# Finite Element Methods
# Final Project
# Spring 2013

David Witman

April 29, 2013

# 1 Introduction

For the final project we were given a number of options to choose from ranging from purely theoretical to purely computational. I choose to investigate the use of the Finite Element Method on non-rectangular or polygonal domains. To accomplish this, an existing MATLAB package called `mesh2d` was used to create a grid of points given an outline of a domain. `Mesh2d` was written by Danny Engwirda and is available through the MATLAB file exchange.

Often we wish to solve problems where the boundaries cannot be represented by a rectangular domain so it becomes necessary for us to find some other method of discretizing our domain. `Mesh2d` implements a Delaunay triangulation method to split up the domain into a set of triangles that will become our finite elements. The aim of Delaunay triangulation is to increase the minimum angle of any triangle within the domain, this avoids long and skinny triangles which we noted in class were not particularly good elements. Using just a few commands we can create a non-rectangular domain and use the `mesh2d` function to automatically discretize it, adjusting the size and shape of our elements as necessary. The following sections will go through some background of the `mesh2d` program, some simple test cases, an analysis of two non-rectangular domains and any results gathered.

# 2 Background of Mesh2d

## 2.1 Delaunay Triangulation

As mentioned, `mesh2d` implements the Delaunay triangulation to discretize the domain. Triangulation refers to splitting up a continuous space into a set of discrete simplices on the space. A simplex refers to a geometric space composed of the smallest number of edges. So in the case of two dimensions, a triangle is the smallest geometric shape on the space and thus, a simplex. Delaunay triangulation attempts to minimize the angles of the simplex we are working with to avoid long and skinny triangles.

Before we can triangulate though we must understand the concept of Voronoi tessellations. Voronoi tessellations take a set of points, called generators, and define regions corresponding to each of these points. These regions are constructed such that if one were to choose a random point in the space, the region that it fell in would define the closest generator to that point. Defining the voronoi tesselation will then help us to shape a set of triangles that minimize the angles.

If we have a set of generators, each with a defining region, then we will set the adjacent regions/generators as potential matches for the vertices of each of our triangles. This means that there are at least three possible points that could be matched with each generator as vertices of the triangle. This property also applies to vertices of the tessellation, giving us a set of points that we can triangulate. One thing that is particular to Delaunay triangulation is the property of being locally Delaunay. This property refers to the the shared edge of two triangles:

- Two triangles $abc$ and $bcd$ with shared edge $bc$
- If $d$ lies within or on the circle defined by points $a$, $b$ and $c$ then the edge is locally Delaunay
- This implies that point $a$ is contained within the circle defined by $bcd$

Using these local Delaunay properties we can define the lemma:

**Lemma 2.1** *If every edge in a triangulation $K$ of $S$ is locally Delaunay then $K$ is the Delaunay triangulation of $S$*

## 2.2 Structure of Mesh2d

Now that we understand the concepts of a Delaunay triangulation we can examine how the `mesh2d` package works to implement the Delaunay triangulation.

`Mesh2d}` takes up to 4 inputs:

1. node: $xy$ data defining the boundaries of the domain.
2. edge: Defines how each of the points defined in the `node` variable are connected to each other. This is useful if you want to define a domain with a boundary inside of it like an annulus or a cylinder in a cross-flow.
3. hdata: This input allows us to set information regarding the size of each element. This option must be defined as a matlab struct data type with options like the maximum allowable element size, specific edge size settings and a user defined function depending upon the position of the edge in the domain.
4. options: Allows the user to set options on the formulation of the triangulation including options like the maximum number of iterations and convergence tolerance.

