MAPPING PLACENTAL VESSEL NETWORKS USING TRIANGULAR MESH TOPOLOGY

TRINH QUOC HUNG
APPLIED MATHEMATICS
CALIFORNIA STATE UNIVERSITY, LONG BEACH
HUNG135790@YAHOO.COM,

NANCY CHE MAHAN
APPLIED MATHEMATICS
CALIFORNIA STATE UNIVERSITY, LONG BEACH
NANCYCHE2@YAHOO.COM,

DAVID J HARR
APPLIED MATHEMATICS
CALIFORNIA STATE UNIVERSITY, LONG BEACH
DJHARR@GMAIL.COM

ABSTRACT. By analyzing the properties of the placenta, it is possible to make accurate predictions about the progression of certain conditions through the life of the child. One of these properties is the blood vessel network of the placenta, which is responsible for providing the child with oxygen and nutrients from the mother. We propose a new technique for automatically extracting the placental blood vessel network data from a 3-dimensional model of the placenta using inherent curvature of the surface of the placenta. By analyzing the curvature, we are able to identify features that likely correspond to blood vessels standing above the surface of the placenta. We discuss the mathematics underlying the technique, which makes use of concepts from graph theory, differential geometry, computer-aided geometric design, and computer graphics. We also provide details of a sample implementation of the algorithm and discuss the results, using some sample models.

1. INTRODUCTION

There have been many studies indicating a strong correlation between the shape, weight, and efficiency of the placenta and the health of the baby. This correlation appears to hold long past the infancy of the child. In fact, the claim has been made that the characteristics of the placenta is the single most important predictive factor in the health of the baby. In light of this, many people are analyzing placental types in order to be able to better use them for advance notice of possible health issues for a child. [1]

Date: May 19, 2011.

One of the characteristics of the placenta that is used to determine its "efficiency" is the network of veins and arteries that supply blood to the child in the womb. There have been various attempts to map this vessel network using automatic techniques working from images, but to date, no one has come up with a method that can reliably extract the vessel network of an arbitrary placenta without human intervention. [2]

This article presents a technique for extracting the vessel network of a placenta. This technique uses the topology of the 3-dimensional surface created from the placenta, which results in a triangle mesh that is a close approximation of the actual placenta. By analyzing the surface, we are able to use the topology of the mesh, primarily the curvature, to identify probable locations for blood vessels which protrude above the surface of the placenta. Then, by eliminating areas with non-matching curvature parameters, we are able to reconstruct the major features of the original vessel network.

2. Mathematical Theory

2.1. **Curvature.** Consider a 2-dimensional surface embedded in 3-space. Now, for a given point p on that surface, there are an infinite number of curves passing through that point in the surface. Each curve has an associated *curvature* associated with it, where the curvature describes how much the curve deviates from being a straight line at the point p. If we choose a small enough neighborhood for a curve around p, then the curve can be made to approximate a circle. Then, the curvature κ of the curve at p is defined as the reciprocal of the radius of the circle which most closely approximates the curve near p. In other words, for a curve C through p,

$$\kappa_C = \frac{1}{R},$$

with R being the radius of the circle most closely approximating C. Clearly, as a curve approaches a straight line, the value of R approaches ∞, and the value of κ approaches 0. Clearly, at p, there will be a curve with a maximum curvature and a curve with a minimum curvature. By convention, these are known as κ_1 and κ_2 to refer to maximum and minimum curvature, respectively. Together, κ_1 and κ_2 are known as the *principal curvatures*. The direction of the principal curvatures are at right angles to each other, and the sign of the the curvatures are arbitrary, but commonly, a positive curvature indicates a convex surface. The *mean curvature*, M, is defined as the average of the principal curvatures,

