AN INTRODUCTION TO FINITE ELEMENT METHODS FOR SOLVING PARTIAL DIFFERENTIAL EQUATIONS ON ARBATRAIRY BOUNDARIES

ARDEN RASMUSSEN

Error

Remove

Appendix/Citation

Addition

Revise

1. Introduction

The Method of Finite Elements (FEM) is extreamly useful in numericaly solving differential equations. We examine the use casses for partial differential equations on a provided boundary condition. This means that our algorithms, and mathematics must be mostly independent of the boundaries and the conditions at the boundaries.

2. Mesh Generation

For the implementation of FEM, we must discretize our global domain into a set of finite subdomains, which we call elements. We can then compute our approximation on each element independent of the rest of the domain, then combine the solutions of each element into our global solution.

These elements can be of the form of an N sided polygon. However, with polygons with more than 3 edges, issues can arise. So for the purposes of FEM, we will be focused on the construction of a mesh of triangles. This is the most common mesh polygon, as any mesh of polygons with more edges can be represented by a triangular mesh. In addition, most research of mesh generation has been focused on triangular meshes for this same reason.

Our process will be construction what is known as a Constrained Delaunay Triangulation, and then apply a refinement algorithm to ensure that the triangles in the mesh are “nice”.

2.1. Delaunay Triangulation.

2.1. Delaunay Triangulation. Delaunay triangulation is the straight line dual of the Voronoi diagram. Delaunay triangulations are used in many different situations, including our purpose of mesh generation for finite element analysis.

Add example image of Delaunay triangulation.

A key property of Delaunay triangulation is that no point may lie inside the circumcircle of any other triangle. We use this as our definition of Delaunay Triangulation.

Definition 2.1 (Delaunay Triangulation)**.** Let S be a set of points in the plane. A triangulation T is a *Delaunay triangulation*(DT) of S if for each triangle t of T there exists a circle C with the following properties:

(1) the vertices of the triangle t are on the boundary of circle C
(2) no other vertex of S is in the interior of C.

Add image showing circumcircle definition.

Is most situation the Delaunay triangulation of a set of points is unique. This is true for all points with one exception of a square, where there are two valid Delaunay triangulations.

A major advantage of Delaunay triangulation is that it avoids triangles with small included angles. This makes this type of triangulation extremely well suited for our purpose of FEA.

There are many different algorithms that can be used for the construction of Delaunay triangulation of a set of points in a plane. Some notable ones include; Divide and Conquer developed by Chew[Che89], Incremental developed by Watson, Sweep Line developed by Žalik[Ž05][Dv08], and Edge Flipping developed by Sloan[Slo87][Slo93]. Each of these methods has different advantages and different efficiency in computation time.

Date: December 2, 2018.

2.2. **Constrained Delaunay Triangulation.** The Constrained Delaunay Triangulation(CDT), is a modification of the Delaunay Triangulation such that specified edges are forced to exist in the triangulation. This has the unfortunate effect of the resulting triangulation not being strictly Delaunay. Thus we define our constrained triangulation at the closest triangulation to the Delaunay triangulation that still includes the specified edges.

Add image showing constrained triangulation.

The constrained edges are most commonly arise for the purpose of defining boundaries of the domain or enforcing a medium change. This makes it necessary for any desired domain without a strictly convex outer hull that is generated from the unconstrained triangulation.

There are several different methods of construction the CDT. A few of these generate enforce the constraints during the initial construction of the Delaunay triangulation. However, many more use a preconstructed Delaunay triangulation and then proceed to enforce the edges, and refactor the mesh around the forced edge.

2.3. **Edge Flipping.** We utilize an implementation of edge flipping algorithms developed by Sloan. Sloan's edge flipping algorithm has implementation specifics for both constrained and unconstrained Delaunay triangulation.

We will provide a short explanation of the method that is used for the construction of our triangular mesh, but more detail can be found in [Slo87][Slo93]. First Sloan's algorithm constructions the unconstrained Delaunay triangulation then enforces the required edges into the triangulation. So, we begin with the process for construction the unconstrained triangulations.

2.3.1. *Construction of Delaunay Triangulation.* Using this method also provides the advantage of generating a triangle adjacency list for each of the triangles. This allows for optimizations later in the process of finite element analysis.

There are seven main stages of the construction of the Delaunay triangulation. We also define G as the set of points to be triangulated, and N as the number of points in G.

1. Normalize coordinates of points.
2. Sort points into bins.
3. Establish the super triangle.
4. Loop over each point, repeating 5-7
5. Insert new point in triangulation.
6. Initialize stack.
7. Restore Delaunay triangulation.

