3D Visualization of Hearts

Johan Henriksson
mahogny@areta.org
830730-7135

Background

Sahlgrenska is working on a cardiac imaging technique. Essentially, points on the inner surface are located in an absolute coordinate system. Using these, an algorithm is to be made which visualizes the heart.

Tools & Usage of Final Program

Matlab was used for this project. It turned out to have the wanted features although it is a bit lacking in visualization.

To use the code, store down the coordinates from an Excel sheet into a pure text file. Load it into a Matlab variable and pass it to the function unconvhull3(). Send the output to plotpoly(). See example code.

The Simple Approach

The first idea would be the following;

1. Enlist all triangles that can be made
2. Sort triangles by increasing area
3. Pick out triangles in order such that at all times triangles for a surface

This thing is $O(n^3)$ and pretty KISS. When I finally got my hands on the data, I realized that the surface points are too few for this algorithm to be stable. Something else is needed.

A Better Approach

What we are essentially dealing with is reconstruction of a non-convex hull. Unlike the convex which is unique, the non-convex obviously isn't. There is however in some sense, “a most probable non-convex hull”. If we consider the method with which we are getting points, we are trying to reach out as far as possible only to hit some obstacle. The obstacle is on the surface (not really. just the idea). Hence we are actually looking for the biggest space that crosses all surface points. A good candidate is the convex hull with minimum shrinking. So essentially, we are going to

1. Find convex hull
2. Minimally shrink the hull
3. Post processing (NURBS?) to improve solution
4. Visualize

The first step is easy; algorithms can be found in any textbook on computational geometry. The shrinking can be done arbitrarily but there are certain choices that are more natural than others. Post-processing will not be done here since I have little clue about the

geometry of the heart (and care should be taken as to not correct the image if actually the heart isn't correct. that would make the tool useless for diagnosis). Visualization is a triviality.

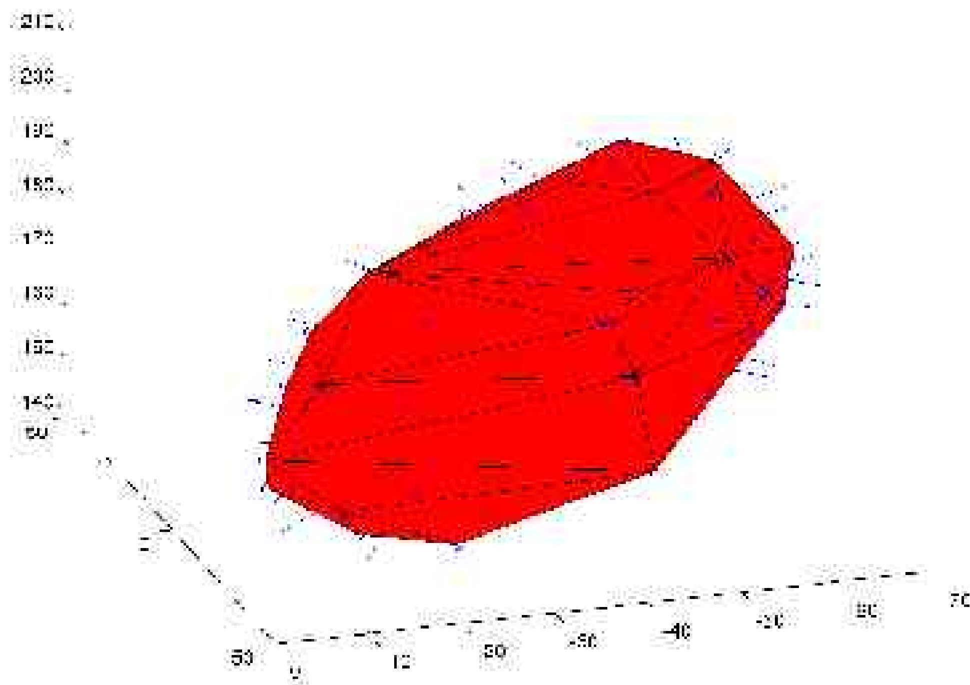
Computing the Convex Hull

I grabbed my favorite textbook on the subject and found an $O(n^2)$ incremental algorithm that is fairly trivial to implement (read: stupid). However, the one in the book is for C and cannot be directly translated into Matlab so I ended up only reusing the idea.

