Assignment #3:
2D Scalar Field Visualization II: Iso-contouring

Due Sept. 20, before midnight

Goals:

This is the continuation of the 2D scalar field visualization project. So, **please continue working on the program you got from the last assignment**!

The goal of this assignment is to implement the iso-contouring technique (i.e., Marching Squares) introduced in the class. You will also be asked to design a proper user interface to aid the data exploration.

Tasks:

1. Writing questions (10 points)

Describe an algorithm to compute the 2D iso-contours on a given triangle mesh. You can implement it in the extra (optional) task 4.

2. Visualize the data using color plots (30 points)

The following describes the tasks for performing the visualization of a scalar field defined on a quadrilateral mesh. Note that different from the triangle meshes you have worked with in the preceding assignment, the 2D elements of this mesh are individual quads (i.e., with 4 vertices and 4 edges). However, the rendering pipeline is the same as the previous assignment.

2.1 Load a 2D scalar field from the given "temperature1.dat" file (**20 points**).

The first line of file shows the numbers of vertices along X and Y direction, respectively. The following lines store the x, y, z coordinates and the scalar data values of the individual vertices. The vertices are stored in the row dominant fashion (i.e., from left to right, then bottom to top). See the first few lines of the data file below.

```
50 50
-1.000000, -1.000000, 0.000000, 70.599884
-0.959184, -1.000000, 0.000000, 71.457848
-0.918367, -1.000000, 0.000000, 71.131317
-0.877551, -1.000000, 0.000000, 69.636490
-0.836735, -1.000000, 0.000000, 67.046814
-0.795918, -1.000000, 0.000000, 63.487019
-0.755102, -1.000000, 0.000000, 59.123165
-0.714286, -1.000000, 0.000000, 54.149830
-0.673469, -1.000000, 0.000000, 48.775784
-0.632653, -1.000000, 0.000000, 43.209663
-0.591837, -1.000000, 0.000000, 37.646980
```

Write a file loader to load this data. To ease your loading process, I recommend to **create a list of vertices** with the following data structure for each node.

In C++ Version

```
typedef struct node{
   float x,  y, z;   // To store the coordinate of the node
   float s;   // To store the scalar value
};
```

In JS Version

```
var node = {
   x: [],
   y: [],
   z: [],   // To store the coordinate of the node
   s: []   // To store the scalar value
}
```

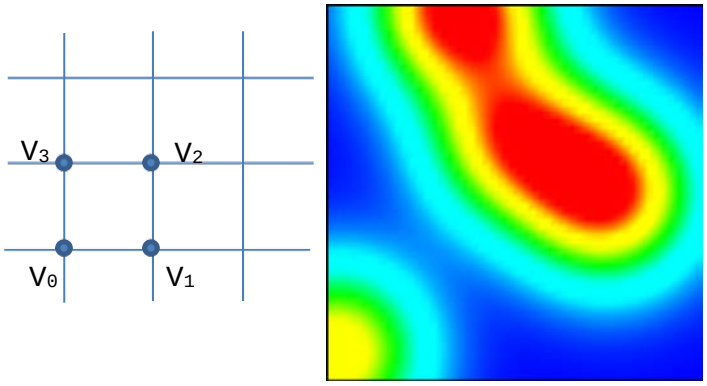
Remember to store the first two integers of the file into the following variables, which you will use later to identify the pair of vertices that form an edge and the four vertices that form a square!!

```
int NX, NY;
```

Alternatively (and optionally), you can also use the vertex list provided in the skeleton code to store the loaded information to make your visualization system consistent through different file formats.

2.2 Visualize the data as a color plot (**10 points**), using

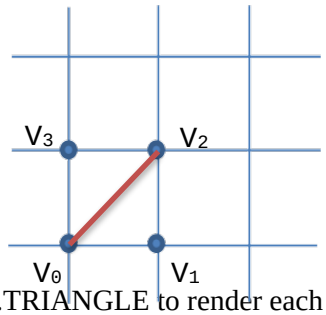
```
glBegin(GL_QUADS); // or GL_POLYGON
      glColor3f(r0, g0,
      b0);
      glVertex3f(x0,y0,z0);
      glColor3f(r1, g1,
      b1);
      glVertex3f(x1,y1,z1);
      glColor3f(r2, g2,
      b2);
      glVertex3f(x2,y2,z2);
      glColor3f(r3, g3,
      b3);
      glVertex3f(x3,y3,z3);
glEnd();
```



You need to be careful about the ordering of the grid points in order to form a valid quad (see the right illustration)!