$$M = \frac{1}{2}(\kappa_1 + \kappa_2).$$

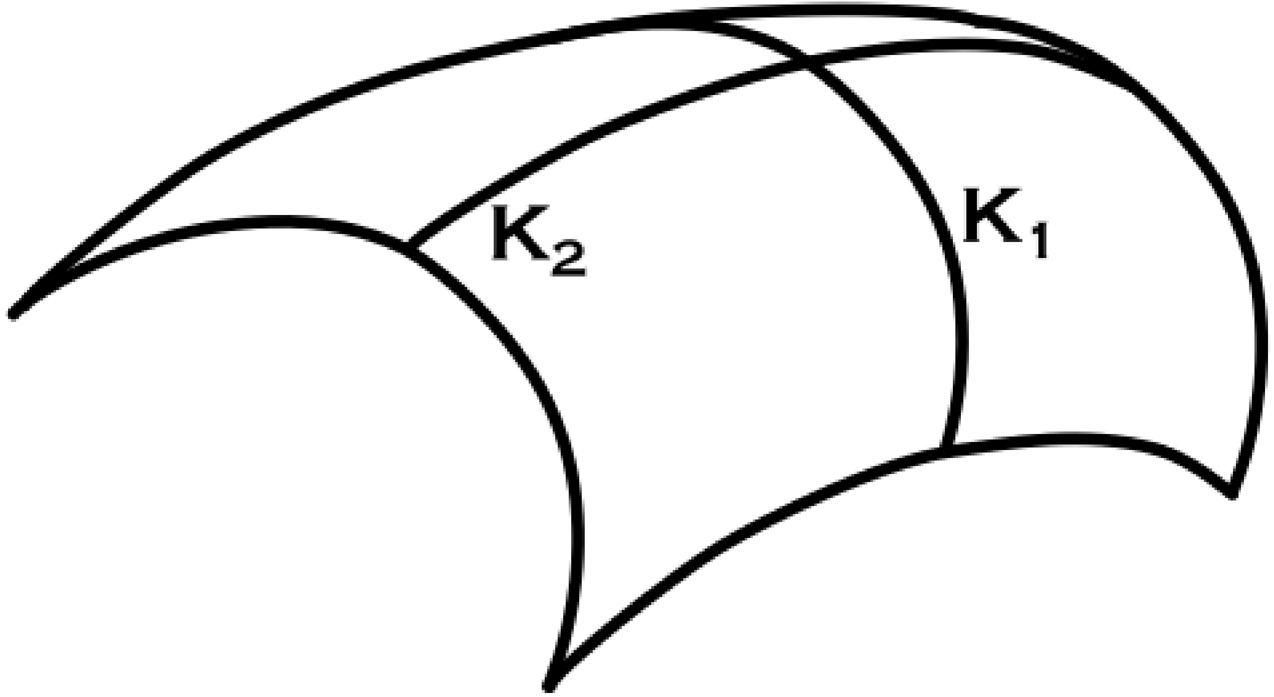

Figure 1: Principal Curvatures of a Surface, κ_1 and κ_2.

The *Gaussian curvature*, K, is defined as the product of the principal curvatures,

$$K = \kappa_1 \kappa_2.$$

2.2. **Surface Fitting.** In order to find the curvature at a specific point on the triangular mesh, we need to create a parametric surface that smoothly interpolates the mesh near the point of interest. This is done interpolating a series of vertices near the point of interest to obtain a *biquadratic Bézier Surface*, which consists of a rectangular mesh with 9 control points. The surface is defined using the equation

$$X(u,v) = \sum_{i=0}^{2} \sum_{j=0}^{2} b_{i,j} B_i^2(u) B_j^2(v).$$

where u and v are parameterizations of the point on the surface, b_{ij} are the Bézier control points, and B_i^2 are the Bernstein polynomials. [3]

After calculating the Bézier surface, we can calculate the curvature using the derivatives of the Bézier formula. The procedure we use is based upon the method described in [4].

3. Implementation

3.1. **STL File Formats.** The placental models are provided in a format known as STL for "stereo lithography." These are created by taking the placenta and slicing it into very small strips and digitizing the resulting pieces. By the nature of the file, it contains a large number of triangle records, with each triangle being completely self-contained. In addition, the triangle records can occur in any arbitrary order, making it impossible to discern any topological information from the mesh. In order to allow us to process the mesh and attempt to extract information such as the curvature, it was necessary to transform the data, creating relations among the various elements.

To begin, we read in all the triangle records. Each record consists of three vertices with three floating point coordinates corresponding to the x, y and z coordinates, respectively. In addition, there is a 3-dimensional vector that gives the normal to the triangular face. The vertices are specified in a counterclockwise order. Since each triangle is a standalone data chunk, there is a large amount of data duplication in the vertices. On average, we found that each vertex was repeated six times. In order to facilitate mesh processing, we eliminated the duplicate vertices, and converted each triangle to use indices into a vertex list instead of specifying the coordinates directly. Then we eliminate the duplicate vertices in the vertex list. Below is the c++ code to perform these operations.

```
// read in the actual triangle data
for(int i=0; i<num_triangles; ++i) {

    // put the triangle data into
    // temporary variable
    stl_tri temp;

    // read the data from the stl file and put
    // it into the temporary variable
    stl_file.read((char *)&temp, stl_tri_size);

    // three vertex variables
    v3d v1, v2, v3;

    // this loads the coordinate data from
    // stl triangle into the three vertex
    // variables
    load_verts(temp, v1, v2, v3);

    // This adds the 3 vertices to the vertex
    // list. We make no attempt to deal with
    // duplicate vertices yet
```

```
    vert_list.push_back(v1);
    vert_list.push_back(v2);
    vert_list.push_back(v3);

    // store the triangle into a triangle list
    // that we will use to build our index based
    // facet list later.
    tri_list.push_back(temp);
}

// this will sort all the vertices into lexicographical
// order, so that all duplicate vertices will occur
// in a series of adjacent entries in the array
std::sort(vert_list.begin(), vert_list.end());

// This checks the array for duplicate adjacent entries
// and moves them to the end of the array, returning a
// pointer to the beginning of the duplicates
std::vector<v3d>::iterator it = std::unique(vert_list.begin(),
         vert_list.end());

// This erases the array from the point where the
// duplicate entries begin to the end of the array
// leaving us with an array containing only one
// copy of each vertex
vert_list.erase(it, vert_list.end());
```