Essentially, this is what to do:

```
Start with a polyhedra
ForAll points p,
        ForAll surfaces that are visible from p, remove
        Create surfaces joining the point and the edges around removed surfaces
```

This is just a matter of linear algebra. For details, see comments in code. Here is an example of the convex hull of a heart:

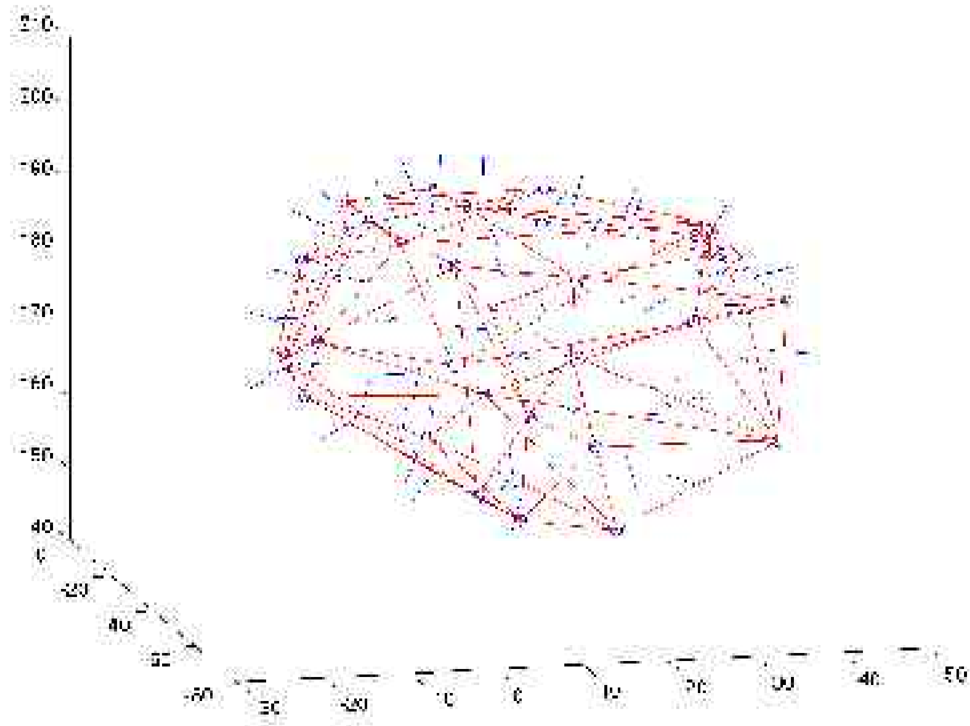


The blue lines are normals, the circles are measured points.

Shrinking the Convex Hull

The algorithm for shrinking can be selected arbitrarily. Here I deploy a greedy algorithm. We will, as long as there are points left, shrink to the point which is closest to a surface. As with the first approach, this requires a certain regularity but I believe this one will be much more stable as it can make a clever guess. It's a bit expensive, $O(sn^3)$. On the other hand, the set of points is small from

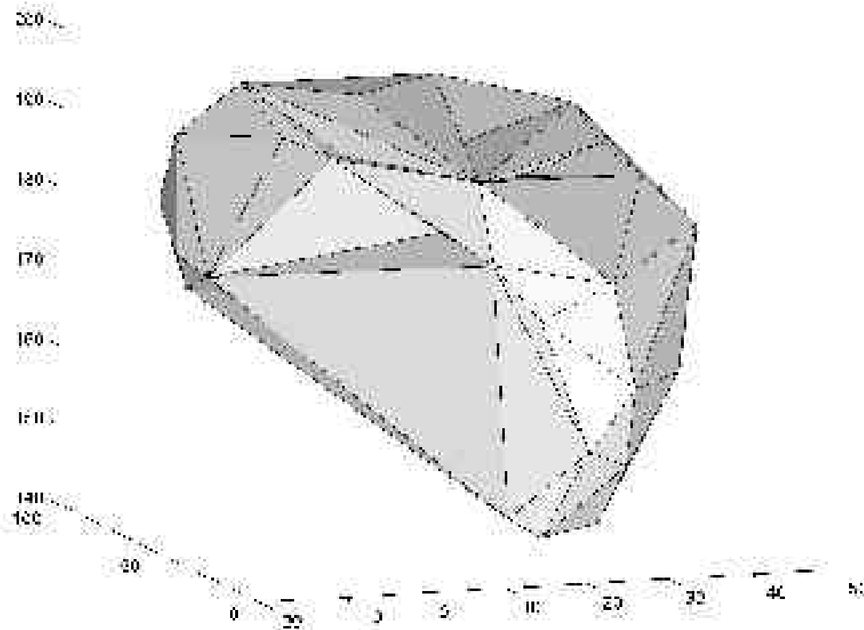
the beginning (at least in the test sets) and it has become even smaller from computing the hull. The more convex it is, the less will be left. The test sets are pretty convex. This wireframe gives a hint (but one has to turn it around to actually see it):