Once the inputs are gathered, `mesh2d` puts together the 2D domain and splits it via a quadtree decomposition. Quadtree decompositions create a data-structure where each parent has exactly four children. This data structure is then used to iteratively generate sets of faces, or elements that obey the Delaunay triangulation criteria. The loop follows this structure:

1. Check that the current geometry is valid ie. no hanging nodes, edges are unique etc.
2. Perform The Delaunay triangulation as explained in the last section to create the triangular elements.
3. For all the previous iterations:
   (a) Check to see the amount that the mesh edges have changed
   (b) If the mesh edges haven't changed more than a tolerance, exit the subloop.
4. Test overall convergence using the formula:

   $$\frac{\sqrt{\sum l^2}}{h} < 3 \tag{1}$$

   Where $l$ represents the edge lengths of each of the elements and $h$ is the average distance from the midpoints each edge to the vertices. If the convergence criteria is satisfied, exit the loop.

Using this procedure an unstructured triangular mesh on a non-rectangular domain can be produced.

## 3 Modifications to the code

Since we have already created a code that solves the two-dimensional heat equation on a rectangular domain with structured triangles there are only a few modifications that needed to be made. The biggest change was in the geometry routine; which previously went through and created the whole domain and elements. Now all we need is a routine that reads in a set of three files:

- Geometry file: Contains the $xy$ points defining the nodes within our domain.
- Nodes file: This file will essentially become our local/global array that points each element to the nodes that comprise it. So it will have three columns with an integer for each element pointing to the node that makes up a vertex of the current element.
- Unknown file: This file will become the unknown array. A negative number represents a known value at a particular node ie. a boundary.

In order to read and write the data files that were used, a comma separated value (csv) package called csv_io was used. This package can be found on John Burkardt's website under the Fortran90 source code section.

Another modification that is worth mentioning is that previously I kept track of the nodes associated with an element in a clockwise pattern; `mesh2d` stores them in a counter-clockwise mapping. This meant that the `phi` routine needed to be changed to ensure that the quadrature point was actually getting picked up as inside the triangle. The nice thing about barycentric coordinates is that regardless of whether the mesh is structured or unstructured, the basis functions are evaluated the same. This means that there were few major modifications to make to the existing code other than the changes mentioned in the geometry routine.

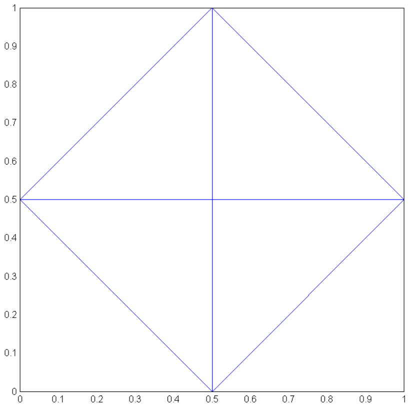
## 4 Test case

In order to validate that the changes made to the program are indeed working, a simple square domain with homogeneous Dirichlet boundary conditions was used as a test case. To generate the mesh the following commands were used:

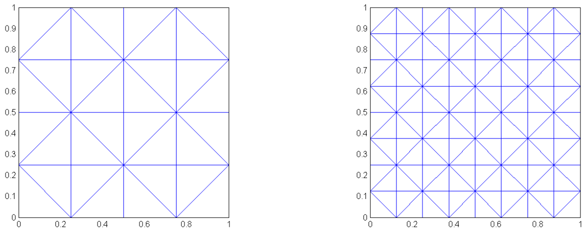
```
% Try a square
node = [0,0;1,0;1,1;0,1];

% Set the edges
edge = [1,2; 2,3; 3,4; 4,1];

% Set the discretization
hdata.hmax = .5;

% Create the mesh
% p is the list of xy nodal points we have
% t corresponds to the node indices and matches local to
%   global nodes
[p,t] = mesh2d(node,edge,hdata);
```

Using this code the following plot is produced:

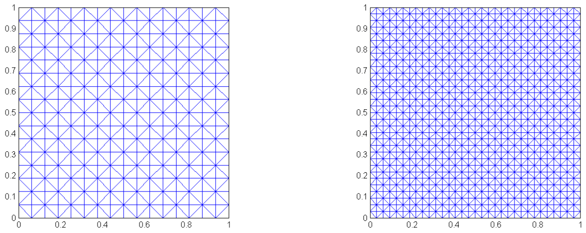
Figure 1: Rectangular mesh at $h = 0.5$



As we increase the discretization our elements should get incrementally smaller and our grid more and more refined:



(a) $h = \frac{1}{4}$ (b) $h = \frac{1}{8}$



(c) $h = \frac{1}{16}$ (d) $h = \frac{1}{32}$

Figure 2: Various Discretization's

Now using these discretization's and our newly modified code, we can calculate some convergence rates using the same test problem that was used in the last assignment:

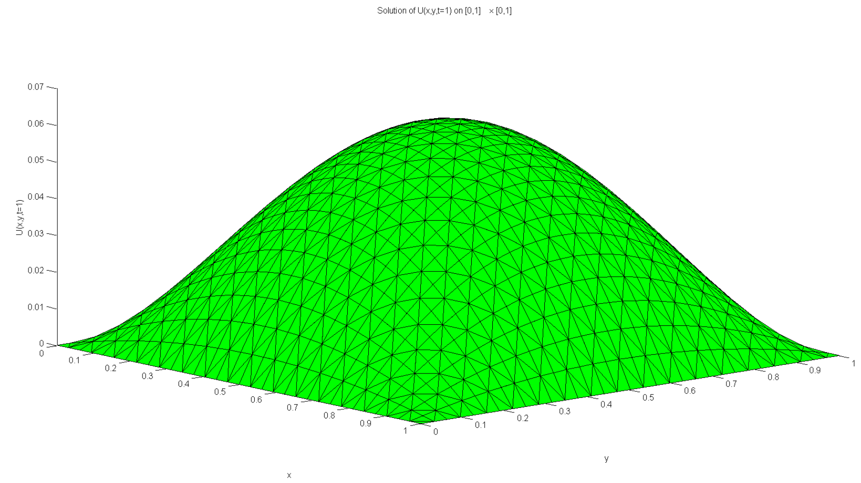
$$u_t - \Delta u = 3(x^2 - x)(y^2 - y)t^2 - 2t^3(y^2 - y + x^2 - x) \tag{2}$$

With $(x, y) \in \Omega = (0,1) \times (0,1), t \in (0,1)$ and $u = 0$ on the boundary of the domain, with initial condition: $u(x, y, t) = 0$. Whose exact solution is:

$$u(x, y, t) = x(x-1)y(y-1)t^3 \tag{3}$$

Implementing the changes to the code, mentioned above and running it for the described resolutions we can compute and plot the solutions. The plot shown below is the solution when $h = \frac{1}{32}$

Figure 3: Rectangular mesh at $h = 0.5$



Using this test problem we can construct a rate of convergence table, shown below:

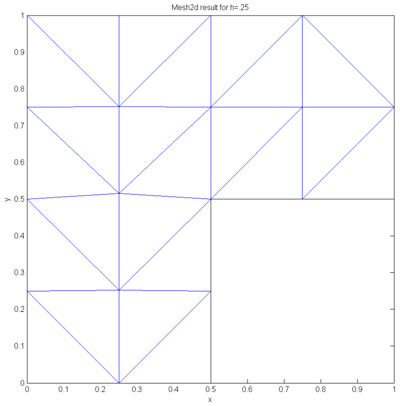
| $h$ | $L_2$ Error | $H^1$ Error | Convergence rate $L_2$ | Convergence rate $H^1$ |
|---|---|---|---|---|
| 0.500000 | 0.745475 | 0.872661 | | |
| 0.250000 | 0.157730 | 0.374413 | 2.240701 | 1.220792 |
| 0.125000 | 0.039029 | 0.187054 | 2.014839 | 1.001171 |
| 0.062500 | 0.009741 | 0.093727 | 2.002361 | 0.996925 |
| 0.031250 | 0.002432 | 0.046879 | 2.001960 | 0.999506 |

From the Rate of convergence table we see that our new Finite Element code converges as we expected with linear basis functions on triangular elements. From this we can conclude that our code is working properly and we can now go ahead and test some more complex geometries.

