crosses this edge.

Any better way to determine whether a contour intersects with an edge?

Marching Squares



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

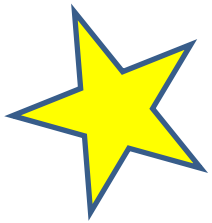
If 0. ≤ t* ≤ 1., then S* crosses this edge. You can compute where S* crosses the edge by using the same linear interpolation equation you used to compute S*. You will need intersection location for the later visualization.

The coordinates of the intersection

$$x^* = (1 - t^*)x_0 + t^* x_1$$
$$y^* = (1 - t^*)y_0 + t^* y_1$$

Marching Squares

- **Do the above intersection computation for all 4 edges** – when you are done, there are 5 possible ways this could have turned out
  - # of intersections = 0
  - # of intersections = 2
  - # of intersections = 1
  - # of intersections = 3



Marching Squares

- **Do the above intersection computation for all 4 edges** – when you are done, there are 5 possible ways this could have turned out

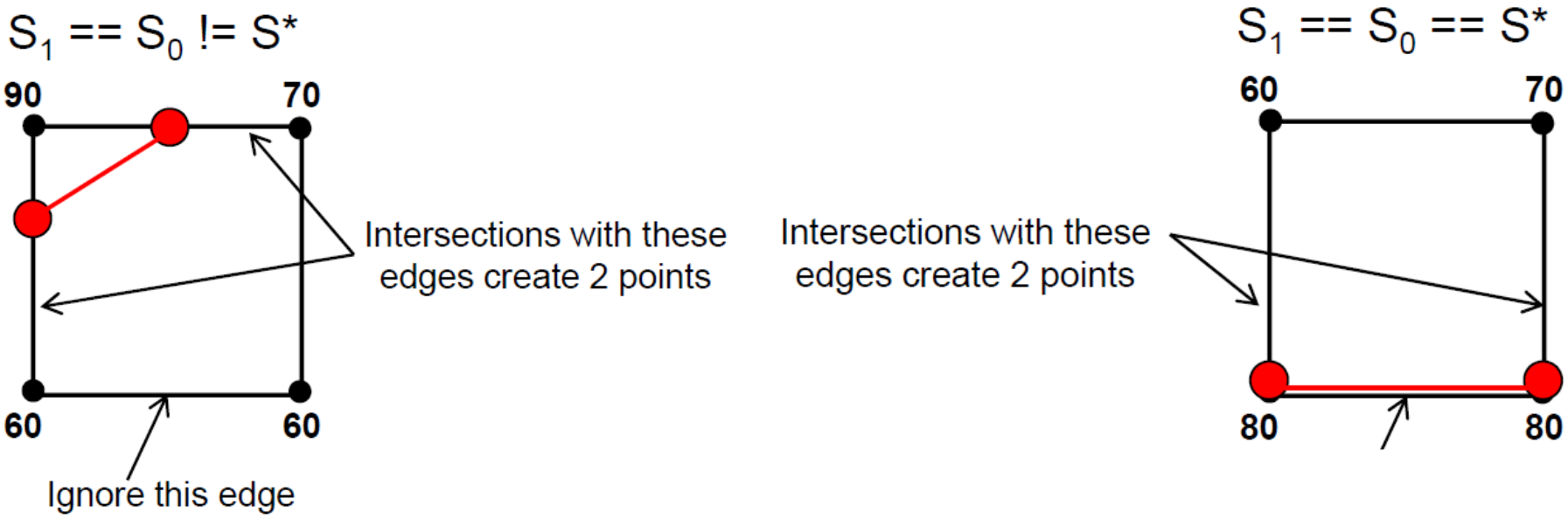
– # of intersections = 0	Do nothing
– # of intersections = 2	Draw a line connecting them
– # of intersections = 1	Error: this means that the contour got into the square and never got out
– # of intersections = 3	If one intersection is not on a vertex ->Error: this means that the contour got into the square and never got out

Marching Squares

Special cases

What if $S_1 == S_0$ (i.e., $t^* = \infty$) because $t^* = \dfrac{S^* - S_0}{S_1 - S_0}$