3.2. **Adjacency List.** After creating the vertex list, we need to convert the triangles to use entries into the vertex list instead of storing the coordinates directly. For this purpose, we create a new data type that stores indices into the vertex list to indicate the three vertices of the triangle. However, we still need to keep the normal information for the face, so we will need to add that, too. In order to create the Bézier surface, we do a nonlinear least squares fitting to a number of vertices around the point of interest. To do this, we require the ability to determine which vertices are connected to a given vertex by an edge. At the same time we match the vertex coordinates to the indices in the vertex list, we also add each edge of the triangle to an *adjacency list*, which, for each vertex, keeps a record of the vertices that share an edge with any given vertex. In additing, we need to be able to determine which triangles contain a particular vertex, so for each vertex, we keep a list of containing triangles.

The c++ code to accomplish this is given below.

```
for(std::vector<stl_tri>::iterator it = tri_list.begin();
        it != tri_list.end(); ++it) {
    // t is the original triangle data
```

```
// as read from the STL file
stl_tri t = *it;
facet f;


// copy the triangle normal to
// the new facet
f.normal.x = t.norm_x;
f.normal.y = t.norm_y;
f.normal.z = t.norm_z;

// normalize it to length 1
f.normal = unit(f.normal);

// copy the vertex coordinate data to
// three temporary variables
load_verts(t, v1, v2, v3);

// this returns the index of the first
// vertex in the triangle record
f.v1 = std::distance(frist,
    std::lower_bound(vert_list.begin(), vert_list.end(), v1))
            ;

// this returns the index of the second
// vertex in the triangle record
f.v2 = std::distance(frist,
    std::lower_bound(vert_list.begin(), vert_list.end(), v2))
            ;

// this returns the index of the third
// vertex in the triangle record
f.v3 = std::distance(frist,
    std::lower_bound(vert_list.begin(), vert_list.end(), v3))
            ;

// calculate the area of the triangle
f.area = norm(cross(v2-v1, v3-v1) * 0.5);

// calculate barycentric center -- this
// is where the normal emanates from
f.center = (v1+v2+v3) * (1.0f/3.0f);

// add the edges of the facet to the adjacency list
// v1->v2, v2->v3, v3->v1
boost::add_edge(f.v1, f.v2, a_list);
```

```
    boost::add_edge(f.v2, f.v3, a_list);
    boost::add_edge(f.v3, f.v1, a_list);

    // this calculates the percetage of the
    // triangle that each vertex occupies.
    // Used to calculate the normal at
    // vertex.
    calc_corner_area(f);

    // this adds this triangle to each
    // facet list for its component vertices
    add_facet_nodes(facet_list.size(), f);

    // appends the facet to the triangle list
    facet_list.push_back(f);
}
```

3.3. **Convex Hull.** In order to calculate the Bézier surface for the point, we need to determine the proper control points for the mesh. This involves finding the em minimal bounding box for the input points. We use a number of points near the triangle of interest. We need a large enough area that the curvature can be calculated, but not so large as to make the determination of the control points require too much computation. Based on work done by [5], we settled on using a set of input points corresponding to all vertices located within three edges of the point of interest. If we label the point of interest v, then all the vertices connected to v by a single edge, the so-called *neighborhood of* v is known by the designation $v*$. In the same way, all the vertices that lie within two edges of v consist of $v*$ as well as all the vertices that share an edge with the members of $v*$. This is known as $v**$. Finally, all the vertices lying within three edges of v consist of the members of $v**$, as well as all the vertices lying within one vertex of these vertices. This is known as $v***$. In order to construct this set of vertices, we first get all the vertices lying within one vertex of v, then recursively determine the neighborhoods of $v*$ and $v**$. This results in many duplicate members, so we have to prune the list, the same way we remove duplicates from the vertex list. Note that since we are calculating the normal at the center of the triangle, we start with the three vertices making up that triangle, instead of the actual triangle centroid. Figure 2 illustrates $v***$ for a triangle.