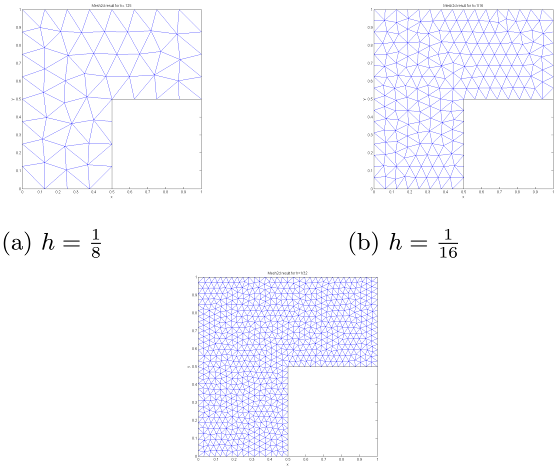
# 5 Test case 1

The first geometry that will be tested is a polygonal boundary that contains a sharp edge. Sharp edges are notoriously difficult to model using Finite Element Methods because there is the potential for singularities occurring at our around the edge. The domain pictured below will be the test case we will attempt to model and discretize using our `mesh2d` software.

Figure 4: Rectangular mesh at $h = 0.5$



And then using this defined domain we can increase the discretization which will be needed when we attempt to gather our convergence rates.



(a) $h = \frac{1}{8}$

(b) $h = \frac{1}{16}$

(c) $h = \frac{1}{32}$

Figure 5: Various Discretization's for sharp edge domain

We notice that there is no discretization when $h = \frac{1}{2}$, this is because `mesh2d` forced the discretization at $h = \frac{1}{2}$ to be $h = \frac{1}{4}$. This is most likely because it doesn't want to fit only two triangles in the lower left quadrant. We also notice that unlike the previous example, the mesh does not appear to be very structured, in fact it gets worse as the discretization increases. At $h = \frac{1}{4}$ the lines separating the upper and lower left quadrants appear slanted, this trend is just amplified in higher discretization's. The reason for this is the sharp edge; the Delaunay triangulation has a tough time handling an edge that juts out into the domain. Sharp edges are used often in flow problems like flow against a wall or something similar. It is very important that the grid be resolved around these edges otherwise singularities can occur.

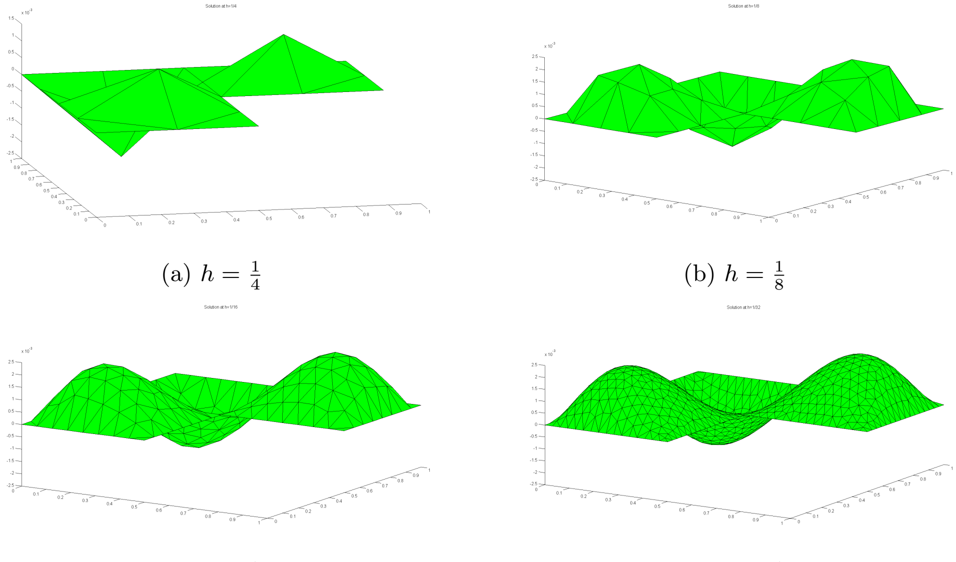
For this test case we will stick with the two-dimensional heat equation with homogeneous Dirichlet boundary conditions and adjust the solution to the differential equation. Because now we no longer have a simple domain with only four edges we now need to account for the edges that are sticking out into the domain in order to make sure that our solution complies with the homogeneous Dirichlet boundary conditions. The solution that fits this problem can be represented by:

$$u(x,y,t) = xy(x-1)(y-1)(x-\frac{1}{2})(y-\frac{1}{2})t^3 \tag{4}$$

Knowing that our differential equation can be written as: $u_t - \Delta u = f(x,y,t)$ we can determine the right hand side of the differential equation should be:

$$f(x,y,t) = 3xy(x-1)(y-1)(x-\frac{1}{2})(y-\frac{1}{2})t^2 - (6x-3)(y-1)(y-\frac{1}{2})t^3 - (6y-3)(x-1)(x-\frac{1}{2})t^3 \tag{5}$$

Using this right hand side and the knowledge about the domain we can run our FEM code to determine an approximation to the solution over the domain. The following four plots are the solution to the differential equation at time $t = 1$ with $\Delta t = \Delta x^2$.



(a) $h = \frac{1}{4}$

(b) $h = \frac{1}{8}$

(c) $h = \frac{1}{16}$

(d) $h = \frac{1}{32}$

Figure 6: Solutions at different Discretization's

From the solutions we can see that as we increase our refinement our solution appears to converge to an actual solution. To verify this we can calculate the convergence rates, shown in the table below:

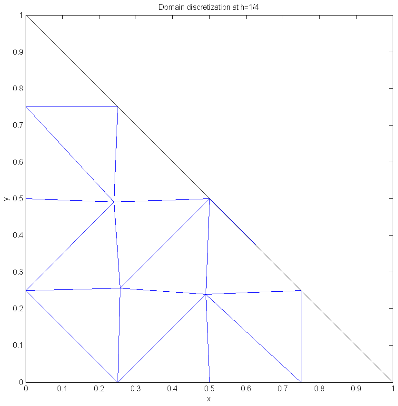
| $h$ | $L_2$ Error | $H^1$ Error | Convergence rate $L_2$ | Convergence rate $H^1$ |
|---|---|---|---|---|
| 0.250000 | 0.648873 | 0.959344 | | |
| 0.125000 | 0.174740 | 0.890761 | 1.892726 | 0.107010 |
| 0.062500 | 0.050304 | 0.874477 | 1.796460 | 0.026618 |
| 0.031250 | 0.020322 | 0.870661 | 1.307608 | 0.006310 |

We notice that although the $L_2$ Error appears like it may be close to expected convergence rate of 2, the $H^1$ error is nowhere close. This can be attributed to the extremely high order of the solution.

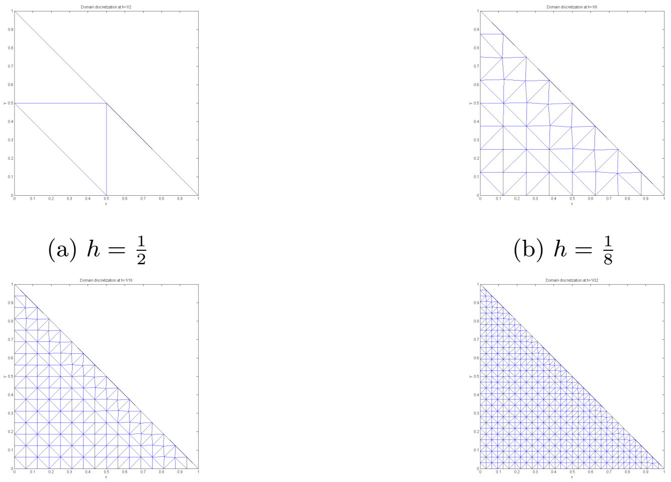
## 6 Test case 2

The second test case that we would like to run is in some ways simpler than the previous problem. The second case again solves the heat equation; but this time on a triangular domain. In theory this domain should be relatively easy to discretize as we are trying to split up a triangle the more smaller triangles but we will see how that works out. The meshed domain below represents the domain that we would like to solve our differential equation over.

