**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.

For all triangles{

For each edge of a triangle Si{

//Determine whether there is an intersection given s\* with the formula.

t = (s\* -s0 )/(s1-s0); if 0<=t<=1 then there is an intersection.

If (s0 <s\*<s1 || s1<s\*<s0){

Compute the intersections the same way as marching squares;

intersection\_counter++;

}

}

//According to the value of the intersection\_counter connect the intersections.

If intersections are 0 no line

If intersections are 1 then error

If intersections are 2 then join

If intersections are 3 then check if one of them is a vertex if not then return error

}

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

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

To Load and Visualize the data I used the sample formats given in the assignment hint section and filled in the highlighted parts. I stored the intersection points but however did not store the actual intersections instead I chose to use a stack and display each intersection as I calculated it and then emptied the stack.

To Implement the actual marching squares algorithm I used the basic marching squares function and looped through the vertices as pairs. For each pair I estimated the intersection points using the iso\_value taken from the user. Then based on the number of points generated per square I drew the lines joining the points.

For the case of multiple contours I looped the marching squares function by passing a different iso values generated uniformly through the range. Then the marching squares algorithm drew the lines for each contour.

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;

grid\_pts.clear();

////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);

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);

}

//building edge list

void build\_edge\_list() {

int i, j;

int cur = 0;

edgeList.clear();

lineseg temp;

for (j = 0; j < NY - 1; j++) {

cur = j \* NX;

for (i = 0; i < NX - 1; i++) {

temp.n1 = cur;

temp.n2 = cur + 1;

edgeList.push\_back(temp);

edge++;

temp.n1 = cur;

temp.n2 = cur + NX;

edgeList.push\_back(temp);

edge++;

cur++;

}

temp.n1 = cur;

temp.n2 = cur + NX;

edgeList.push\_back(temp);

edge++;

}

cur = (NY - 1)\*NX;

for (i = 0; i < NX - 1; i++) {

temp.n1 = cur;

temp.n2 = cur + 1;

edgeList.push\_back(temp);

edge++;

cur++;

}

// Building Faces

void build\_face\_list() {

int i, j, p;

int cur = 0;

quadList.clear();

quad temp;

for (j = 0; j < NY - 1; j++) {

cur = j \* NX;

for (i = 0; i < NX - 1; i++) {

//adding Vertices

temp.verts[0] = cur;

temp.verts[1] = cur + 1;

temp.verts[2] = cur + NX + 1;

temp.verts[3] = cur + NX;

square++;

quadList.push\_back(temp);

cur++;

}

}

}