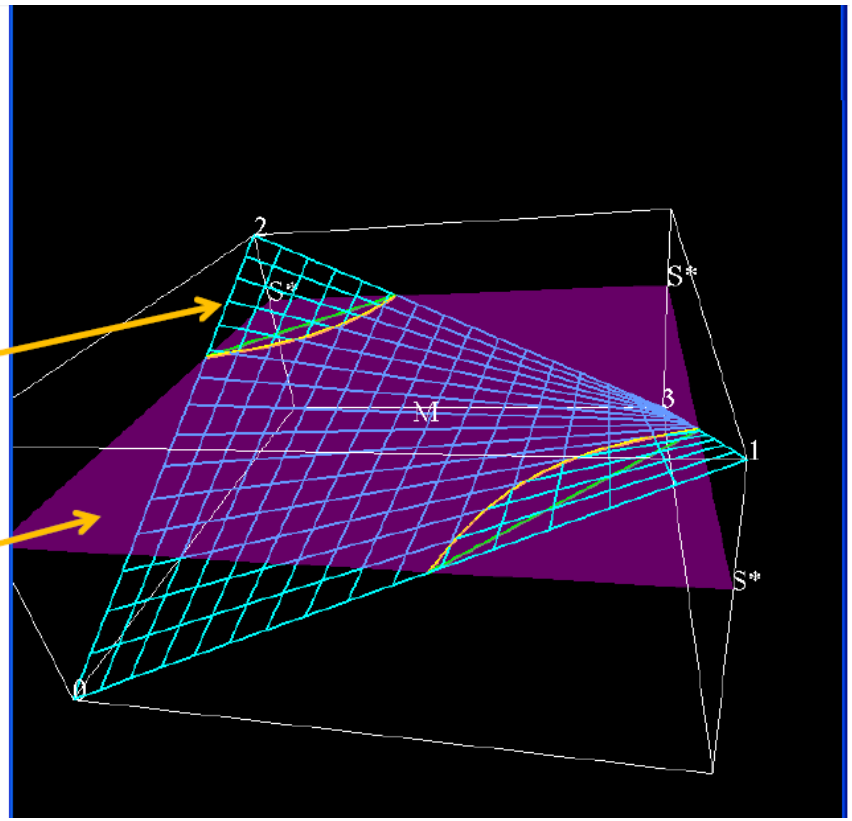
There are two possibilities.