Below is the c++ code to create $v***$.

```
// initialize the neighborhood with the vertices
// of the triangle
v_A.push_back(f.v1);
v_A.push_back(f.v2);
v_A.push_back(f.v3);
```

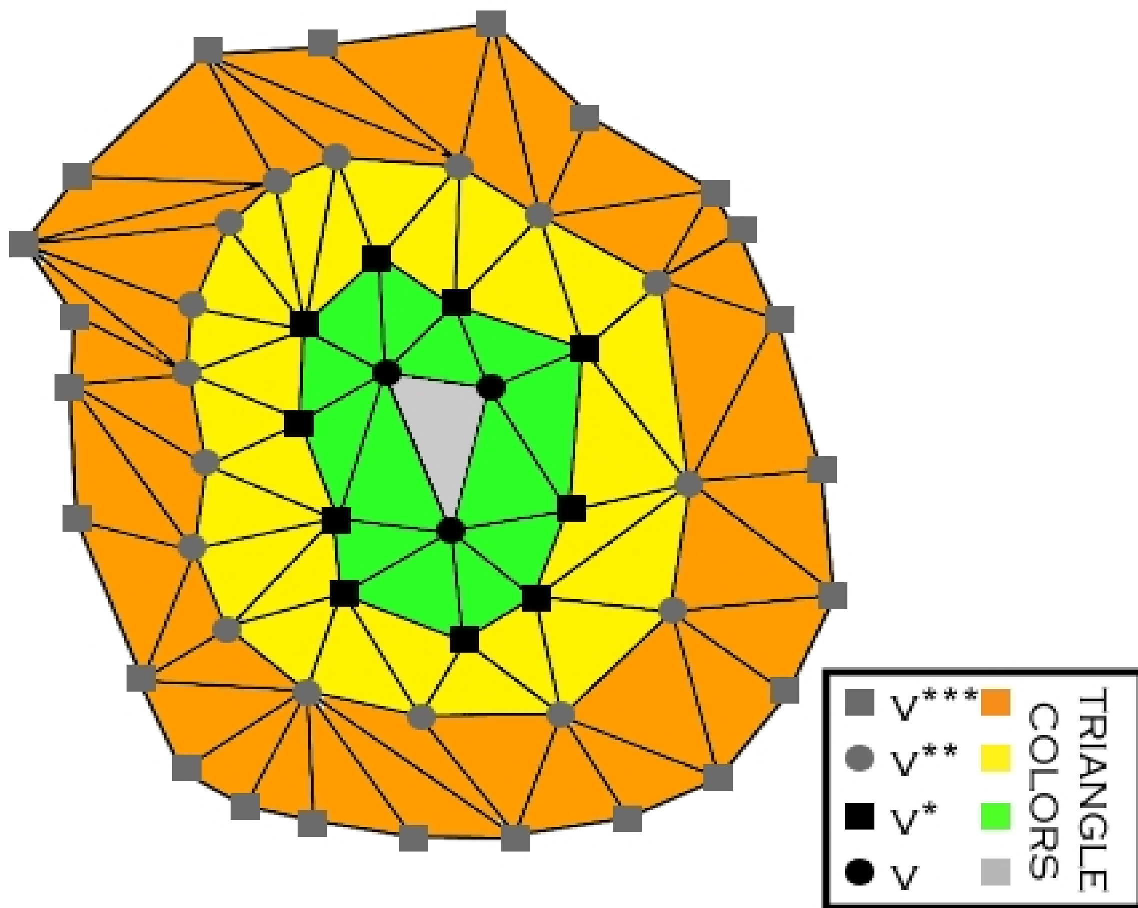

Figure 2: $v***$: 3-ring neighborhood surrounding vertex v.

```
// calculate v*
v_star(v_A, v_star_);

// this adds all the members of v**
// to the list of data points
add_to_neighborhood(v_star_,
    neighborhood);


// calculate v**
v_star(v_star_, v_star_star_);

// this adds all the members of v**
// to the list of data points
```

```
add_to_neighborhood(v_star_star_,
    neighborhood);

// calculate v***
v_star(v_star_star_, v_star_star_star_, true);

// this adds all the members of v**
// to the list of data points
add_to_neighborhood(v_star_star_star_,
    neighborhood);


// remove duplicates by first sorting, then
// moving duplicates to the back, then
// erasing them.
std::sort(neighborhood.begin(),
    neighborhood.end());
std::deque<v3d>::iterator it =
    std::unique(neighborhood.begin(),
        neighborhood.end());
neighborhood.erase(it, neighborhood.end());
```

Once we have the data points, we have to create a *convex hull* for the data set. The convex hull is the smallest convex polygon that can contain all the points in the data set. If the data points were nails driven into a piece of wood, the convex hull would be formed by a rubber band stretched around the outside of the nails.