In stage 1, we scale all the points that will be used to construct the triangulation between 0 and 1, this should be done such that all relative positions of points are unchanged. This is done using

$$\hat{x} = \frac{x - x_{min}}{d_{max}}, \quad \hat{y} = \frac{y - y_{min}}{d_{max}},$$

where

$$d_{max} = \max\left\{x_{max} - x_{min},\ y_{max} - y_{min}\right\}.$$

This will scale all coordinates between 0 and 1 but will maintain the relative positions to all other points.

In stage 2, the normalized points are sorted into a set of bins. Each bin is associated with a rectangular portion of global space that the points are included in. We construction the bins such that each bin contains roughly $\sqrt{N}$ points. The bins are then ordered such that adjacent bins are numbered consecutively. This is done to improve the efficiency of later searching of the triangular mesh.

In step 3, we construction a "super triangle". This triangle is a triangle of three additional points, such that all points in G are enclosed within the super triangle.

In stage 4, we repeat stages 5-7 for every P in G.

In step 5, we determine the triangle which encloses P(Since the super triangle encloses all points, then this triangle must exist for P). We now delete this triangle, and construct three new triangles, by connection P to the three vertices of the deleted triangle.

In step 6, we add up to three triangles that are adjacent to the edges opposite P to a stack(A *stack* is a last in first out data structure, like a stack of trays).

In step 7, while the stack is not empty we proceed as following

7.1. Remove triangle from top of the stack.
7.2. If P is within the circumcircle of this triangle, then the adjacent triangle containing P and the unstacked triangle form a convex quadrilateral whose diagonal is drawn in the non-optimal direction. Swap this diagonal edge.
7.3. Add any new triangles which are opposite P to the stack.

After the stack is empty, then Delaunay triangulation has been restored.

Once every point has been added to the triangulation, then the completed Delaunay triangulation can be returned.

2.3.2. *Construction of Constrained Delaunay Triangulation.* Now that a Delaunay triangulation has been constructed for our set of G points, we now modify the triangulation so to ensure that certain edges are present.

This process proceeds as follows

1. Loop over each constrained edge.
2. Find intersecting edges.
3. Remove intersecting edges.
4. Restore Delaunay triangulation.
5. Remove superfluous triangles.

In step 1, we loop over every edge to be constrained, and we define the edge by the endpoints of the edge V_i and V_j. Using this edge repeat steps 2-4.

In stage 2, we find all edges in the triangulation that intersect $V_i - V_j$. Store all of these edges in a list. If our constrained edge is already present in the triangulation, the go to step 1.

In step 3, repeat while there are still edges that intersect $V_i - V_j$.

3.1. Remove an edge from the list of intersecting edges, call this edge $V_k - V_l$.
3.2. If the triangles that share the edge $V_k - V_l$ do not form a strictly convex quadrilateral, then place the edge back on the list of edges and go to step 3. Else, swap this diagonal, and define the new diagonal as $V_m - V_n$. If $V_m - V_n$ still intersections $V_i - V_j$, then place it back on the list of new edges, otherwise place $V_m - V_n$ on a list of new edges.

In step 4, repeat until no changes occur.

4.1. Loop over each newly created edge, define each edge by $V_k - V_l$.
4.2. If $V_k - V_l$ is equal to $V_i - V_j$, then skip to step 4.1.
4.3. If the two triangles that share the edge $V_k - V_l$ do not satisfy the Delaunay criterion, then swap the diagonal, and place the new diagonal on the list of new edges.

In step 5, we need to remove all unwanted triangles. For this stage, we differ from the process of Sloan. This is because for our implementation we desire the ability to have holes in the mesh, and the simplistic method of removing exterior triangles will not achieve this.

We use what is known as a triangle infection algorithm. Where we define a few points that will infect the triangles which the points are within, then the infection spreads to all adjacent triangles that are not separated by a constrained edge. This process is repeated until no new triangles become infected. Then all infected triangles are removed.

With this process, we need to define a point that is within every one of the holes in the mesh, as the initial infection point, then the three vertices of the super triangle are also used as initial infection points.

After all infected triangles are removed, we have constructed our constrained Delaunay triangulation, which can then be used for FEA.

2.4. Mesh Refinement.

The raw result of our constrained Delaunay triangulation algorithm may be sufficient in a few situations, but for most purposes, it returns sub-optimal meshes. Because of this, we must implement a mesh refinement algorithm to improve upon the quality of our mesh.

There are two main mesh refinement algorithms, Ruppert's algorithm [Rup95], and Chew's second algorithm [Che93]. These two algorithms work with a similar principle, but we will focus on Chew's second algorithm. We do this because of the advantages that Chew's second algorithm provides over Ruppert's.