Now to see what we can do about it; here is the algorithm, in pseudo-code:

```
While there are points left

        ForAll surfaces

                ForAll points

                        Compute distances

        Take point and surface with smallest distance

        Remove surface and create surfaces joining the old edges with the point
```

This is what the heart looks after shrinking:



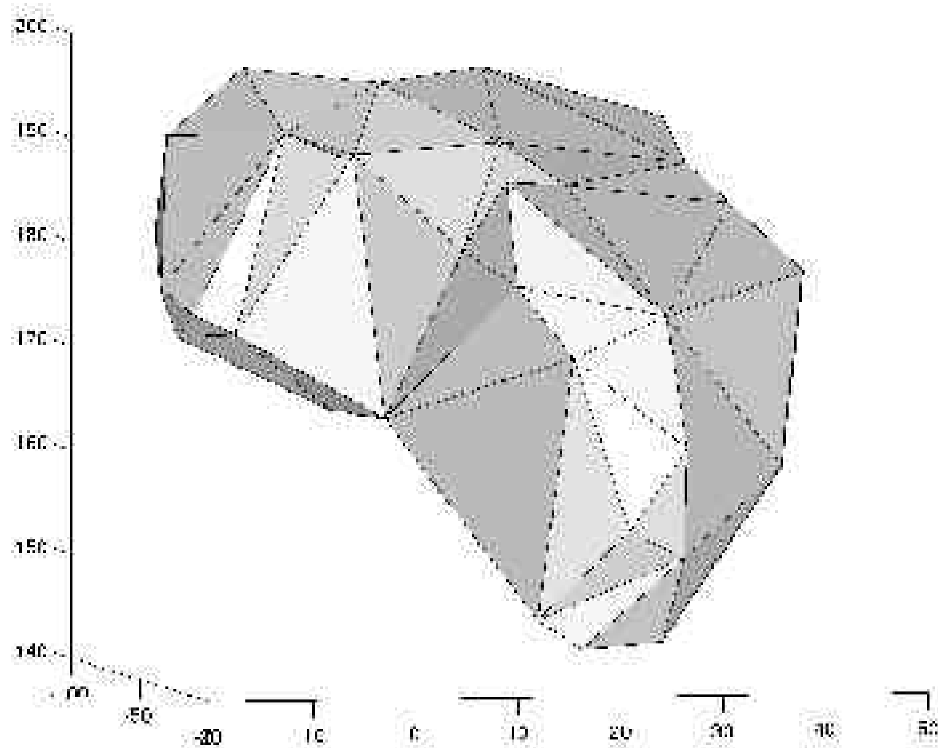
Getting Rid of Bad Slopes

It turns out that this algorithm isn't done just yet. The above image has some slopes that I am in doubt about. Because we just move down a single polygon, the shrinking is very local and really doesn't affect the hull. Can we smooth out a bit? Certainly. I believe rotating the splitting of squares can lessen the effect a lot. Two examples are seen above. When do we rotate a square? The goal is to minimize the angles between triangle normals (reduces slope). A simple iterative approach that converges might be worth a shot:

```
Until we are satisfied
        Take 2 triangles that share an edge
        Compute angle
        Rotate and compute angle
        Keep if new angle is smaller than before,
                otherwise restore
```

In lack of a good iteration scheme, just picking the pairs at random a few dozen times will suffice.

This is what comes out:



It's not free from distortions. I believe the next step would be some kind of simulated annealing, where the chance of rotation depends on how big the gain is. If the slope is reduced a lot, the chance of rotation is high. The chance of any rotation at all decreases over time. There are theoretic results that tells that this in general gives the optimum result, given long enough time.

Further Post-processing

Here I just leave some suggestions since one needs a better understanding of heart geometry than I do for this.

Ultimately, fitting a model of a real heart with texture and everything would give the best idea of the shape. This wouldn't be impossible by formulating it as a continuous minimization problem (certainly non-linear) given a single known point as a constraint. This requires higher resolution of points than what is available to me.