Figure 7: Rectangular mesh at $h = 0.5$



And then using this defined domain we can increase the discretization which will be needed when we attempt to gather our convergence rates.


Domain discretization at h=1/2
Domain discretization at h=1/8
Domain discretization at h=1/16
Domain discretization at h=1/32

(a) $h = \frac{1}{2}$

(b) $h = \frac{1}{8}$

(c) $h = \frac{1}{16}$

(d) $h = \frac{1}{32}$

Figure 8: Various Discretization's for triangular domain

But we must notice that for discretization $h = \frac{1}{2}$ there are no unknown nodes! This means that we probably shouldn't try to solve our differential equation at this discretization. From these resolved domains we notice that even though there appears to be some irregularities in some of them, the domain is relatively structured unlike the previous test case. Though there are a few spots that the mesh does not appear to conform to any sort of pattern.

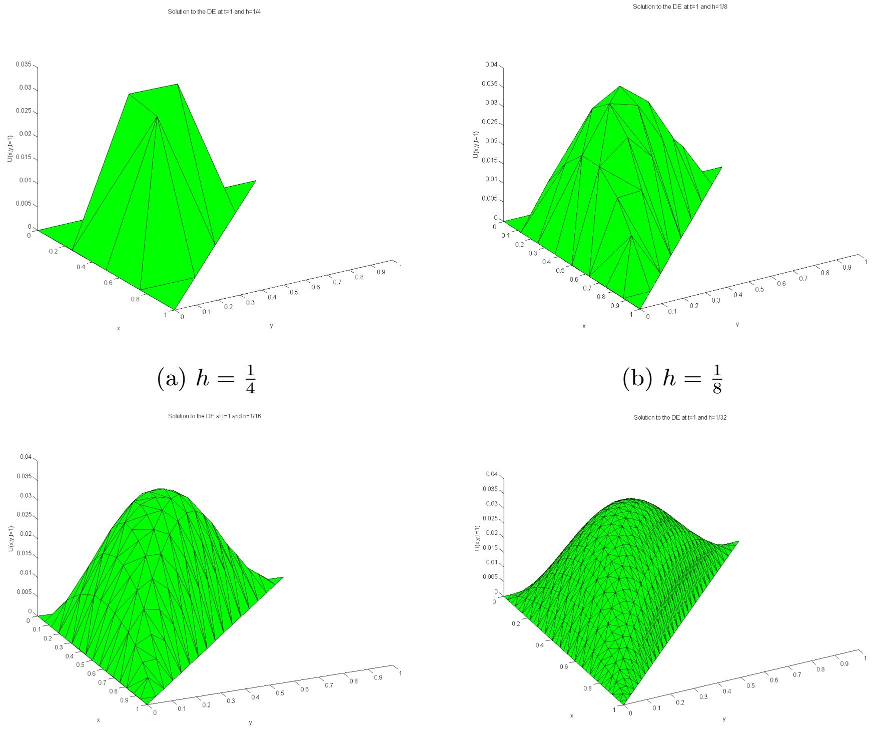
Now if we wish to keep our homogeneous Dirichlet boundary conditions we once again need to modify the exact solution that we wish to solve the differential equation for. The two boundaries where $x = 0$ and $y = 0$ are simple; the tricky one is the diagonal. But one property we know is that the equation $y = 1 - x$ defines that line so we should be able to say $1 - (x + y) = 0$ and that will be our final side. So the solution to our differential equation can be written as:

$$u(x, y, t) = xy(1 - x - y)t^2 \tag{6}$$

Thus our differential equation becomes:

$$u_t - \Delta u = 2t(xy - x^2y - xy^2) - 2yt^2 - 2xt^2 \tag{7}$$

Using our finite element heat equation code to solve this problem yields the following solutions:



(a) $h = \frac{1}{4}$ (b) $h = \frac{1}{8}$

(c) $h = \frac{1}{16}$ (d) $h = \frac{1}{32}$

Figure 9: Various Solutions for triangular domain

From the plots of the solutions it appears that as we increase our resolution, the solution appears to converge. To verify this we can calculate the rate of convergence at some of our different discretization's.

| $h$ | $L_2$ Error | $H^1$ Error | Convergence rate $L_2$ | Convergence rate $H^1$ |
|---|---|---|---|---|
| 0.250000 | 0.375170 | 0.609626 | | |
| 0.125000 | 0.129630 | 0.363553 | 1.533147 | 0.745759 |
| 0.062500 | 0.034890 | 0.188638 | 1.893513 | 0.946546 |
| 0.031250 | 0.008891 | 0.095594 | 1.972362 | 0.980626 |

From this table we see that as we increase the resolution, our convergence rate is trending towards 2 and 1 in the $L_2$ and $H^1$ error norms respectively. The fact that they are not nearly exact like the first test case we did is due to the domain and how it it discretized. Although this domain appears to be much better than the sharp edge examined in the last test case.

# 7 Conclusion, Remarks and a bonus (partial) problem

Through the analysis of three test cases it is clear that the `mesh2d` package is a robust tool that implements the Delaunay triangulation on non-rectangular domains. Starting with a test case on a rectangular domain, it was shown that the 2 dimensional heat equation with homogeneous Dirichlet boundary conditions converges to the exact solution as we increase our resolution. Then two other test cases were presented: one with a sharp edge jutting out into a rectangular domain and the other a triangular domain. Although the solutions did not converge as fast as the first test case, it was clear that there was convergence. Irregularities in the domain and difficulties trying to form the mesh are possible sources of error that could have contributed to the poor convergence but a thorough analysis of where the error is coming from is outside the scope of this project.

Overall I really enjoyed this project, it was a lot of fun putting together what we have learned and coded throughout the semester with an external meshing software and observing how everything works together. Initially I was very optimistic and wanted to try some very complex domains; but unfortunately I learned how quickly this introduces error. And since my code only implements piecewise linear elements on triangles; its ability to model complex solutions is limited. I was pleasantly surprised at how easy it was to integrate this new software into my existing code as everything was already set up the way that our code needed it to be.

During the preliminary stages of this project I attempted to try to use FEM to solve a differential equation over a very complex domain like the gulf coast shoreline near St. Marks FL. NOAA has a database of latitude/longitude coordinates of all shorelines that have been mapped. The three plots below are: the data points extracted from the NOAA database, near St. Marks, the resolved domain using the `mesh2d` package and a zoom in on an inlet from the Gulf of Mexico.