Marching Squares

Special cases

What if there are **four intersections**

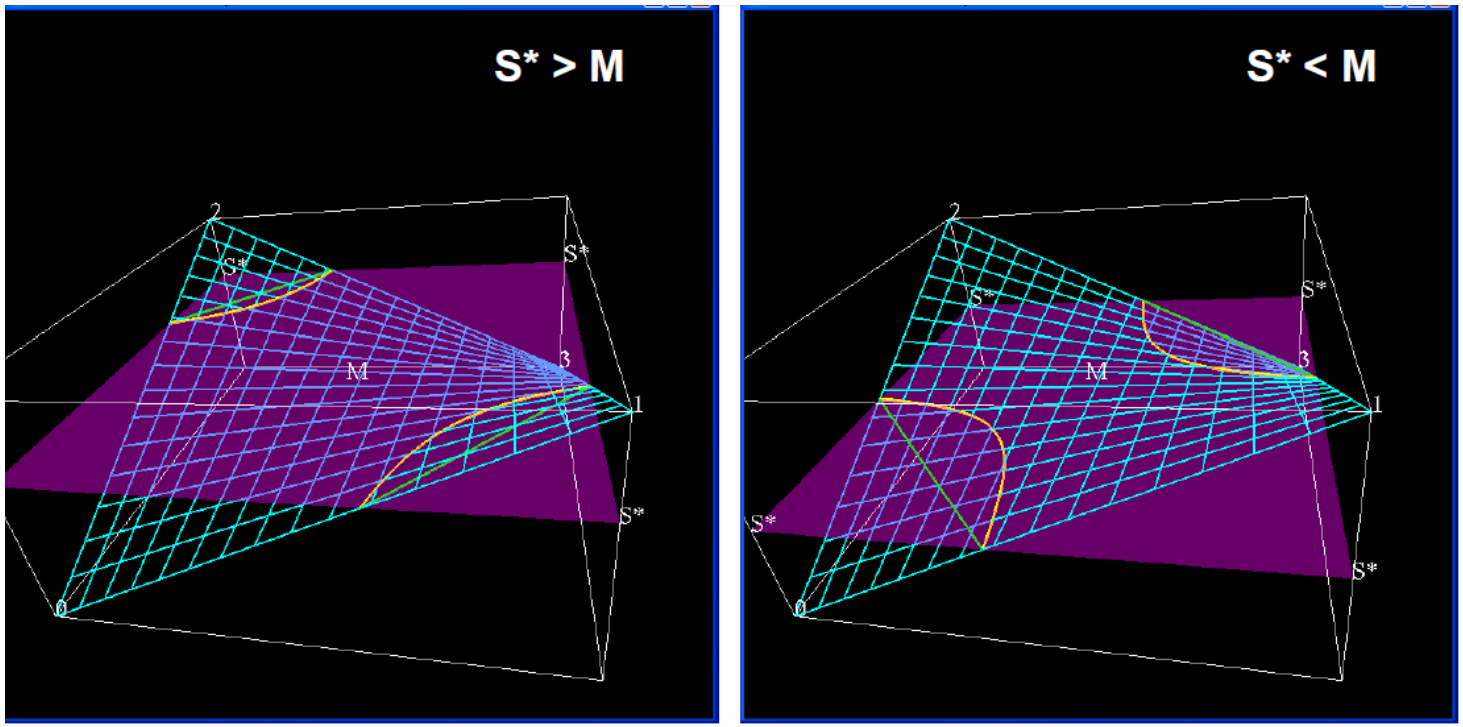


This means that going around the square, the nodes are >S*, <S*, >S*, and <S* in that order. This gives us a **saddle function**, shown here in cyan. The plane in magenta represents the plane with scalar value S*.

The intersection of this plane with the saddle function gives rise to the contours (in **orange**).

Marching Squares

The 4-intersection case



The exact contour curve is shown in orange. The Marching Squares contour line is shown in green.

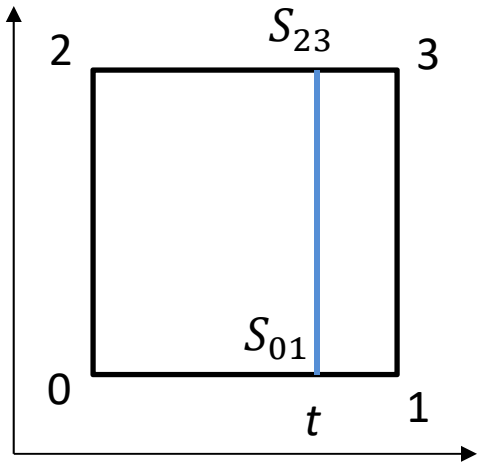
Notice what happens as we lower S* -- there is a change in which sides of the square get connected. That change happens when S* > M becomes S* < M (where M is the middle scalar value).

Marching Squares

The 4-intersection case: Compute the middle scalar value

Let's linearly interpolate scalar values along the 0-1 edge, and along the 2-3 edge:

$$S_{01} = (1-t)S_0 + tS_1$$
$$S_{23} = (1-t)S_2 + tS_3$$



Marching Squares

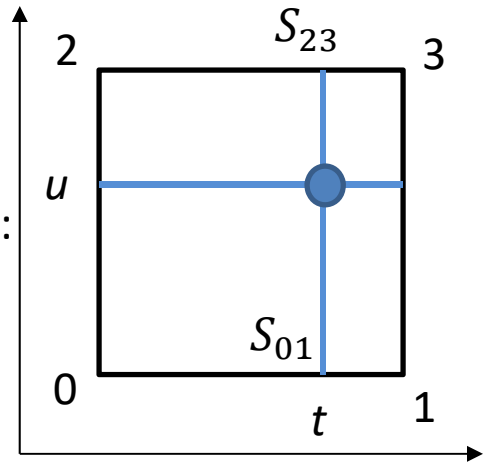
The 4-intersection case: Compute the middle scalar value