The Convex Hull is a convex polygon that bounds the outermost edges of the 3-ring neighborhood v. It acts like an elastic rubberband that stretches open to encompass the object.

In order to construct the convex hull, we use a variation of the Graham scan method, as detailed in [4].

See Figure 3 for an example of a convex hull.

Here is the c++ code to create the convex hull.

```
std::sort(pts.begin(), pts.end());
std::deque<v3d>::iterator it = std::unique(pts.begin(),
    pts.end());
pts.erase(it, pts.end());


// create the dividing line
```

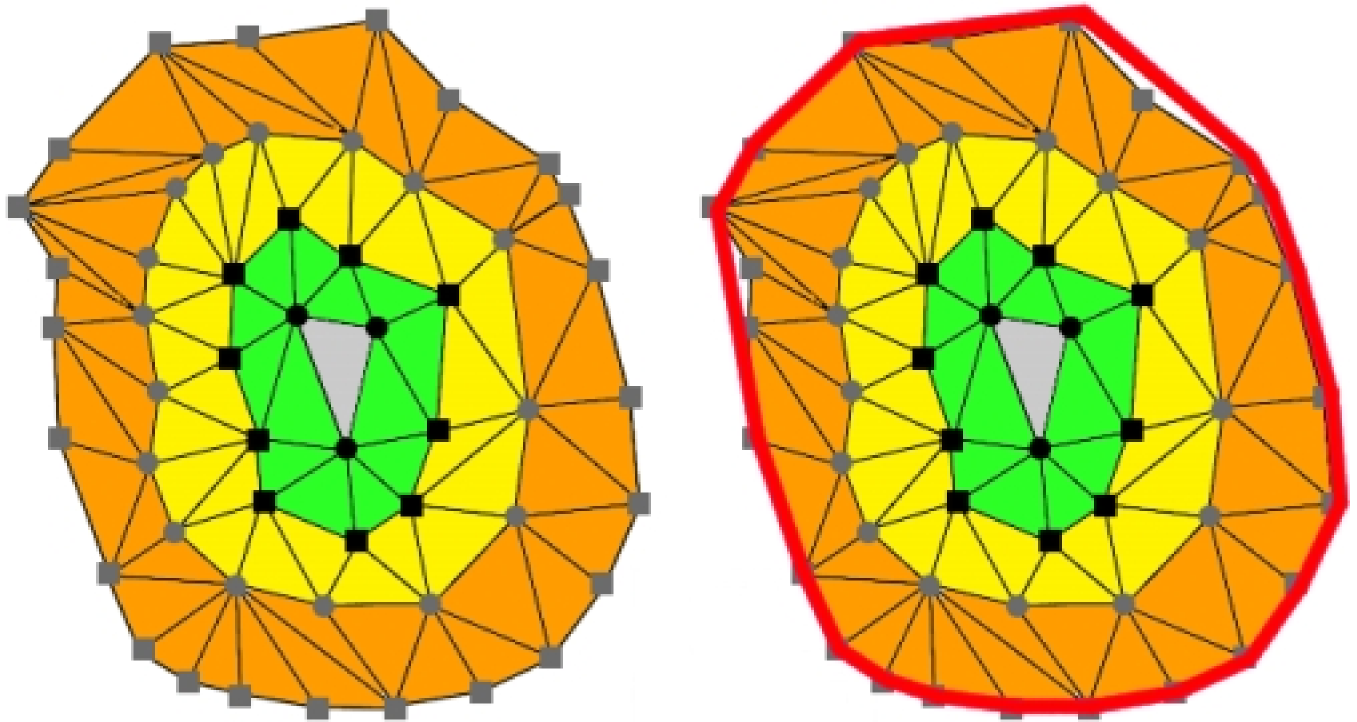

Figure 3: Convex Hull of Data Points

```
v3d left = v_copy.front(),
    right = v_copy.back();

v_copy.pop_front();
v_copy.pop_back();

float det_x = right.x - left.x;
float det_z = right.z - left.z;
float m = det_z/det_x;


std::deque<v3d> upper, lower;

upper.push_front(left);
lower.push_front(left);
// partition the vertices into points
// above and below the line left->right
while(!v_copy.empty()) {
    v3d v = v_copy.front();
    v_copy.pop_front();
    float slope = (v.z - left.z)/(v.x - left.x);
    if(m<=slope) {
```

```
        // above line
        upper.push_back(v);
    } else {
        // below line
        lower.push_back(v);
    }
}

upper.push_back(right);
lower.push_back(right);

out.clear();
// create upper hull
std::deque<v3d> u_hull(upper);
half_hull(u_hull, -1.0f);

// create lower hull
std::deque<v3d> l_hull(lower);
half_hull(l_hull, 1.0f);

// create final convex hull
// lower is in clockwise order, with right at the end