For webGL (JS skeleton code):

In webGL there is no gl.QUAD available, in fact you need to divide each quad into two triangles as shown below,



Now you can use gl.TRIANGLE to render each quad using two triangles, to do this you need to arrange the vertices in the following way,

indices.push(i_3, i_2, i_0);

indices.push(i_2, i_1, i_0);

Where i_0, i_1, i_2 and i_3 are indices of vertex V_0, V_1, V_2 and V_3 respectively.

Alternately, you can also divide the quad along the other diagonal.

You should apply the color coding schemes you have implemented in Assignment 2 for this task.

3. Extract iso-contours corresponding to the user specified scalar values. (60 points)

3.1 Implement a Marching Square algorithm as described in the class

For simplicity, you can go over each square and **compute and render** the line segment of the iso-contour if it ever passes through the square. There is no need to store the intersections and iso-contour.

If you use the provided (in the end of this description) functions to build the edge and face lists (consisting of the individual squares), this marching square should be quite straight forward to implement. If you are not building the edge and face lists using the provided routine, you will need to figure out each pair of vertices that forms an edge and the four vertices that form a square during the Marching Square computation.

Hint: there are two cases that an edge connects two vertices based on their indices i, j (i < j)

(1) If i and j are in the same row (i.e., i/NX == j/NX) and j = i+1

(2) If i and j are in two neighboring rows and j = i+NX;

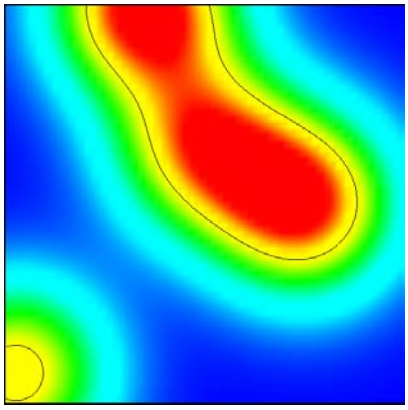
Hint: The following four vertices (labeled by their indices) will form a quad: i, i+1, i+NX+1, i+NX (and i < NX-1)

3.2 Add a user interface to allow the user to perform the following operations:

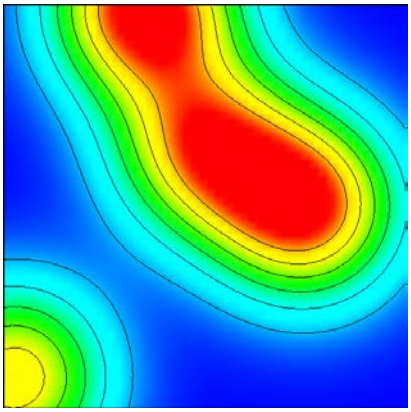
(1) Select a scalar value from which the iso-contour is computed

(2) Specify an integer K so that K iso-contours can be shown AT THE SAME TIME. You can either randomly generate K values or uniformly select K values from the value range of the data [S_{min}, S_{max}] (see below).

```
S_i = Smin + i*(Smax- Smin)/(k-1);  // compute K iso-values by uniformly sampling the value range
```



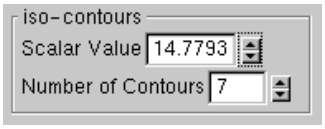
Single iso-value 76.141



K = 6

If you use GLUI as your interface, add two spinners (see **page 29** of the GLUI library) to your interface. One is for the user to specify the scalar value that he/she wants to extract the contour. The other is to specify the total number, say "k", of contours that will be computed.