Mesh refinement is required because triangular elements along constrained edges are currently forced to have the constrained edge and one of their edges. This can cause these triangles to be skinny sliver triangles, ones that are undesirable in our final mesh. To fix this we define a notion of "nice" triangles, and insert new points into the mesh until all triangles satisfy our definition of "nice".

Definition 2.2 (*Nice* Triangle)**.** A triangle is *nice* if it is both well-shaped(Def 2.3) and well-sized(Def 2.4).

Definition 2.3 (well-shaped)**.** A triangle is well-shaped if all its angles are greater than or equal to some angle α (commonly $\alpha = 30^\circ$).

Definition 2.4 (well-sized)**.** A triangle is well-sized if the area of the triangle satisfies some user defined grading function g (commonly $g = \text{const}$). This function can use any criteria as long as there exists a value $\delta > 0$ such that any well-shaped(Def 2.3) triangle that fits within a circle of radius δ would satisfy the grading function.

The use of a non-constant grading function is to allow dynamically sized meshes. Such that in areas of interest there can be significantly more refinement, and areas of less interest can be generalized. The restrictions on the function simply stated is that there must be some size of a triangle that will satisfy the grading function everywhere, and that the grading function does not get infinitely strict at any point.

Using these definitions we are able to implement the mesh refinement algorithm. Chew's second algorithm proceeds as follows.

1. Grade any triangles that are currently ungraded. A triangle only passes if it is *nice*(Def 2.2).

2. If all triangles pass then Halt. Otherwise select the larges triangle that fails Δ, and determine its circumcenter c.
3. Traverse the triangulation from any vertex of Δ in the direction of c until either running into a source-edge or finding the triangle containing c.
4. If the triangle containing c was found then insert c into the triangulation, and update the triangulation to be Delaunay. This process is similar to that of sec 2.3. Then go to step 2.
5. If a source edge was encountered, then split the source edge into two equal sized edges and update the triangulation. Let l be the length of the new edges. Delete each circumcenter-vertex(vertices that are not part of the original triangulation) that is within l(line-of-sight distance, where a source-edge means that a vertex is infinitely far away) of the new vertex. Then go to step 2.

Using this algorithm it is possible to construct triangulations with a required minimum angle and force a size function. This provides the ability of guaranteed *nice* triangular meshes for any desired input domain.

Both Chew's second algorithm and Ruppert's algorithm have proven termination for minimum angles below a certain point. For Rupert's algorithm, any minimum angle below $20.7°$ is guaranteed to halt. While Chew's second algorithm is guaranteed to halt for angles below $28.6°$ and will often succeed with angles below $34°$. Because of this improvement on the guaranteed minimum angle is why our focus is on Chew's algorithm for mesh refinement.

Add series of images demonstrating the increase in triangle refinement.

2.5. **Implementation.** There are many implementations of this element discretization process. Several of the original papers provide FORTRAN code, that can be used in implementations. We transcribed Sloan's FORTRAN code for his implementation of Constrained Delaunay Triangulation construction into C++, which is included in [APPENDIX CODE]. However, for mesh refinement, we utilized the open source software Triangle[She96], which implements the Delaunay triangulation construction and mesh refinement.

Add code to the appendix.

3. Linear Systems

(1) Issues
(2) Data Structure
    (a) FULL
    (b) INDEX
    (c) CSR
(3) Multiplication
(4) Krylov Subspaces
    (a) Reasoning
    (b) Definition?
(5) Arnoldi Iteration
(6) GMRES

As the finite element method constructs a global system of equations, we need to implement a method for solving this system of linear equations. However, there are issues with the magnitude of this system. Let us take a sample situation where our mesh has $\sim 100,000$ elements. This would result in a global matrix of $10,000,000,000$ values, which would require $\sim 80Gb$ of memory. Solving this matrix using straight forward gaussian elimination, which has order $\mathcal{O}(n^3)$ time complexity, would require extream amounts of time and memory to compute. Because of these issues we need to utilize methods that allow us to avoid these issues that arise in the brute force method of solving the system of linear equations.

3.1. **Data Structures.**

The first issue that we need to solve is the issue of memory requirements for the large matricies. We can greatly use that fact that the global matrix will be a sparse matrix. This is true by the method of construction of our matrix, most of the elements will be zero.

Using this knowledge we examine three methods of matrix storage. We use a sense of *efficiency* to impy memory efficiency, and the memory usage required to store the matrix.

We assume that the indicies of the matrix are stored as `uint64_t` which requires 8 bytes of memory, and that the values are stored as `double` which requires 8 bytes of memory. We use N to denote the dimenson of the square matrix, and S to represent the percent of elements that are non zero in the matrix.