// add it to upper, and omit right
u_hull.pop_front();
u_hull.pop_back();
while(!u_hull.empty()) {
    l_hull.push_back(u_hull.back());
    u_hull.pop_back();
}

// l_hull contains the convex hull
std::deque<v3d>(l_hull).swap(out);
```

3.4. **Bezier Surface.** Our method of computing curvature at a mesh vertex requires approximation of the neighborhood of that vertex by a biquadratic Bezier surface. In order to calculate the Bézier surface, we need to determine the control points. We do this by creating an initial input for the control net, and then apply a non-linear least squares fitting to the control points in order to minimize the distance for each point from its parameterized analog on the surface. As an initial estimate, we create a minimal bounding box for the point set and then create a rectangle that contains the point of interest perpendicular to the normal of the facet, and then subdivide this rectangle into four quadrants and use the nine points defining these quadrants as the control points. However, in order to do this, we need to first find the minimal bounding box. To do this we use an algorithm called *rotating calipers*, which was developed by

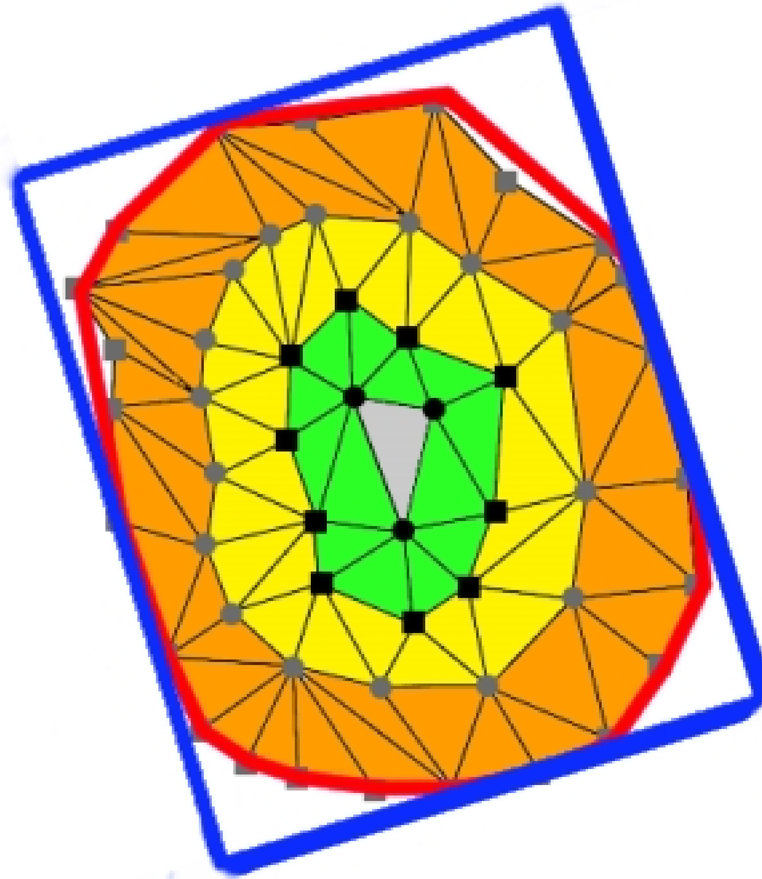

Figure 4: Minimal Bounding Box surrounding Convex Hull

Godfrey Toussaint [6]. This procedure involves rotating a rectangular shape around the convex hull, taking advantage of the fact that any minimal bounding box will have at least one side that contains an edge of the convex hull. See Figure 4 for an example.