NURBS or a similar technique could be used to smooth out the surfaces. Even better would be if a biomechanical model of heart musculature were available. One could then find the actual surface using a similar minimization problem (as one knows how the musculature would prefer to bend). The algorithm would essentially start by fitting all the flesh precisely around the model. The polygons would be further tesselated down to whatever granularity is required (like FEM). The measured points could become constraints. Then a relaxation would be done like the

rotation of triangle pairs right now, except now some points are free to move, smoothing it all out (This completely replaces the rotation step).

Visualization

Matlab has the command fill3() and friends which pretty much solves the whole problem.

Conclusions

The algorithm made here gives something that I believe resembles a heart and executes in about 5 seconds on a 2.8Ghz P3. This shows that the problem is far from impossible. Simple geometry can be made just out a basic idea of the measuring process. A better visualization would require some biomechanical knowledge of the heart tissue.

Further Work

The Matlab implementation of convex hull search doesn't use any sensible data structures making it far from $O(n^2)$ but rather as slow as the shrinker. If bigger data sets are to be used, the algorithm should be reimplemented with better structures (and maybe change to a language with pointers like Java).

If better quality is wanted, post processing should be implemented.

References

Joseph O'Rourke, **Computational Geometry in C,** Cambridge University Press, ISBN 0-521-64976-5

Matlab Code

Compute Euler distance:

```
function r=mag(v)
r=sqrt(sum(v.^2));
```

Compute Euler distance squared:

```
function d=dist2(v1,v2)
d=sum((v1-v2).^2);
```

Plot polygons

```
%Plot a set of polygons
function plotpoly(p,v)
px=[];
py=[];
pz=[];
hold on
colormap(gray);
for i=1:size(p,1)
        n1=p(i,1);
        n2=p(i,2);
        n3=p(i,3);
        px=[v(n1,1), v(n2,1), v(n3,1)];
        py=[v(n1,2), v(n2,2), v(n3,2)];
        pz=[v(n1,3), v(n2,3), v(n3,3)];

        n=normal(p(i,:),v);
        n=n./sqrt(sum(n.^2));
        rcol=n(1)/2 + 0.5;
        col=[rcol;rcol;rcol];

        fill3(px',py',pz',col)

        %plot3(px',py',pz','r')
end
hold off
%plot normals
%plotpolynorm(p,v);
%plot points
%plotpoints(v);
```

Plot points

```
function plotpoints(v)
hold on
plot3(v(:,1),v(:,2),v(:,3),'o');
hold off
```

Plot normals to the polygons

```
%Plot a set of polygon normals
function plotpoly(p,v)
hold on
for i=1:size(p,1)
        mx=mean(v(p(i,:),1));
        my=mean(v(p(i,:),2));
        mz=mean(v(p(i,:),3));

        n=normal(p(i,:),v);
        n=5.*n./sqrt(sum(n.^2));

        plot3([mx, mx+n(1)], ...
              [my, my+n(2)], ...
              [mz, mz+n(3)], ...
              'b');
end
hold off
```

This is the code to compute the convex hull, the first step

```
%Function that computes the convex hull of a 3d set
%p=convhull3(v)
%p is a list of three indexes into v, one set for each polygon
function p=convhull3(v)

%Number of vectors
nv=size(v,1);

%Build a trivial polyhedra first by taking the first 4 points
%Here a major cheat is applied; add 4 new points in the middle of
%the heart. These will be removed over time but saves me from
%having to figure out the orientation if I had taken 4 existing points.
%The polyhedra is made so small that it is guaranteed to fit.
mv=mean(v);
s=(max(v(:,1))-min(v(:,1)))/50;
v=[v; ...
   mv+[0,0,0]; ...
   mv+[s,0,0]; ...
   mv+[0,s,0]; ...
```

```
   mv+[0,0,s]];

%Initial set of polygons
p=[1,2,4; ...
   2,3,4; ...
   1,4,3; ...
   2,1,3]+nv;

%Now, induction over all points
%for i=1:4
for i=1:size(v,1)
        nextv=v(i,:);

        %Check which faces are visible
        pkeep=[];
        prem=[];
        for j=1:size(p,1)
                %Normal for surface
                n1=normal(p(j,:),v);

                %Normal towards point
                n2=nextv-v(p(j,1),:);

                %Check angle between these two
                if dot(n1,n2) >= 0
                        prem=[prem;p(j,:)];
                else
                        pkeep=[pkeep;p(j,:)];
                end
        end

       if size(pkeep,1) > 0
                %The point is not within the convex polyhedra. Otherwise discard it
                %Start with the polygons to keep
                p=pkeep;

                %Now we wish to connect new polygons between the new point and the edges
                %that are left. Which are the edge points?
                %By "moving" out a polygon, we get the normal right and everything. So
                %look for a polygon to reuse
                for j=1:size(prem,1)
                        %Consider all three edges
                        for k=1:3
                                e=prem(j,k);
                                if k==3
                                        f=prem(j,1);
                                else
                                        f=prem(j,k+1);
                                end
```

```
                                %Is there some polygon sharing this edge?
                                %If not, then this is a border edge. Use it
                                c=0;
                                for m=1:size(prem,1)
                                        c=c+((sum(prem(m,:)==e)+sum(prem(m,:)==f))>1);
                                end
                                if c==1
                                        p=[p; e, f, i];
                                end
                        end
                end
        end
end
```