Spinners for iso-contouring using GLUI interface



If you use AntTweakBar as your interface, you need to add two modifiable variables for the specification of the scalar value and number of iso-contours. See the following example (already in your skeleton code)

```
// Add 'g_Zoom' to 'bar': this is a modifiable (RW) variable of type TW_TYPE_FLOAT. Its key shortcuts are [z] and [Z].
TwAddVarRW(bar, "Zoom", TW_TYPE_FLOAT, &g_Zoom,
" min=0.01 max=2.5 step=0.01 keyIncr=z keyDecr=Z help='Scale the object (1=original size).' ");
```

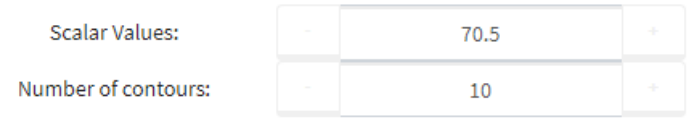
If you use Web as your interface, you can use the bootstrap input spinners. In the html file, add an input tag as follows:

```
<input id="scalar_input" type="number" value="50" min="0" max="100" step="0.5" data-decimals="1" />
```

and use this code for Javascript:

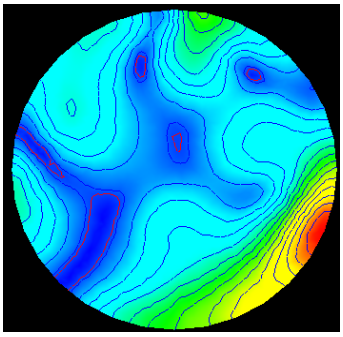
```
"$("#scalar_input").inputSpinner();
$("#scalar_input").on("change", function(event){
  // You can get the input data with this function call $("#scalar_input").val()
});
```

You should see the final interface like this:



For more information, please see the readme file.

4. Iso-contouring on triangular mesh (Extra/Optional 10 points)



Implement the triangle-mesh version of the iso-contouring algorithm you describe in Question 1. Apply your algorithm to the triangle meshes you are given from Assignment 2 (**10 points**).

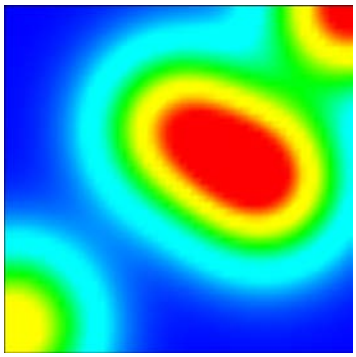
PLEASE submit your source code and a report describing how you design and implement your algorithm (including the data structures used to store the intersections and line segments and the functions involved in the algorithm) in a .zip file to the blackboard.

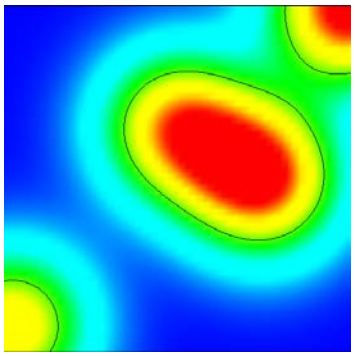
Grades:

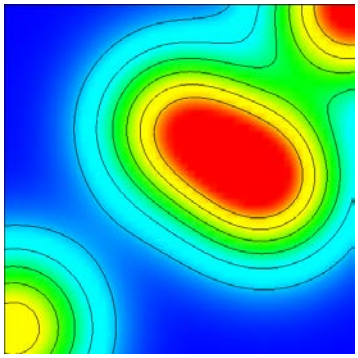
Tasks	*Total points*
1	10
2	30
3	60
4(extra)	10

For your reference:

This is how the "temperature2.dat" should look like


Continuous color plot


Single iso-contour (56.3766)


K= 6

Hints: A reference code for file reading and data storing for the data stored on a uniform grid. You are free to implement your own version of the file loading and data storage.

For C++ version:

```
///Data structure to store the data defined on the uniform grid
typedef struct node
{
	float x, y, z, s;
};

typedef struct lineseg
{
	int n1, n2;
};

typedef struct quad
{
	int verts[4];
	int edges[4];
};

int NX, NY;

std::vector<node> grid_pts;
std::vector<lineseg> edgeList;
std::vector<quad> quadList;


////Load the data stored in uniform grid
void Load_data_on_uniformGrids(const char *name)
{
	int i;
	FILE *fp = fopen(name, "r");
	if (fp == NULL) return;

	fscanf(fp, "%d %d\n", &NX, &NY);

	grid_pts.clear();

	for (i = 0; i < NX*NY; i++)
	{
		node tmp;
		fscanf(fp, "%f, %f, %f, %f \n", &tmp.x, &tmp.y, &tmp.z, &tmp.s);
		grid_pts.push_back(tmp);
	}
	fclose(fp);
}
```