Here is the c++ code to create the minimal bounding box.

```
const unsigned int num_box_pts = 4;
// array of current bounding box
// vectors associated with each point
// initialize to unit vectors in the
// positive and negative horizontal and
// vertical directions
v2d v2_list[4] = {v2d(0.0f, -1.0f), v2d(1.0f, 0.0f),
        v2d(0.0f, 1.0f), v2d(-1.0f, 0.0f)};

// find the vertices with the minimum and maximum x and z
// coordinates. These are p_i, p_j, p_k, and p_m,
// respectively. p_m is called p_l in the paper,
```

```
// we call it p_m because it is difficult to
// distinguish p_l (p_ell) from p_1 (p_one)
std::deque<v3d>::size_type p_i = 0, p_j = 0,
        p_k = 0, p_m = 0;
std::deque<v3d>::size_type pI, pJ, pK, pM;
v2d vI, vJ, vK, vM;

float max_y = in[0].y;
for(std::deque<v3d>::size_type i = 0; i<in.size(); ++i) {
    float x = in[i].x, z = in[i].z;
    float y = in[i].y;

    // vertex with minimum x value
    if(in[p_i].x > x) p_i = i;

    // vertex with maximum z value
    if(in[p_m].z < z) p_m = i;

    // vertex with maximum x value
    if(in[p_k].x < x) p_k = i;

    // vertex with minimum z value
    if(in[p_j].z > z) p_j = i;

    if(max_y < y) max_y = y;

}

max_y += 0.1f;


// now in[p_i] is minimum x, in[p_m] is minimum z
// in[p_k] is maximum x, and in[p_j] is maximum z
pI = p_i;
pJ = p_j;
pK = p_k;
pM = p_m;
vI = v2_list[0];
vJ = v2_list[1];
vK = v2_list[2];
vM = v2_list[3];
v2d p2_list[4] = {v2d(in[p_i]), v2d(in[p_j]),
        v2d(in[p_k]), v2d(in[p_m])};
v3d temp;
temp.y = max_y;
temp.x = vI.x;
```

```
temp.z = vK.z;

// initialize the corner matrix to
// the axis-aligned bb corners
out.push_back(temp);
temp.z = vM.z;
out.push_back(temp);
temp.x = vK.x;
out.push_back(temp);
temp.z = vJ.z;
out.push_back(temp);
out.clear();
float area = calc_bb_area(p2_list, v2_list);
// array of indices for current extreme
// points for calculating bounding box
std::deque<v3d>::size_type *p_list[num_box_pts] = {&p_i, &p_j, &
         p_k, &p_m};

// starter vector to determine when done
v2d first(in[wrap(in, p_i+1)] - in[p_i]);
first = unit(first);
// We now dot the unit vectors parallel with
// p_i->p_i+1, p_j-p_j+1, p_k->p_k+1,
// p_m->p_m+1etc with the vectors <0 0 1>,
// <1 0 0> <0 0 -1> and <-1 0 0>,
// respectively to get the cosines of the
// of the angles between each pair. Then,
// we choose the side with the smallest
// angle (the largest cosine).
int count = 0;
float dot_check = 1.0f;
std::vector<v3d> cur_bb;
while(dot_check > 0.0f) {
    // initialize to < min value of cos,
    // guaranteeing that it will hit at least
    // one cycle
    float max_cos = -2.0f;
    ++ count;
    unsigned int max_index = 0;
    v2d max_v;
    for(unsigned int ci = 0; ci < num_box_pts; ++ci) {
        std::deque<v3d>::size_type i0 = *p_list[ci], i1 = wrap(in
                 , i0+1);
        v2d old_v(v2_list[ci]);
        v2d new_v(in[i1] - in[i0]);
        new_v = unit(new_v);
```

```
        float cur_cos = dot(new_v, old_v);
        if(cur_cos > max_cos) {
            max_cos = cur_cos;
            max_index = ci;
            max_v = new_v;
        }
    }
    *p_list[max_index] = wrap(in, *p_list[max_index]+1);
    for(unsigned int i=0; i<num_box_pts; ++i) {
        unsigned int cur_i = (max_index + i)%num_box_pts;
        v2_list[cur_i] = max_v;
        max_v = v2d(-max_v.z, max_v.x);
        p2_list[cur_i] = v2d(in[*p_list[cur_i]]);
    }
    float new_area = calc_bb_area(p2_list, v2_list);
    if(new_area < area) {

        pI = p_i;
        pJ = p_j;
        pK = p_k;
        pM = p_m;
        vI = v2_list[0];
        vJ = v2_list[1];
        vK = v2_list[2];
        vM = v2_list[3];
        area = new_area;
        out.clear();
        for(unsigned int i=0; i < num_box_pts; ++i) {
            unsigned int ci = i, ni = (i+1)%num_box_pts;
            v2d p1 = p2_list[ci] - 20.0f*v2_list[ci], p2 = p1 +
                     40.0f*v2_list[ci];
            v2d p3 = p2_list[ni] - 20.0f*v2_list[ni], p4 = p3 +
                     40.0f*v2_list[ni];
            float x1 = p1.x, x2 = p2.x, x3 = p3.x, x4 = p4.x;
            float z1 = p1.z, z2 = p2.z, z3 = p3.z, z4 = p4.z;

            float x = (((x1*z2 - z1*x2) * (x3 - x4)) -
                    ((x1 - x2) * (x3*z4 - z3*x4)))/
                    (((x1 - x2) * (z3 - z4)) -
                    ((z1 - z2) * (x3 - x4)));
            float z = (((x1*z2 - z1*x2) * (z3 - z4)) -
                    ((z1 - z2) * (x3*z4 - z3*x4)))/
                    (((x1 - x2) * (z3 - z4)) -
                    ((z1 - z2) * (x3 - x4)));
            v3d temp;
```

```
            temp.x = x;
            temp.y = max_y;
            temp.z = z;
            out.push_back(temp);
        }
    }
```