The second step, to compute the non-convex hull

```
%Function that computes an approximate maximal-size non-convex hull
%p=unconvhull3(v)
%p is a list of three indexes into v, one set for each polygon
function p=unconvhull3(v)

%Start by computing the convex hull
p=convhull3(v);
np=p;

%Figure out which points are left out
vo=[];
for i=1:size(v,1)
        if sum(reshape(p,1,size(p,1)*3)==i) == 0
                vo=[vo;i];
        end
end

%vo=vo(1,:);

vo2=vo;

%Now, shrink until all points are on the surface
while size(vo,1)>0
        %Find shortest distance between a point and a surface
        mind=10^100;
        for i=1:size(p,1)
                thep=p(i,:);
                for j=1:size(vo,1)
                        thev=v(vo(j),:);
                        %Project down to surface
                        v1=v(thep(1),:);
                        v2=v(thep(2),:);
                        v3=v(thep(3),:);
```

```
                va=v2-v1;
                vb=v3-v1;
                n=cross(va, vb);

                %v1 + a*va + b*vb + c*n = thev
                %Solve:
                A=[va', vb', n'];
                B=A\(thev-v1)';
                a=B(1);
                b=B(2);
                c=B(3);

                %Find closest point of intersection (some cheating being done here)
                if a<0
                        a=0;
                end
                if b<0
                        b=0;
                end
                ab=a+b;
                if ab>1
                        a=a/ab;
                        b=b/ab;
                end

                %Find distance
                ip=v1+a.*va+b.*vb;
                thed=sum((thev-ip).^2);

                %Check if this is better than before
                if thed<mind
                        mind=thed;
                        minv=j;
                        minpoly=i;
                end
        end
end

%What is the current polygon?
cp=p(minpoly,:);

%Replace this face with 3 new, connected to the vertex and the edges
thev=vo(minv);
p(minpoly,:)=[cp([1,2]),thev];
p=[p;
   cp([2,3]),thev;
   cp([3,1]),thev];

%Remove this vertex from the set
vo=vo([1:(minv-1),(minv+1):size(vo,1)],:);
```

```
end
p=rotsmooth(p,v);
```

The third step, to smooth out

```
%Smoothes a tesselated (fully joined) surface by trying to rotate triangle-pairs
function p=rotsmooth(p,v)

%Number of polygons
np=size(p,1);

for wtf=1:1000
        %Pick a polygon at random
        p1i=round(rand*(np-1)+1);
        p1=p(p1i,:);

        %Pick an edge at random
        e=round(rand*2)+1;
        f=e+1;
        if f>3
                f=1;
        end
        g=f+1;
        if g>3
                g=1;
        end

        v1=p1(e);
        v2=p1(f);
        v3=p1(g);

        %Find neighbouring triangle
        for i=1:np
                ne=sum(p(i,:)==v1)+sum(p(i,:)==v2);
                if ne==2 && i~=p1i
                        p2i=i;
                        p2=p(p2i,:);
                        break;
                end
        end

        %Find fourth vertex
        v4 = p2.*(p2~=v1);
        v4 = v4.*(v4~=v2);
        v4 = sum(v4);

        %Compute angle between original
        n1=normal(p1,v);
        n2=normal(p2,v);
```

```
        a1=dot(n1,n2)/(mag(n1)*mag(n2));

        %Form two new polygons to compare with
        p3=[v1,v4,v3];
        p4=[v3,v4,v2];

        %Compute new angle
        n3=normal(p3,v);
        n4=normal(p4,v);
        a2=dot(n3,n4)/(mag(n3)*mag(n4));

        %Compare angles
        if a2 > a1
                %New version has less angle between normals. Update polygons
                p(p1i,:)=p3;
                p(p2i,:)=p4;
        end
end
```