The following two functions build the edge list and face list of the uniform grid, which may be useful if you decide to first compute the contour intersections before connecting them to form the

complete contour. If you decide to use them, please finish the following highlighted parts of the code.

```
////Build the edge list for the loaded uniform grid
void build_edge_list()
{
	int i, j;
	int cur = 0;
	edgeList.clear();
	for (j = 0; j < NY - 1; j++)
	{
		cur = j * NX;
		for (i = 0; i < NX - 1; i++)
		{
		    //<TODO:> Build a horizontal edge between vertex i and i+1

		    //<TODO:> Build a vertical edge between vertex i and i+NX


			cur++;
		}
		    //<TODO:> Build a rightmost edge between vertex cur and cur+NX
	}

	// Build the edges on the top boundary
	cur = (NY - 1) * NX;
	for (i = 0; i < NX - 1; i++)
	{
		//<TODO:> Build a horizontal edge between vertex i and i+1
		cur++;
	}
}

////Build the quad face list for the loaded uniform grid
void build_face_list()
{
	int i, j;
	int cur = 0;
	quadList.clear();
	for (j = 0; j < NY - 1; j++)
	{
		cur = j * NX;
		for (i = 0; i < NX - 1; i++)
		{
			//<TODO:> find the four vertices (their indices) that form the quad


			//<TODO:> find the four edges (their indices) that form the quad.
There could have multiple different scenarios!!!

			cur++;

		}
	}
}
```

For JavaScript version

```
///Data structure to store the data defined on the uniform grid
var node = {
        x: [],
        y: [],
        z: [],
        s: []
}

var lineseg = {
        n1: [],
        n2: [],
}

var quad = {
        verts: [],
        edges: []
}

var NX, NY;

////Load the data stored in uniform grid
function Load_data_on_uniformGrids(dat_file_path) {

  // create a file loader
  var loader = new FileLoader();

  //load a text file
  loader.load(
    dat_file_path, // resource URL

    // onLoad callback
    function (data) {

      // get line-by-line data
      var lines = data.split('\n');

      // get NX and NY from the first line
      NX = lines[0].split(' ')[0];
      NY = lines[0].split(' ')[1];

      for (var j = 0; j < NY; j++) {
        for (var i = 0; i < NX; i++) {
          // get the line index
          var curr_index = j * NX + i + 1;
          var line = lines[curr_index];
```

```
        // parse each line to get x,y,z and scalar values
         var line_values = line.split(' ');

         var node = {
           x: parseFloat(line_values[0]),
           y: parseFloat(line_values[1]),
           z: parseFloat(line_values[2]),
           s: parseFloat(line_values[3])
         };
         // add a new grid node
         grid_pts.push(node);

       }
     }
   },

   // onProgress callback
   function (xhr) {
     console.log((xhr.loaded / xhr.total * 100) + '% loaded');
   },

   // onError callback
   function (err) {
     console.error('An error happened in FileLoader');
   }
 );

};
```

The following two functions build the edge list and face list of the uniform grid, which may be useful if you decide to first compute the contour intersections before connecting them to form the complete contour. If you decide to use them, please finish the following highlighted parts of the code.

```
////Build the edge list for the loaded uniform grid
function build_edge_list()
{
        var i, j;
        var cur = 0;
        for (j = 0; j < NY - 1; j++)
        {
                cur = j * NX;
                for (i = 0; i < NX - 1; i++)
                {
           //<TODO:> Build a horizontal edge between vertex i and i+1

                  //<TODO:> Build a vertical edge between vertex i and i+NX


                        cur++;
                }
                  //<TODO:> Build a rightmost edge between vertex cur and cur+NX
```

```
	}

	// Build the edges on the top boundary
	cur = (NY - 1) * NX;
	for (i = 0; i < NX - 1; i++)
	{
		//<TODO:> Build a horizontal edge between vertex i and i+1
		cur++;
	}
}
////Build the quad face list for the loaded uniform grid
function build_face_list()
{
	var i, j;
	var cur = 0;
	for (j = 0; j < NY - 1; j++)
	{
		cur = j * NX;
		for (i = 0; i < NX - 1; i++)
		{
			//<TODO:> find the four vertices (their indices) that form the quad


			//<TODO:> find the four edges (their indices) that form the quad. There could have multiple different scenarios!!!

			cur++;

		}
	}
}
```

Be creative and have fun!