1. Full matrix storage.
2. Dictionary of Keys.
3. Compressed Row Storage.

3.1.1. *Full Matrix.*

This method simply stores a two dimensional array of `double` values. This means that the memory requirement will be $N^2 \cdot 8$ bytes. Note that this is independent of the sparicty of the matrix, so even if the matrix has only one element the memory usage is still the same as a full matrix.

An example of this is the following

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 0 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 0 & 0 & 5 & 0 \\ 0 & 8 & 0 & 9 & 0 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

For this method we directly store this matrix into memory saving all of the zero elements.

This is the most simplistic method for storing a matrix. It will not suffice for most of our situations, but it is important to have a base line to compare to, to determin the efficiency of our later methods.

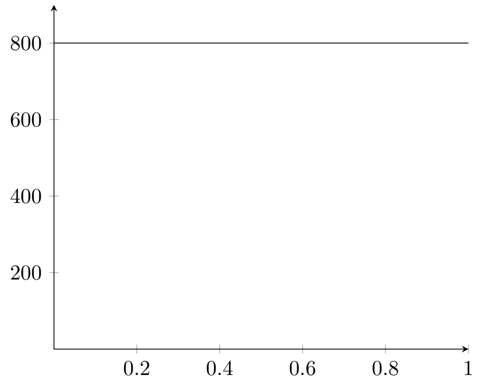

Figure 1. Memory usage for full matrix storage at different percentages of sparicty.

3.1.2. *Dictionary of Keys.* The next logical step for improving our matrix efficiency, is to only store the elements that are not zero. A simple way of doing this is to store the *row* and *column* indecies and the associated value at that point. This means that for each value we store three numbers, but we only store the values that are not zero. This means that this is only effective to an extent, as a full matrix would require three times the memory of the method in section 3.1.1.

Using the same example from above

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 0 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 0 & 0 & 5 & 0 \\ 0 & 8 & 0 & 9 & 0 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix},$$

we implement the DOK method for sparse matrix storage.

$$A = \begin{bmatrix} 0 & 0 & 10 \\ 0 & 4 & -2 \\ 1 & 0 & 3 \\ 1 & 5 & 3 \\ 2 & 1 & 7 \\ 2 & 2 & 8 \\ 2 & 3 & 7 \\ 3 & 0 & 3 \\ 3 & 4 & 5 \\ 4 & 1 & 8 \\ 4 & 3 & 9 \\ 4 & 5 & 13 \\ 5 & 1 & 3 \\ 5 & 4 & 2 \\ 5 & 5 & -1 \end{bmatrix}$$

Where the first column in the row index (0 based), the second column in the column index (0 based), and the third column is the value stored at that position.

We find that this method requires $SN^2 \cdot 24$ bytes. It is clear to see that if S is large, then this method is significantly less effective but if S is small enough, then there is additional efficiency to be gained. We can compute the minimum value of S that would allow for increased efficiency like so

$$\begin{aligned} SN^2 \cdot 24 &= N^2 \cdot 8 \\ S &= \frac{8}{24} \\ S &= \frac{1}{3}. \end{aligned}$$

Thus for any matrix where more than a third of its elements not zero, means that we would be better off using a full matrix storage system.

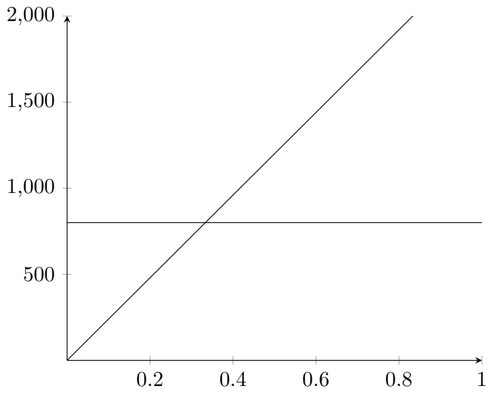

Figure 2. Memory usage for DOK at different percentages of sparicty.

3.1.3. *Compressed Row Storage.* This method take the concept of DOK, and modifies it by the realization that many elements will be on the same row, so we just need to store the number of elements in a row, the elements column index, and the value. This means that for every value, we now only need to store two numbers, and we have a list of row counts that must exists for all number of non-zero elements.

A slight optimization that we introduce, is instead of storing the number of elements in each row, we store the total number of elements so far. This is a slight improvement in the coputational implementation for element access.

This means that for every matrix, we need to store three vectors. `val`, `col_ind`, and `row_ptr`.