Figure 10: Data extracted from NOAA database

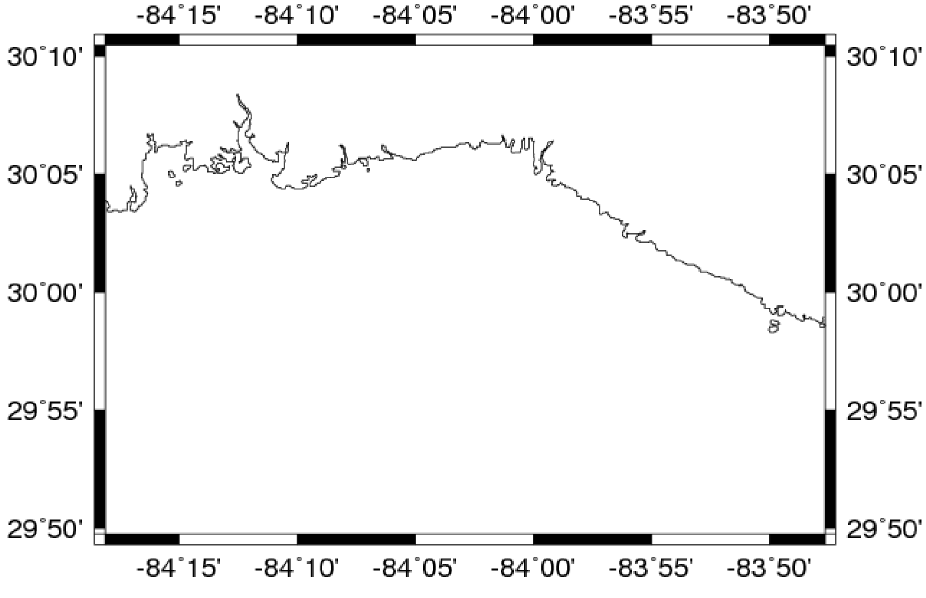


Figure 11: Meshed Domain

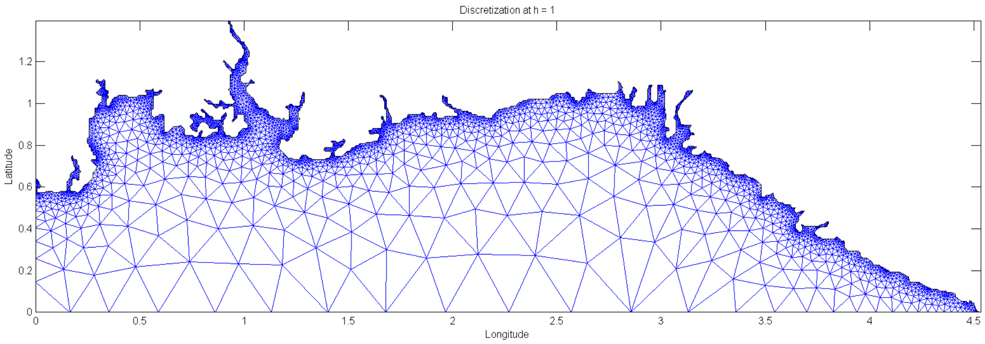
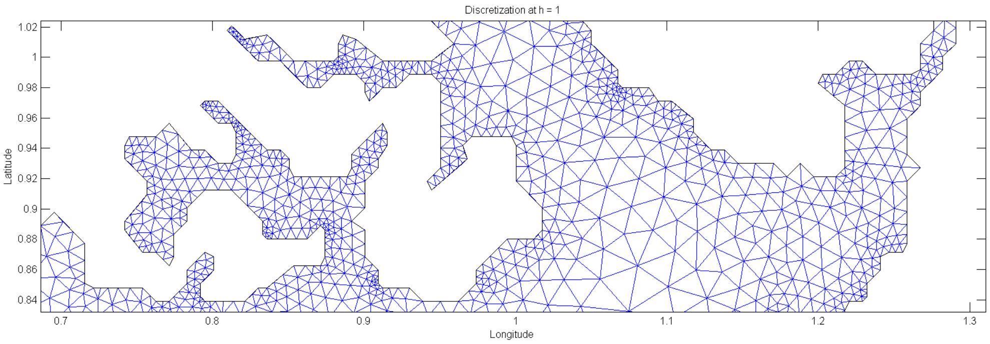


Figure 12: Zoom in on inlet



## 8 Work Cited

1. NOAA National Geophysical Data Center, Coastline extractor tool: http://www.ngdc.noaa.gov/mgg_coastline/index.js
2. Duke fall 2008 CPS230 Lectures: Lecture 21 on Delaunay triangulations, http://www.cs.duke.edu/courses/fall08/cps230 21.pdf
3. John Burkardt presentation: The Poisson Pump and the Spitting Fish (on Delaunay triangulations) http://people.sc.fsu.edu/ jburkardt/presentations/vt_2005.pdf