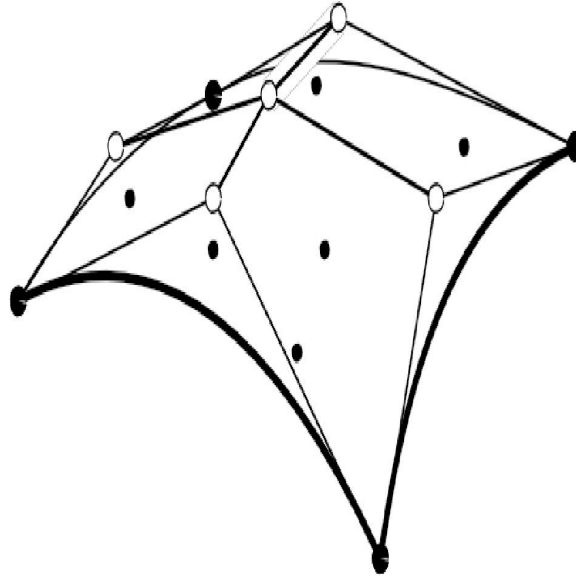

Figure 5: Biquadratic Bezier patch.with 9 control points

A Bezier surface can be thought of as a surface constructed of patches. Each patch is defined as the image of a Bezier function that maps the unit square into a smooth-continuous surface embedded within a space of the same dimensionality. In order to compute the curvature of this Bezier surface, we need to obtain the 9 Bezier control points $b_{i,j}$. These 9 points are obtained by the least squares fit method, in which the $v***$ set of points are used as data points in the nonlinear least squares fit method. Unfortunately, this is the point at which we ran out of time, as we have been unable to get the fitting to work properly. See Figure 5 for an example of a biquadratic Bézier surface.

4. Future Directions

We have code to evaluate a Bézier surface, but the least squares packages we have been using seem to fail at fitting the data points in order to determine the control points. We believe that the problem is the parameters we are using for the least squares fit.

Once we have determined the Bézier coefficients for the local surface, we can then go forward with trying to determine the location of the vessels by using the areas of extreme curvature to pinpoint “ridge” locations, which are likely to correspond to protruding blood vessels. Once we obtain the portion of the vessels protruding above the surface of the placenta, it is our hope that we can then use this geometry to reconstruct the major features of the vessel network, including the diameter of the most prominent arteries and veins.

5. Acknowledgements

We would like to thank Dr. Salafia and her team for providing the original STL files that we used in preparing this report. We would also like to thank Professor Jen-Mei Chang for her advice and encouragement on this project. In addition, we would like to thank all the members of the Math 579 class for great questions and ideas that significantly improved this project.

References

[1] C M Salafia, M Yampolsky, D P Misra, O Shlakhter, D Haas, B Eucker, and J Thorp. Placental surface shape, function, and effects of maternal and fetal vascular pathology. *Placenta*, 31(11):958–62, 2010.

[2] Nizar Almoussa, Brittany Dutra, Bryce Lampe, Pascal Getreuer, Todd Wittman, Carolyn Salafia, and Luminita Vese. Automated vasculature extraction from placenta images, 2009.

[3] Gerald Farin. *Curves and surfaces for CAGD - a practical guide (3. ed.).* Computer science and scientific computing. Academic Press, 1992.

[4] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[5] Anshuman Razdan and MyungSoo Bae. Curvature estimation scheme for triangle meshes using biquadratic bézier patches. *Comput. Aided Des.*, 37:1481–1491, December 2005.

[6] Godfried Toussaint. Solving geometric problems with the rotating calipers. In *In Proc. IEEE MELECON Õ83*, pages 10–02, 1983.