Let's linearly interpolate scalar values along the 0-1 edge, and along the 2-3 edge:

$$S_{01} = (1-t)S_0 + tS_1$$
$$S_{23} = (1-t)S_2 + tS_3$$

Now linearly interpolate these two linearly-interpolated scalar values:

$$S(t,u) = (1-u)S_{01} + uS_{23}$$

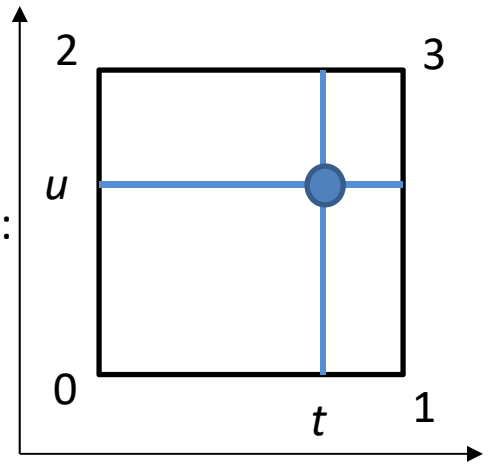


Marching Squares

The 4-intersection case: Compute the middle scalar value

Let's linearly interpolate scalar values along the 0-1 edge, and along the 2-3 edge:

$$S_{01} = (1-t)S_0 + tS_1$$
$$S_{23} = (1-t)S_2 + tS_3$$



Now linearly interpolate these two linearly-interpolated scalar values:

$$S(t,u) = (1-u)S_{01} + uS_{23}$$

Expand this we get

$$S(t,u) = (1-t)(1-u)S_0 + t(1-u)S_1 + (1-t)uS_2 + tuS_3$$

This is the bilinear interpolation equation.

Marching Squares

The 4-intersection case: Compute the middle scalar value

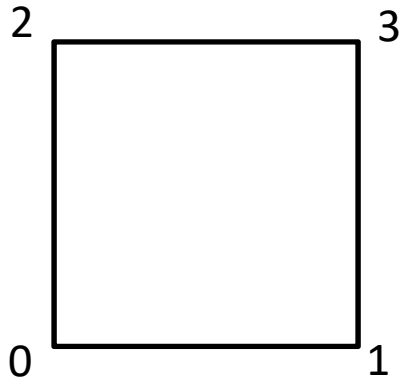
The middle scalar value, M, is what you get when you set t = .5 and u = .5:

$$S(t,u) = (1-t)(1-u)S_0 + t(1-u)S_1 + (1-t)uS_2 + tuS_3$$

$$M = S(.5,.5) = \frac{1}{4}S_0 + \frac{1}{4}S_1 + \frac{1}{4}S_2 + \frac{1}{4}S_3 = \frac{S_0 + S_1 + S_2 + S_3}{4}$$

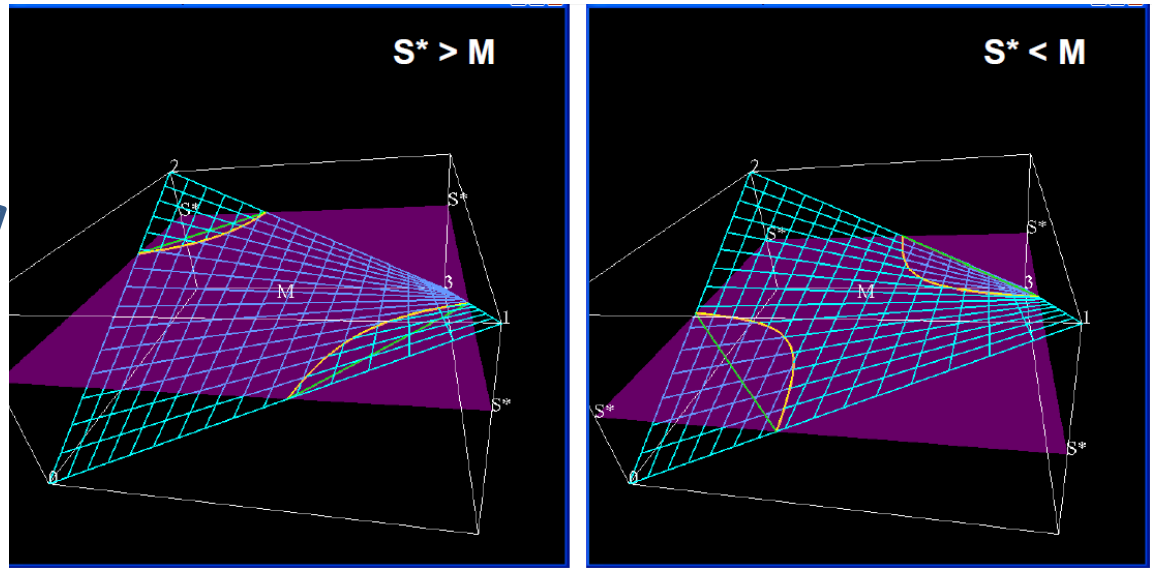
Thus, M is the simple average of the four corner scalar values.