Using our example matrix,

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 0 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 0 & 0 & 5 & 0 \\ 0 & 8 & 0 & 9 & 0 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix},$$

we find our three vectors to be

$$\begin{aligned} \texttt{row\_ptr} &= [0, 2, 4, 7, 8, 12, 15] \\ \texttt{col\_ind} &= [\ 0,\ \ 4, 0, 5, 1, 2, 3, 0, 4, 1, 3,\ \ 5, 1, 4,\ \ 5] \\ \texttt{val} &= [10, -2, 3, 3, 7, 8, 7, 3, 5, 8, 9, 13, 4, 2, -1] \end{aligned}$$

Note that we can easily find the number of nonzero elements by reading the last value in `row_ptr`.

We can find the efficiency of this method of matrix storage to be $8N + SN^2 \cdot 16$. Again it is clear that if the matrix is full, then this method is incredibly inefficient. It is also intresting to note that if the matrix is empty, then the DOK method is more efficient. There are two values of S where

1. Dictionary of keys transitions from being more efficent to less.
2. Full matrix transitions from being less efficient to more.

We solve for both of these values of S. First, for when DOK becomes less efficient than CRS.

$$\begin{aligned} 8N + SN^2 \cdot 16 &= SN^2 \cdot 24 \\ 8N &= 8SN^2 \\ S &= \frac{1}{N}. \end{aligned}$$

And when the full matrix becomes more efficent.

$$\begin{aligned} 8N + SN^2 \cdot 16 &= N^2 \cdot 8 \\ 16SN^2 &= 8N^2 - 8N \\ S &= \frac{N-1}{2N} \end{aligned}$$

It is intresting to take notice that the range in which this method is most efficient is dependent on the size of our matrix. Thus we only want to use this value when

$$\frac{1}{N} < S < \frac{N-1}{2N}.$$

Using this relation we can construct a plot for the range of S values in relation of N inwhich this method should be utilized.

Add filled plot of range of optimal S for given N. It looks really cool.

We can use this relation to see that for large N, this method is the most efficent between

$$0 < S < 0.5$$

Because of this asymptotic approch of this range of sparcity for maximum efficiency, we use this method of sparse matrix storeage in our implementation.

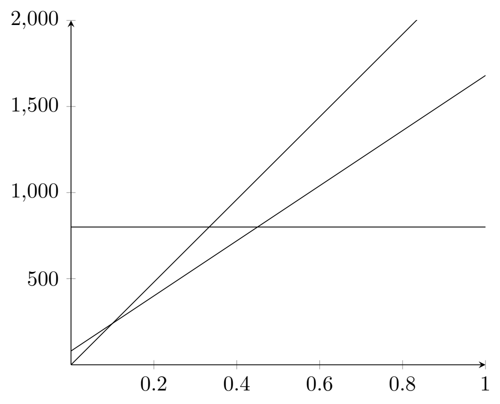

Figure 3. Memory usage for CRS and DOK at different percentages of sparicty.

3.1.4. *Vectors.* We will simmilarly have large vectors, and can implement a similar method for sparse vector storage, but this would be counter productive, as our large vectors are not likly to be sparse. Thus using a sparce represetnation of a vector would actualy consume more memory than the full vector storage method.

References

[BBC+93] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charels Romine, and Hen Van der Vorst. *Templates for the Solution of Linear Systems: Buildign Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 2 edition, 1993.

[Che89] L. Paul Chew. Contrained delaunay triangulations. *Algorithmica*, 4:92–108, 1989.

[Che93] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. *9th Annual Symposium on Computation Geometry*, pages 275–280, 1993.

[Dv08] V. Domiter and B. Žalik. Sweep-line algorithm for constrained delaunay triangulation. *International Journal of Geographical Information Science*, 22(4):449–462, 2008.

[KH15] Dmitri Kuzmin and Jari Hämäläinen. *Finite Element Methods for Computational Fluid Dynamics*. Society for Industrial and Applied Mathematics, Philadelphia, 2015.

[Rup95] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, 1995.

[She96] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[She02] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1-3):21,74, 2002.

[Slo87] S. W. Sloan. A fast algorithm for construction delaunay triangulations in the plane. *Adv. Eng. Software*, 9(1):34–42, 1987.

[Slo93] S. W. Sloan. A fast algorithm for generating constrained delaunay triangulations. *Computers & Structures*, 47(3):441–450, 1993.

[SS86] Youcef Saad and Martin H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *Sci. Stat. Comput.*, 7(3):856–869, 1986.

[Ž05] Borut Žalik. An efficient sweep-line delaunay triangulation algorithm. *Computer-Aided Design*, 37:1027–1038, 2005.