Marching Squares



The logic for the 4-intersection case is as follows:

1. Compute M
2 If S0 is on the same side of M as S* is, then connect the 0-1 and 0-2 intersections, and the 1-3 and 2-3 intersections
3. Otherwise, connect the 0-1 and 1-3 intersections, and the 0-2 and 2-3 intersections



Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

```
intersection_counter = 0;
for each edge of a square Si {
    //Determine whether there is an intersection given s*
    if (s0 < s* < s1 || s1<s*<s0)
    {  // compute the intersection
       intersection_counter ++;
    }
}
//According to the value of intersection_counter, connect the intersections
```

Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

```
for all squares{
    intersection_counter = 0;
    for each edge of a square Si {
        //Determine whether there is an intersection given s*
        if (s0 < s* < s1 || s1<s*<s0)
        {  // compute the intersection
           intersection_counter ++;
        }
    }
    //According to the value of intersection_counter, connect the intersections

}
```

Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

```
for all squares{
    intersection_counter = 0;
    for each edge of a square Si {
        //Determine whether there is an intersection given s*
        if (s0 < s* < s1 || s1<s*<s0)
        {  // compute the intersection
           intersection_counter ++;
        }
    }
    //According to the value of intersection_counter, connect the intersections
    Handling special cases, e.g. s0==s1
}
```

Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

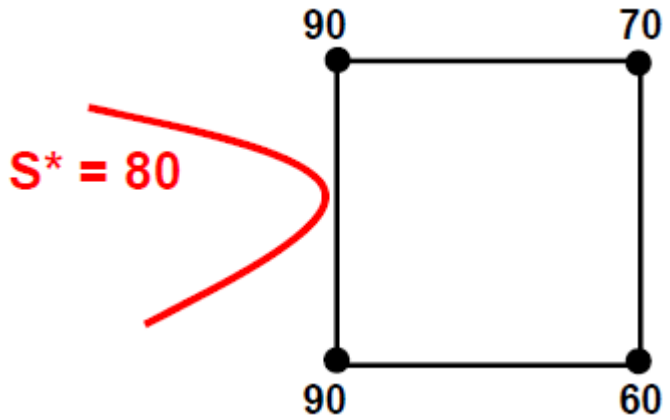
```
for all squares{
    intersection_counter = 0;
    for each edge of a square Si {
        //Determine whether there is an intersection given s*
        if (s0 < s* < s1 || s1<s*<s0)
        {  // compute the intersection
           intersection_counter ++;
        }
    }
    //According to the value of intersection_counter, connect the intersections
    Handling special cases, e.g. s0==s1
}
```

What is the issue with the above code and how to optimize?

Artifacts?

What if the distribution of scalar values along the square edges isn't linear?

What if you have a contour that really looks like this?



Artifacts?

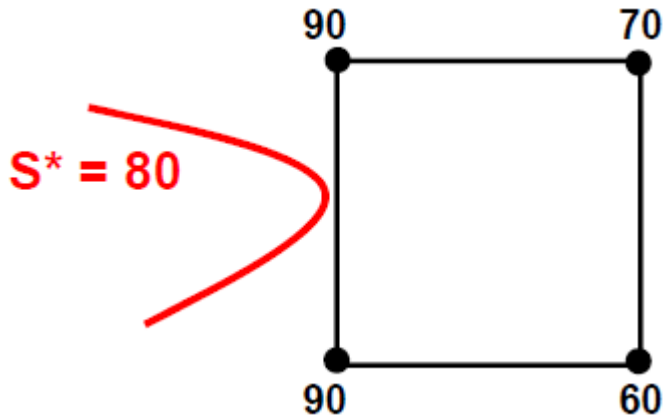
What if the distribution of scalar values along the square edges isn't linear?

We have no basis to assume *anything*. So linear is as good as any other guess, and lets us consider just one square by itself. Some people like looking at adjacent nodes and using quadratic or cubic interpolation on the edge. This is harder to deal with computationally, and is also making an assumption for which there is no evidence.

What if you have a contour that really looks like this?

You'll never know. We can only deal with the data that we've been given.

There is no substitute for having an adequate number of Points data

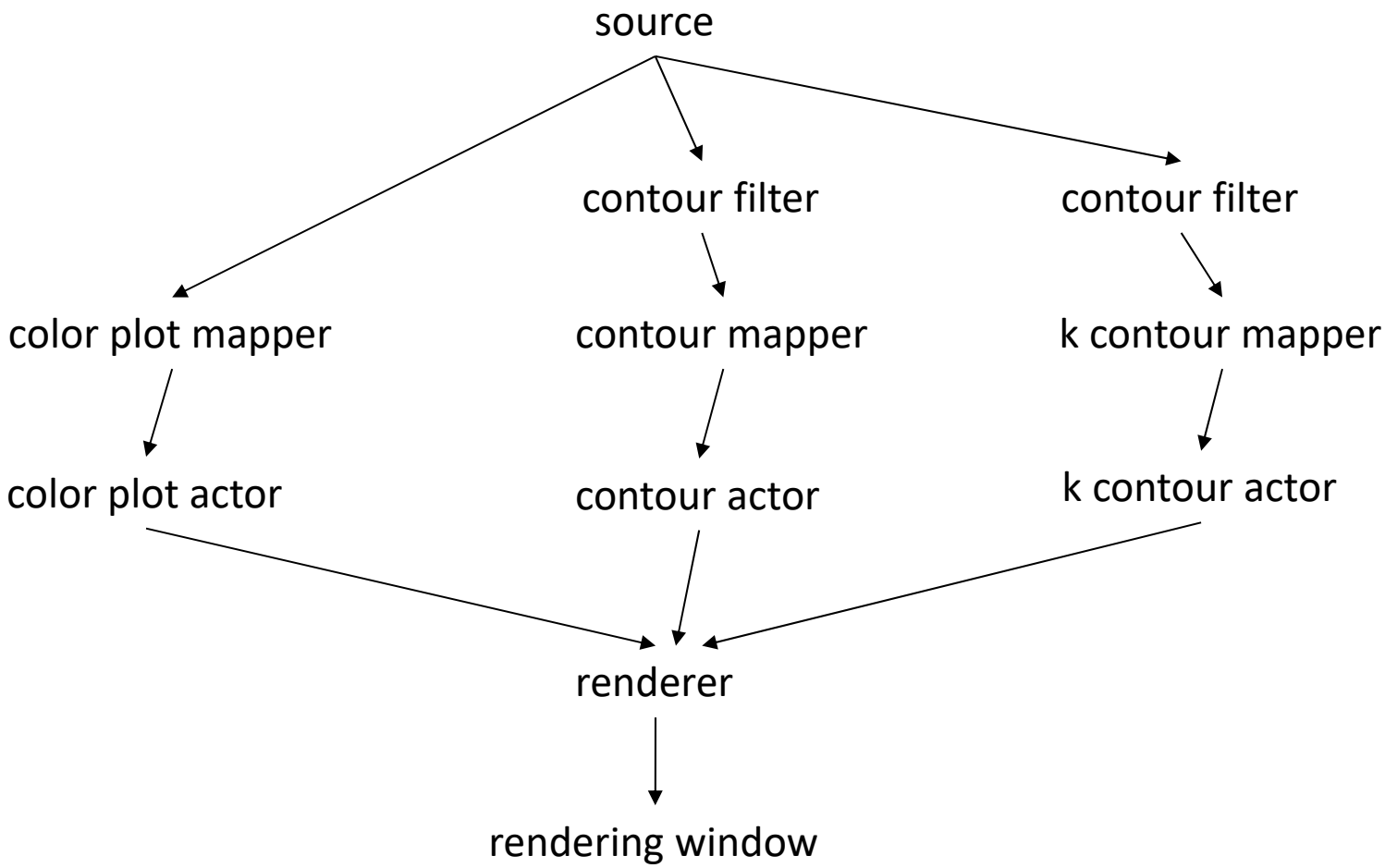


In VTK

For a **single** iso-contour, use `vtkContourFilter` filter and its `SetValue(0, [scalar value of the contour])`

For a **multiple** iso-contours, use the `vtkContourFilter` and its member function `GenerateValues([number of contours], [scalar data range])`.

In VTK



For **Assignment 1**